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 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.
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.


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.
Apple A4 also appeared in iPhone 4 (2010).


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.


Also some larger 64-bit A9, A15, A17, and some 64-bit cores are “out of order”: CPU tries to reorder instructions to compensate for dumb compilers.
Apple A4 also appeared in iPhone 4 (2010).


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 insn set:
- 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.
Apple A4 also appeared in iPhone 4 (2010).


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.


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.
Apple A4 also appeared in iPhone 4 (2010).


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, few...
Apple A4 also appeared in iPhone 4 (2010).


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.


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.
Apple A4 also appeared in iPhone 4 (2010).


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.


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.

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.

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. ⇒ Simpler, cheaper, more energy-efficient.

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. ⇒ 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.

Also some larger 64-bit cores.

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

More than one billion Cortex-A7 devices 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:
9 million for OpenSSL 0.9.8k,
4.8 million for OpenSSL 1.0.1c,
3.9 million for OpenSSL 1.0.2j.

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

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];
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];
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:
\[
\begin{align*}
a[0] &= b[0] + c[0]; \\
a[1] &= b[1] + c[1]; \\
\end{align*}
\]
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:
\[ a[0] = b[0] + c[0]; \]
\[ a[1] = b[1] + c[1]; \]
“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:

\[
\begin{align*}
    a[0] &= b[0] + c[0] \\
\end{align*}
\]

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

\[
a[0] = b[0] + c[0];
\]

\[
a[1] = b[1] + c[1];
\]

\[
\]

\[
\]

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.

ADD $\rightarrow$ ADD $\rightarrow$ ADD
Bernstein–Schwabe “NEON crypto” software: new Cortex-A8 speed records for various crypto primitives.

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];

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.

ADD → END

Also logic insns, shifts, etc.

4x a = b - c
is a vector of 4 32-bit subtractions:
  a[0] = b[0] - c[0];
  a[1] = b[1] - c[1];

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

ADD → END

Also logic insns, shifts, etc.
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:
\[
\begin{align*}
    a[0] &= b[0] + c[0]; \\
    a[1] &= b[1] + c[1]; \\
\end{align*}
\]

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.

\[
\begin{align*}
    \text{ADD} & \quad 2 \text{ cycles} \quad \rightarrow \quad \text{ADD} \quad 2 \text{ cycles} \quad \rightarrow \quad \text{ADD}
\end{align*}
\]

Also logic insns, shifts, etc.
NEON instructions

4x \( a = b + c \)

is a vector of 4 32-bit additions:

\[
\begin{align*}
    a[0] &= b[0] + c[0] \\
\end{align*}
\]

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.

\[
\begin{array}{cccccc}
    \text{ADD} & 2 \text{ cycles} & \rightarrow & \text{ADD} & 2 \text{ cycles} & \rightarrow \text{ADD} \\
\end{array}
\]

4x \( a = b - c \)

is a vector of 4 32-bit subtractions:

\[
\begin{align*}
    a[0] &= b[0] - c[0] \\
\end{align*}
\]

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

\[
\begin{array}{cccccc}
    \text{ADD} & \text{2 or 3 cycles} & \rightarrow & \text{SUB} \\
\end{array}
\]

Also logic insns, shifts, etc.
NEON instructions

4x \( a = b + c \)
is a vector of 4 32-bit additions:

\[
\begin{align*}
a[0] &= b[0] + c[0] \\
\end{align*}
\]

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.

\[ \text{ADD} \overset{2 \text{ or } 3 \text{ cycles}}{\rightarrow} \text{SUB} \]

Also logic insns, shifts, etc.

4x \( a = b - c \)
is a vector of 4 32-bit subtractions:

\[
\begin{align*}
a[0] &= b[0] - c[0] \\
\end{align*}
\]
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];

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.

ADD 2 or 3 cycles → SUB

Also logic insns, shifts, etc.

4x a = b - c

is a vector of 4 32-bit subtractions:

a[0] = b[0] - c[0];
a[1] = b[1] - c[1];

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

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

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.
4x $a = b - c$
is a vector of 4 32-bit subtractions:

- $a[0] = b[0] - c[0]$;

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

Also logic insns, shifts, etc.

![Cortex-A8 NEON arithmetic unit](NEON-NEON.png)

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

Also logic insns, shifts, etc.

Stage N8: $c$ is ready.
4x \( a = b + c \)
is a vector of 4 32-bit additions:
\[
\begin{align*}
    a[0] &= b[0] + c[0]; \\
    a[1] &= b[1] + c[1]; \\
\end{align*}
\]
Stage N2: reads \( b \) and \( c \).
Stage N3: performs addition.
Stage N4: \( a \) is ready.

Also logic insns, shifts, etc.

4x \( a = b - c \)
is a vector of 4 32-bit subtractions:
\[
\begin{align*}
    a[0] &= b[0] - c[0]; \\
    a[1] &= b[1] - c[1]; \\
\end{align*}
\]
Stage N1: reads \( c \).
Stage N2: reads \( b \), negates \( c \).
Stage N3: performs addition.
Stage N4: \( a \) is ready.

Also logic insns, shifts, etc.

Multiplication insn:
\[
\begin{align*}
    c[0,1] &= a[0] \text{ signed} \times b[0]; \\
    c[2,3] &= a[1] \text{ signed} \times b[1].
\end{align*}
\]
Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[
\begin{align*}
    c[0,1] &= a[0] \text{ signed} \times b[0]; \\
    c[2,3] &= a[1] \text{ signed} \times b[1].
\end{align*}
\]
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.
$4x \ a = b - c$
is a vector of 4 32-bit subtractions:
\[a[0] = b[0] - c[0];\]

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

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

Also logic insns, shifts, etc.

Multiplication insn:
\[c[0,1] = a[0] \text{ signed} \ast b[0];\]
\[c[2,3] = a[1] \text{ signed} \ast b[1]\]

Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[c[0,1] += a[0] \text{ signed} \ast b[0];\]
\[c[2,3] += a[1] \text{ signed} \ast b[1]\]

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.
8x a = b - c
is a vector of 4 32-bit subtractions:

\[
\begin{align*}
    a[0] &= b[0] - c[0]; \\
    a[1] &= b[1] - c[1]; \\
\end{align*}
\]

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

ADD 2 or 3 cycles \( \rightarrow \) SUB

Also logic insns, shifts, etc.

Multiplication insn:

\[
\begin{align*}
    c[0,1] &= a[0] \text{ signed* } b[0]; \\
    c[2,3] &= a[1] \text{ signed* } b[1]
\end{align*}
\]

Two cycles on Cortex-A8.

Multiply-accumulate insn:

\[
\begin{align*}
    c[0,1] &=+ a[0] \text{ signed* } b[0]; \\
    c[2,3] &=+ a[1] \text{ signed* } b[1]
\end{align*}
\]

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.

9x Typical sequence of three insns:

\[
\begin{align*}
    c[0,1] &= a[0] \text{ signed* } b[0]; \\
    c[2,3] &= a[1] \text{ signed* } b[1] \\
    c[0,1] &=+ e[2] \text{ signed* } f[2]; \\
    c[0,1] &=+ g[0] \text{ signed* } h[2]; \\
    c[2,3] &=+ g[1] \text{ signed* } h[3]
\end{align*}
\]

Cortex-A8 recognizes this pattern.

Reads \( c \) in N6 instead of N3.
8-bit subtractions:
\[ a[0] = b[0] - c[0]; \]
\[ a[1] = b[1] - c[1]; \]

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

\text{ADD} \quad 2 \text{ or } 3 \text{ cycles} \quad \rightarrow \quad \text{SUB}

Also logic insns, shifts, etc.

Multiplication insn:
\[ c[0,1] = a[0] \text{ signed} \ast b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \ast b[1]; \]
Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[ c[0,1] += a[0] \text{ signed} \ast b[0]; \]
\[ c[2,3] += a[1] \text{ signed} \ast b[1]; \]
Also two cycles on Cortex-A8.

Stage N1: reads \( b \).
Stage N2: reads \( a \).
Stage N3: reads \( c \) if accumulate.
Stage N4: \( a \) is ready.
Stage N5: \( c \) is ready.

Typical sequence of three insns:
\[ c[0,1] = a[0] \text{ signed} \ast b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \ast b[1]; \]
\[ c[0,1] += e[2] \text{ signed} \ast f[2]; \]
\[ c[2,3] += e[3] \text{ signed} \ast f[3]; \]
\[ c[0,1] += g[0] \text{ signed} \ast h[2]; \]
\[ c[2,3] += g[1] \text{ signed} \ast h[3]; \]
Cortex-A8 recognizes this pattern.
Reads \( c \) in N6 instead of N3.
Multiplication insn:
\[ c_{0,1} = a_0 \text{ signed* } b_0; \]
\[ c_{2,3} = a_1 \text{ signed* } b_1 \]

Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[ c_{0,1} += a_0 \text{ signed* } b_0; \]
\[ c_{2,3} += a_1 \text{ signed* } b_1 \]

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:
\[ c_{0,1} = a_0 \text{ signed* } b_0; \]
\[ c_{2,3} = a_1 \text{ signed* } b_1 \]
\[ c_{0,1} += e_2 \text{ signed* } f_2; \]
\[ c_{2,3} += e_3 \text{ signed* } f_3 \]
\[ c_{0,1} += g_0 \text{ signed* } h_2; \]
\[ c_{2,3} += g_1 \text{ signed* } h_3 \]

Cortex-A8 recognizes this pattern.
Reads \( c \) in N6 instead of N3.
Multiplication insn:
\[ c[0,1] = a[0] \text{ signed} \times b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \times b[1] \]

Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[ c[0,1] += a[0] \text{ signed} \times b[0]; \]
\[ c[2,3] += a[1] \text{ signed} \times b[1] \]

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:
\[ c[0,1] = a[0] \text{ signed} \times b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \times b[1] \]
\[ c[0,1] += e[2] \text{ signed} \times f[2]; \]
\[ c[0,1] += g[0] \text{ signed} \times h[2]; \]
\[ c[2,3] += g[1] \text{ signed} \times h[3] \]

Cortex-A8 recognizes this pattern. Reads \( c \) in N6 instead of N3.
Typical sequence of three insns:
\[ c[0,1] = a[0] \text{ signed} \times b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \times b[1] \]
\[ c[0,1] += e[2] \text{ signed} \times f[2]; \]
\[ c[0,1] += g[0] \text{ signed} \times h[2]; \]
\[ c[2,3] += g[1] \text{ signed} \times h[3] \]

Cortex-A8 recognizes this pattern.

Reads c in N6 instead of N3.
Multiplication insn:
\[ c_{0,1} = a[0] \text{ signed}^* b[0]; \]
\[ c_{2,3} = a[1] \text{ signed}^* b[1]; \]

Two cycles on Cortex-A8.

Multiply-accumulate insn:
\[ c_{0,1} += a[0] \text{ signed}^* b[0]; \]
\[ c_{2,3} += a[1] \text{ signed}^* b[1]; \]

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:
\[ c_{0,1} = a[0] \text{ signed}^* b[0]; \]
\[ c_{2,3} = a[1] \text{ signed}^* b[1]; \]
\[ c_{0,1} += e[2] \text{ signed}^* f[2]; \]
\[ c_{2,3} += e[3] \text{ signed}^* f[3]; \]
\[ c_{0,1} += g[0] \text{ signed}^* h[2]; \]
\[ c_{2,3} += g[1] \text{ signed}^* h[3]; \]

Cortex-A8 recognizes this pattern.
Reads \( c \) in N6 instead of N3.
Typical sequence of three insns:

\[
\begin{align*}
\text{Stage N1:} & \quad \text{reads } b \\
\text{Stage N2:} & \quad \text{reads } a \\
\text{Stage N3:} & \quad \text{reads } c \text{ if accumulate.}
\end{align*}
\]

Cortex-A8 recognizes this pattern.

Reads \( c \) in N6 instead of N3.
Typical sequence of three insns:
\[
c[0,1] = a[0] \text{ signed} \times b[0];
\]
\[
c[2,3] = a[1] \text{ signed} \times b[1]
\]
\[
c[0,1] += e[2] \text{ signed} \times f[2];
\]
\[
c[2,3] += e[3] \text{ signed} \times f[3]
\]
\[
c[0,1] += g[0] \text{ signed} \times h[2];
\]
\[
c[2,3] += g[1] \text{ signed} \times h[3]
\]

Cortex-A8 recognizes this pattern.
Reads c in N6 instead of N3.
sequence of three insns:

\[
\begin{align*}
&c[0,1] = a[0] \text{ signed} \ast b[0]; \\
&c[2,3] = a[1] \text{ signed} \ast b[1] \\
&c[0,1] += e[2] \text{ signed} \ast f[2]; \\
&c[0,1] += g[0] \text{ signed} \ast h[2]; \\
&c[2,3] += g[1] \text{ signed} \ast h[3]
\end{align*}
\]

Cortex-A8 recognizes this pattern.
Reads \(c\) in N6 instead of N3.

NEON also has load/store insns and permutation insns: e.g.,

\[
r = s[1] \ast t[2] \ast r[2,3]
\]

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

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

Cortex-A7 is different: one unit handling all NEON insns.
Typical sequence of three insns:
\[ c_{0,1} = a_{0} \text{ signed} \times b_{0}; \]
\[ c_{2,3} = a_{1} \text{ signed} \times b_{1}; \]
\[ c_{0,1} += e_{2} \text{ signed} \times f_{2}; \]
\[ c_{2,3} += e_{3} \text{ signed} \times f_{3}; \]
\[ c_{0,1} += g_{0} \text{ signed} \times h_{2}; \]
\[ c_{2,3} += g_{1} \text{ signed} \times h_{3}; \]
Cortex-A8 recognizes this pattern.
Reads \( c \) in N6 instead of N3.

NEON also has load/store insns and permutation insns: \( r = s[1] \times t[2] \times r[2,3] \)
Cortex-A8 has a separate NEON load/store unit that runs in parallel with the NEON arithmetic unit.

Arithmetic is typically the most important bottleneck, so we can often schedule insns to hide loads/stores/perms.
Cortex-A7 is different: one unit handling all NEON insns.
Typical sequence of three insns:
\[
c[0,1] = a[0] \text{ signed} \times b[0];
\]
\[
c[2,3] = a[1] \text{ signed} \times b[1];
\]
\[
c[0,1] += e[2] \text{ signed} \times f[2];
\]
\[
c[2,3] += e[3] \text{ signed} \times f[3];
\]
\[
c[0,1] += g[0] \text{ signed} \times h[2];
\]
\[
c[2,3] += g[1] \text{ signed} \times h[3];
\]
Cortex-A8 recognizes this pattern.

Reads \(c\) in N6 instead of N3.

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.

Arithmetic is typically most important bottleneck: can often schedule insns to hide loads/stores/perms.
<table>
<thead>
<tr>
<th>Time</th>
<th>N1</th>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
<th>N6</th>
<th>N7</th>
<th>N8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

NEON also has load/store insns and permutation insns: e.g.,

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.
<table>
<thead>
<tr>
<th>N2</th>
<th>N3</th>
<th>N4</th>
<th>N5</th>
<th>N6</th>
<th>N7</th>
<th>N8</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>+</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>c</td>
</tr>
</tbody>
</table>

NEON also has load/store insns and permutation insns: e.g.,

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
\[ f = f_0 + 2^{102} f_4 + 2^{204} f_8 + \text{other terms} \]

Unscaled polynomial view:
\[ f \text{ is value at } 2^{25} \text{ of the poly} \]
\[ f_0 t^0 + 2^{102} f_4 t^4 + 2^{204} f_8 t^8 + \text{other terms} \]
<table>
<thead>
<tr>
<th>N4</th>
<th>N5</th>
<th>N6</th>
<th>N7</th>
<th>N8</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>c</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>×</td>
<td>+</td>
<td>c</td>
</tr>
<tr>
<td>×</td>
<td>×</td>
<td>+</td>
<td>+</td>
<td>c</td>
</tr>
</tbody>
</table>

NEON also has load/store insns and permutation insns: e.g.,

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 \(f\):

\[
f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \mod 2^{255} - 19.
\]

Unscaled polynomial view:

\[ f \text{ is value at } 2^{25.5} : \]

\[
f_0 t^0 + 2^{0.5} f_1 t^1 + f_4 t^4 + 2^{0.5} f_5 t^5 + f_8 t^8 + 2^{0.5} f_9 t^9.
\]
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
\[ f = f_0 + 2^{26}f_1 + 2^{51}f_2 + 2^{77}2^{102}f_4 + 2^{128}f_5 + 2^{153}f_6 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9 \mod 2^{255} - 19.\]

Unscaled polynomial view: \(f\) is value at \(2^{25.5}\) of the polynomial
\[ f_0 t^0 + 2^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5} f_4 t^4 + 2^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5} f_8 t^8 + 2^{0.5} f_9 t^9.\]
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
\[ f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \mod 2^{255} - 19. \]

Unscaled polynomial view:
\[ f \] is value at $2^{25.5}$ of the poly
\[ f_0 t^0 + 2^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5} f_3 t^3 + f_4 t^4 + 2^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5} f_7 t^7 + f_8 t^8 + 2^{0.5} f_9 t^9. \]
NEON also has load/store insns and permutation insns: e.g.,
\[ r = s[1] \ t[2] \ r[2,3] \]
Cortex-A8 has a separate load/store unit to hide loads/stores/perms. Cortex-A7 is different: one unit handling all NEON insns.

Unscaled polynomial view:
\[ f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \]
modulo \( 2^{255} - 19 \).

Proof: multiply polys mod \( t^{10} - 19 \).

\[ h \equiv f g \]
\[ h_0 = f_0 g_0 \]
\[ h_1 = f_0 g_1 \]
\[ h_2 = f_0 g_2 \]
\[ h_3 = f_0 g_3 \]
\[ h_4 = f_0 g_4 \]
\[ h_5 = f_0 g_5 \]
\[ h_6 = f_0 g_6 \]
\[ h_7 = f_0 g_7 \]
\[ h_8 = f_0 g_8 \]
\[ h_9 = f_0 g_9 \]

Curve25519 on NEON
Radix \( 2^{25.5} \): Use small integers 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 \( 2^{255} - 19 \).

\[ h = f g \]
\[ h_0 = f_0 g_0 \]
\[ h_1 = f_0 g_1 \]
\[ h_2 = f_0 g_2 \]
\[ h_3 = f_0 g_3 \]
\[ h_4 = f_0 g_4 \]
\[ h_5 = f_0 g_5 \]
\[ h_6 = f_0 g_6 \]
\[ h_7 = f_0 g_7 \]
\[ h_8 = f_0 g_8 \]
\[ h_9 = f_0 g_9 \]

Proof: multiply polys mod \( t^{10} - 19 \).
NEON also has load/store insns
and permutation insns: e.g.,

Cortex-A8 has a... 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
\[
f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 +
2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 +
2^{204} f_8 + 2^{230} f_9 \mod 2^{255} - 19.
\]

Unscaled polynomial view:
$f$ is value at $2^{25.5}$ of the poly
\[
f_0 t^0 + 2^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5} f_3 t^3 +
f_4 t^4 + 2^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5} f_7 t^7 +
f_8 t^8 + 2^{0.5} f_9 t^9.
\]

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

$h \equiv f g \pmod{2^{255} - 19}$

$h_0 = f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 +
38 f_3 g_7 + 19 f_4 g_6 + f_5 g_5 + f_6 g_4 + f_7 g_3 + f_8 g_2 + f_9 g_1 + f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + f_5 g_4 + f_6 g_3 + f_7 g_2 + f_8 g_1 + f_9 g_0$
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

$$f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9 \mod 2^{255} - 19.$$

Unscaled polynomial view:

$f$ is value at $2^{25.5}$ of the poly

$$f_0 t^0 + 2^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5} f_3 t^3 + f_4 t^4 + 2^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5} f_7 t^7 + f_8 t^8 + 2^{0.5} f_9 t^9.$$

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

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

$$h_0 = f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g_7 + 19 f_4 g_6 + 38 f_5 g_5 + 19 f_6 g_4 + 38 f_7 g_3 + 19 f_8 g_2 + 38 f_9 g_1$$

$$h_1 = f_0 g_1 + f_1 g_0 + 19 f_2 g_9 + 19 f_3 g_8 + 19 f_4 g_7 + 19 f_5 g_6 + 19 f_6 g_5 + 19 f_7 g_4 + 19 f_8 g_3 + 19 f_9 g_2$$

$$h_2 = f_0 g_2 + 2 f_1 g_1 + f_2 g_0 + 38 f_3 g_9 + 38 f_4 g_8 + 38 f_5 g_7 + 38 f_6 g_6 + 38 f_7 g_5 + 38 f_8 g_4 + 38 f_9 g_3$$

$$h_3 = f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 38 f_4 g_9 + 38 f_5 g_8 + 38 f_6 g_7 + 38 f_7 g_6 + 38 f_8 g_5 + 38 f_9 g_4$$

$$h_4 = f_0 g_4 + 2 f_1 g_3 + f_2 g_2 + 2 f_3 g_1 + f_4 g_0 + 38 f_5 g_9 + 38 f_6 g_8 + 38 f_7 g_7 + 38 f_8 g_6 + 38 f_9 g_5$$

$$h_5 = f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + f_5 g_0 + 38 f_6 g_9 + 38 f_7 g_8 + 38 f_8 g_7 + 38 f_9 g_6$$

$$h_6 = f_0 g_6 + 2 f_1 g_5 + f_2 g_4 + 2 f_3 g_3 + f_4 g_2 + 2 f_5 g_1 + f_6 g_0 + 38 f_7 g_9 + 38 f_8 g_8 + 38 f_9 g_7$$

$$h_7 = f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + f_5 g_2 + f_6 g_1 + f_7 g_0 + 38 f_8 g_9 + 38 f_9 g_8$$

$$h_8 = f_0 g_8 + 2 f_1 g_7 + f_2 g_6 + 2 f_3 g_5 + f_4 g_4 + 2 f_5 g_3 + f_6 g_2 + 2 f_7 g_1 + f_8 g_0 + 38 f_9 g_9$$

$$h_9 = f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + f_5 g_4 + f_6 g_3 + f_7 g_2 + f_8 g_1 + f_9 g_0$$
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
\[ f = f_0 + 2^{26}f_1 + 2^{51}f_2 + 2^{77}f_3 + 2^{102}f_4 + 2^{128}f_5 + 2^{153}f_6 + 2^{179}f_7 + 2^{204}f_8 + 2^{230}f_9 \mod 2^{255} - 19. \]

Unscaled polynomial view:
\( f \) is value at $2^{25.5}$ of the poly
\[ f_0 t^0 + 2^{0.5}f_1 t^1 + f_2 t^2 + 2^{0.5}f_3 t^3 + f_4 t^4 + 2^{0.5}f_5 t^5 + f_6 t^6 + 2^{0.5}f_7 t^7 + f_8 t^8 + 2^{0.5}f_9 t^9. \]

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

\[ h \equiv f g \pmod{2^{255} - 19} \text{ where} \]
\[ h_0 = f_0 g_0 + 38f_1 g_9 + 19f_2 g_8 + 38f_3 g_7 + 19f_4 g_6 + \]
\[ h_1 = f_0 g_1 + f_1 g_0 + 19f_2 g_9 + 19f_3 g_8 + 19f_4 g_7 + \]
\[ h_2 = f_0 g_2 + 2f_1 g_1 + f_2 g_0 + 38f_3 g_9 + 19f_4 g_8 + \]
\[ h_3 = f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19f_4 g_9 + \]
\[ h_4 = f_0 g_4 + 2f_1 g_3 + f_2 g_2 + f_3 g_1 + f_4 g_0 + \]
\[ h_5 = f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + \]
\[ h_6 = f_0 g_6 + 2f_1 g_5 + f_2 g_4 + 2f_3 g_3 + f_4 g_2 + \]
\[ h_7 = f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + \]
\[ h_8 = f_0 g_8 + 2f_1 g_7 + f_2 g_6 + 2f_3 g_5 + f_4 g_4 + \]
\[ h_9 = f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + \]
519 on NEON

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
\[ h \equiv f g \pmod{2^{255} - 19} \]
where
\[ h_0 = f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + \]
\[ h_1 = f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + \]
\[ h_2 = f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + \]
\[ h_3 = f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + \]
\[ h_4 = f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + \]
\[ h_5 = f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + \]
\[ h_6 = f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + \]
\[ h_7 = f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + \]
\[ h_8 = f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + \]
\[ h_9 = f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + \]

Proof: multiply polys mod \(t^{10} - 19\).
Curve25519 on NEON
Radix 2
25 : 5
: Use small integers
(f0 ; f1 ; f2 ; f3 ; f4 ; f5 ; f6 ; f7 ; f8 ; f ... : 5
f3 t
3
+
f4 t
4
+ 2
0 : 5
f5 t
5
+ f6 t
6
+ 2
0 : 5
f7 t
7
+
f8 t
8
+ 2
0 : 5
f9 t
9
.

\[ h \equiv f g \pmod{2^{255} - 19} \]

\[ h_0 = f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g_7 + 19 f_4 g_6 + 38 f_5 g_5 + 19 f_6 g_4 + 38 f_7 g_3 + 19 f_8 g_2 + 38 f_9 g_1 + 19 f_{10} g_0 \]

\[ h_1 = f_0 g_1 + f_1 g_0 + 19 f_2 g_9 + 19 f_3 g_8 + 19 f_4 g_7 + 19 f_5 g_6 + 19 f_6 g_5 + 19 f_7 g_4 + 19 f_8 g_3 + 19 f_9 g_2 + 19 f_{10} g_1 \]

\[ h_2 = f_0 g_2 + 2 f_1 g_1 + f_2 g_0 + 38 f_3 g_9 + 19 f_4 g_8 + 38 f_5 g_7 + 19 f_6 g_6 + 38 f_7 g_5 + 19 f_8 g_4 + 38 f_9 g_3 + 19 f_{10} g_2 \]

\[ h_3 = f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_9 + 19 f_5 g_8 + 19 f_6 g_7 + 19 f_7 g_6 + 19 f_8 g_5 + 19 f_9 g_4 + 19 f_{10} g_3 \]

\[ h_4 = f_0 g_4 + 2 f_1 g_3 + f_2 g_2 + 2 f_3 g_1 + f_4 g_0 + 38 f_5 g_9 + 19 f_6 g_8 + 38 f_7 g_7 + 19 f_8 g_6 + 19 f_9 g_5 + 19 f_{10} g_4 \]

\[ h_5 = f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + f_5 g_0 + 19 f_6 g_9 + 19 f_7 g_8 + 19 f_8 g_7 + 19 f_9 g_6 + 19 f_{10} g_5 \]

\[ h_6 = f_0 g_6 + 2 f_1 g_5 + f_2 g_4 + 2 f_3 g_3 + f_4 g_2 + 2 f_5 g_1 + f_6 g_0 + 38 f_7 g_9 + 19 f_8 g_8 + 19 f_9 g_7 + 19 f_{10} g_6 \]

\[ h_7 = f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + f_5 g_2 + f_6 g_1 + f_7 g_0 \]

\[ h_8 = f_0 g_8 + 2 f_1 g_7 + f_2 g_6 + 2 f_3 g_5 + f_4 g_4 + 2 f_5 g_3 + f_6 g_2 + 2 f_7 g_1 \]

\[ h_9 = f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + f_5 g_4 + f_6 g_3 + f_7 g_2 \]

Proof: multiply polys mod \( t^{10} - 19 \).
Curve25519 on NEON
Radix 2

\[ f(0) \equiv f_0 \mod 2^{255} - 19 \]

\[ h \equiv f g \mod 2^{255} - 19 \]

\[ h_0 = f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + \cdots \]

\[ h_9 = f_0g_9 + 38f_1g_8 + 38f_2g_7 + 38f_3g_6 + 38f_4g_5 + \cdots \]

Proof: multiply polys \( f \) and \( g \) \( \mod t^{10} - 19 \).
\[ h \equiv f g \pmod{2^{255} - 19} \] where

\[

t_0 = f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + \\
38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,
\]

\[

t_1 = f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + \\
19f_5g_6 + 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,
\]

\[

t_2 = f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + \\
38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,
\]

\[

t_3 = f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + \\
19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,
\]

\[

t_4 = f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + \\
38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,
\]

\[

t_5 = f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + \\
f_5g_0 + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,
\]

\[

t_6 = f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + \\
2f_5g_1 + f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,
\]

\[

t_7 = f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + \\
f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_9 + 19f_9g_8,
\]

\[

t_8 = f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + \\
2f_5g_3 + f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_9,
\]

\[

t_9 = f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + \\
f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\]

Proof: multiply polys mod \( t^{10} - 19 \).
(mod $2^{255} - 19$) where

\[ h \equiv f g \pmod{2^{255} - 19} \]

\[ h_0 = f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g_7 + 19 f_4 g_6 + \]
\[ f_1 g_0 + 19 f_2 g_9 + 19 f_3 g_8 + 19 f_4 g_7 + \]
\[ 2 f_1 g_1 + f_2 g_0 + 38 f_3 g_9 + 19 f_4 g_8 + \]
\[ f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_9 + \]
\[ 2 f_1 g_3 + f_2 g_2 + 2 f_3 g_1 + f_4 g_0 + \]
\[ f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + \]
\[ 2 f_1 g_5 + f_2 g_4 + 2 f_3 g_3 + f_4 g_2 + \]
\[ f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + \]
\[ 2 f_1 g_7 + f_2 g_6 + 2 f_3 g_5 + f_4 g_4 + \]
\[ f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + \]

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

Each $h_i$ is a sum of ten products after precomputation of $2 f_1, 2 f_3, 2 f_5, 2 f_7, 2 f_9, 19 g_1, 19 g_3, 19 g_5, 19 g_7, 19 g_9$.

Each $h_i$ for subsequent multiplication.

(Analyze this very carefully: sizes of $h_i$ are too large under reasonable limits on $t$.)

See 2011 Brumley–Page–Barbosa–Vercauteren bugs can slip past most tests!

$h_0, h_1, \ldots$ are too large for subsequent multiplication.
\begin{align*}
9f_2g_8 + 38f_3g_7 + 19f_4g_6 + \\
9f_2g_9 + 19f_3g_8 + 19f_4g_7 + \\
f_2g_0 + 38f_3g_9 + 19f_4g_8 + \\
f_2g_1 + f_3g_0 + 19f_4g_9 + \\
f_2g_2 + 2f_3g_1 + f_4g_0 + \\
f_2g_3 + f_3g_2 + f_4g_1 + \\
f_2g_4 + 2f_3g_3 + f_4g_2 + \\
f_2g_5 + f_3g_4 + f_4g_3 + \\
f_2g_6 + 2f_3g_5 + f_4g_4 + \\
f_2g_7 + f_3g_6 + f_4g_5 + \\
38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,
\end{align*}

\begin{align*}
9f_2g_9 + 19f_3g_8 + 19f_4g_7 + \\
f_2g_0 + 38f_3g_9 + 19f_4g_8 + \\
f_2g_1 + f_3g_0 + 19f_4g_9 + \\
f_2g_2 + 2f_3g_1 + f_4g_0 + \\
f_2g_3 + f_3g_2 + f_4g_1 + \\
f_2g_4 + 2f_3g_3 + f_4g_2 + \\
f_2g_5 + f_3g_4 + f_4g_3 + \\
f_2g_6 + 2f_3g_5 + f_4g_4 + \\
f_2g_7 + f_3g_6 + f_4g_5 + \\
38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,
\end{align*}

\begin{align*}
19f_5g_6 + 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,
\end{align*}

\begin{align*}
38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,
\end{align*}

\begin{align*}
19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,
\end{align*}

\begin{align*}
38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,
\end{align*}

\begin{align*}
f_5g_0 + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,
\end{align*}

\begin{align*}
2f_5g_1 + f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,
\end{align*}

\begin{align*}
f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_9 + 19f_9g_8,
\end{align*}

\begin{align*}
2f_5g_3 + f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_9,
\end{align*}

\begin{align*}
f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\end{align*}
where
\[ h ≡ f g \pmod{2^{255} - 19} \]
where \( h \equiv f g \pmod{2^{255} - 19} \).

Proof: multiply polys mod \( t^{10} - 19 \).

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.
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.
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 $h_0 \rightarrow h_1$: i.e., replace $(h_0; h_1)$ with $(h_0 \mod 2^{26}; h_1 + \lfloor h_0 / 2^{26} \rfloor)$. 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.
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 $h_0 \rightarrow h_1$: i.e., replace $(h_0, h_1)$ with $(h_0 \mod 2^{26}, h_1 + \lfloor h_0 / 2^{26} \rfloor)$.

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 $ab + c$ etc.

Some things we haven't tried yet:

• Mix signed, unsigned carries.
• Interleave reduction, carrying.
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 $h_0 \rightarrow h_1$: i.e., replace $(h_0, h_1)$ with $(h_0 \mod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$. This makes $h_0$ small.

Similarly for other $h_i$.

Eventually all $h_i$ are small enough.

We actually use signed coefficients. 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.
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 $h_0 \rightarrow h_1$: i.e., replace $(h_0, h_1)$ with $(h_0 \mod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$. 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.
Each $h_i$ is a sum of ten products after precomputation of $2f_1, 2f_3, 2f_5, 2f_7, 2f_9, g_2, \ldots, 19g_9$.

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

Take 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 $h_0 \rightarrow h_1$: i.e.,
replace $(h_0, h_1)$ with
$(h_0 \mod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$.
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 $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$ have long chain of dependencies.
Each $h_i$ is a sum of ten products after precomputation of $2f_1, 2f_3, 2f_5, 2f_7, 2f_9, 19g_1, 19g_2, \ldots$.

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

Carefully: bugs can slip past most tests! See 2011 Brumley–Page–Barbosa–Vercauteren and several recent OpenSSL bugs.)

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 $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$ have long chain of dependencies.
Each $h_i$ is a sum of ten products after precomputation of $2^{f_1}; 2^{f_3}; 2^{f_5}; 2^{f_7}; 2^{f_9}; 19^{g_1}; 19^{g_2}; \ldots$.

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 $h_0 \rightarrow h_1$: i.e., replace $(h_0, h_1)$ with $(h_0 \mod 2^{26}, h_1 + \lceil h_0/2^{26} \rceil)$. 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 $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$ have long chain of dependencies.
Carry $h_0 \to h_1$: i.e., replace $(h_0, h_1)$ with $(h_0 \mod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$. 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.
Carry $h_0 \rightarrow h_1$: i.e.,
$(h_0, h_1)$ with
$2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor$.

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 $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$

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

Alternative: carry
$h_0 \rightarrow h_1$ and $h_5 \rightarrow h_6$;
$h_1 \rightarrow h_2$ and $h_6 \rightarrow h_7$;
$h_2 \rightarrow h_3$ and $h_7 \rightarrow h_8$;
$h_3 \rightarrow h_4$ and $h_8 \rightarrow h_9$;
$h_4 \rightarrow h_5$ and $h_9 \rightarrow h_0$;
$h_5 \rightarrow h_6$ and $h_0 \rightarrow h_1$.

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

Now much easier
to find independent insns
for CPU to handle in parallel.
Carry $h_0 \rightarrow h_1$: i.e., replace $(h_0; h_1)$ with $(h_0 \mod 2^{26}; h_1 + \hat{h}_0 = 2^{26}\hat{h})$.
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) 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 $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$
have long chain of dependencies.

Alternative: carry $h_0 \rightarrow h_1$ and $h_5 \rightarrow h_6$; $h_1 \rightarrow h_2$ and $h_6 \rightarrow h_7$; $h_2 \rightarrow h_3$ and $h_7 \rightarrow h_8$; $h_3 \rightarrow h_4$ and $h_8 \rightarrow h_9$; $h_4 \rightarrow h_5$ and $h_9 \rightarrow h_0$; $h_5 \rightarrow h_6$ and $h_0 \rightarrow h_1$.

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

Now much easier to find independent insns for CPU to handle.
Carry $h_0 \rightarrow h_1$ : i.e.,
replace $(h_0; h_1)$ with
$(h_0 \mod 2^{26}; h_1 + h_0 = 2^{26} \cdot h_0)$.
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 $ab + c$ 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 $h_0 \rightarrow h_1 \rightarrow$
$h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow$
$h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$
have long chain of dependencies.

Alternative: carry
$h_0 \rightarrow h_1$ and $h_5 \rightarrow h_6$;
$h_1 \rightarrow h_2$ and $h_6 \rightarrow h_7$;
$h_2 \rightarrow h_3$ and $h_7 \rightarrow h_8$;
$h_3 \rightarrow h_4$ and $h_8 \rightarrow h_9$;
$h_4 \rightarrow h_5$ and $h_9 \rightarrow h_0$;
$h_5 \rightarrow h_6$ and $h_0 \rightarrow h_1$.
12 carries instead of 11,
but latency is much smaller.
Now much easier
to find independent insns
for CPU to handle in parallel.
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.
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.

Alternative: carry
\[ h_0 \rightarrow h_1 \text{ and } h_5 \rightarrow h_6; \]
\[ h_1 \rightarrow h_2 \text{ and } h_6 \rightarrow h_7; \]
\[ h_2 \rightarrow h_3 \text{ and } h_7 \rightarrow h_8; \]
\[ h_3 \rightarrow h_4 \text{ and } h_8 \rightarrow h_9; \]
\[ h_4 \rightarrow h_5 \text{ and } h_9 \rightarrow h_0; \]
\[ h_5 \rightarrow h_6 \text{ and } h_0 \rightarrow h_1. \]

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.
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 when the first insn is in progress. Sometimes helps to adjust higher-level computations. Example: carries $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_0 \rightarrow h_1$ of dependencies. Alternative: carry $h_0 \rightarrow h_1$ and $h_5 \rightarrow h_6$; $h_1 \rightarrow h_2$ and $h_6 \rightarrow h_7$; $h_2 \rightarrow h_3$ and $h_7 \rightarrow h_8$; $h_3 \rightarrow h_4$ and $h_8 \rightarrow h_9$; $h_4 \rightarrow h_5$ and $h_9 \rightarrow h_0$; $h_5 \rightarrow h_6$ and $h_0 \rightarrow h_1$. 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.
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.

Example: carries $h_0 \rightarrow h_1 \rightarrow h_2 \rightarrow h_3 \rightarrow h_4 \rightarrow h_5 \rightarrow h_6 \rightarrow h_7 \rightarrow h_8 \rightarrow h_9 \rightarrow h_{10} \rightarrow h_{11}$

have long chain of dependencies.

Alternative: carry

$h_0 \rightarrow h_1$ and $h_5 \rightarrow h_6$;  
$h_1 \rightarrow h_2$ and $h_6 \rightarrow h_7$;  
$h_2 \rightarrow h_3$ and $h_7 \rightarrow h_8$;  
$h_3 \rightarrow h_4$ and $h_8 \rightarrow h_9$;  
$h_4 \rightarrow h_5$ and $h_9 \rightarrow h_{10}$;  
$h_5 \rightarrow h_6$ and $h_{10} \rightarrow h_{11}$.

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.
Alternative: carry

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

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.
Alternative: carry
and $h_5 \rightarrow h_6$;
and $h_6 \rightarrow h_7$;
and $h_7 \rightarrow h_8$;
and $h_8 \rightarrow h_9$;
and $h_9 \rightarrow h_0$;
and $h_0 \rightarrow h_1$.

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:
$h = f g$;
$h' = f' g'$.

Vectorize across those mults.

Merge $f_0$, $f_1$, ..., $f_9$,
and $f'_0$, $f'_1$, ..., $f'_9$ 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.,
$c[0,1] = a[0] \text{ signed} \ast b[0]$;
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 these:

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.,

\[
c[0,1] = a[0] \text{ signed} \times b[0];
\]

\[
c[2,3] = a[1] \text{ signed} \times b[1];
\]
19

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.

20

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.,
$c[0,1] = a[0] \text{ signed} \ast b[0]$
$c[2,3] = a[1] \text{ signed} \ast b[1]$
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:
$h = f \ g; h' = f' \ g'$.

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* $b[0]$;
Major challenge: vectorization.

\[ a = b + c \]

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.,

\[ c[0,1] = a[0] \text{ signed} \ast b[0]; \]

\[ c[2,3] = a[1] \text{ signed} \ast b[1] \]
Often higher-level operations do a pair of mults in parallel:
\[ h = fg; \quad h' = f'g'. \]

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.,
\[ \begin{array}{l}
  c[0,1] = a[0] \text{ signed* } b[0]; \\
  c[2,3] = a[1] \text{ signed* } b[1]
\end{array} \]
Major challenge: vectorization.

\[ 4\times a = b + c \]
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:
\[ h = f g; h' = f' g'. \]
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.,
\[ c[0,1] = a[0] \text{ signed} \times b[0]; \]
\[ c[2,3] = a[1] \text{ signed} \times b[1] \]

Example: Recall \( C = X_1 \cdot X_2; D = Y_1 \cdot Y_2 \) inside point-addition formulas for Edwards curves.
Often higher-level operations do a pair of mults in parallel: $h = f g; h' = f' g'$.

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.,

\[
\begin{align*}
    c[0,1] &= a[0] \text{ signed* } b[0]; \\
    c[2,3] &= a[1] \text{ signed* } b[1]
\end{align*}
\]

Example: Recall $C = X_1 \cdot X_2; D = Y_1 \cdot Y_2$ inside point-addition formulas for Edwards curves.
Often higher-level operations do a pair of mults in parallel:
\[ h = f g; \quad h' = f' g'. \]

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.,
\[
\begin{align*}
c[0,1] &= a[0] \text{ signed}\star b[0]; \\
c[2,3] &= a[1] \text{ signed}\star b[1]
\end{align*}
\]

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

Example: Can compute \( 2P, 3P, 4P, 5P, 6P, 7P \) as
\[
\begin{align*}
2P &= P + P; \\
3P &= 2P + P \quad \text{and} \quad 4P = 2P + 2P; \\
5P &= 4P + P \quad \text{and} \quad 6P = 3P + 3P \\
\quad \text{and} \quad 7P &= 4P + 3P.
\end{align*}
\]
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.,
\[
\begin{align*}
c[0,1] & = a[0] \text{ signed} \ast b[0]; \\
c[2,3] & = a[1] \text{ signed} \ast b[1]
\end{align*}
\]

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

Example: Can compute
\[
\begin{align*}
2P, \ 3P, \ 4P, \ 5P, \ 6P, \ 7P \ &= \ P + P; \\
3P &= 2P + P \text{ and } 4P = 2P + 2P; \\
5P &= 4P + P \text{ and } 6P = 3P + 3P \\
&\text{and } 7P = 4P + 3P.
\end{align*}
\]

Example: Typical algorithms for fixed-base scalmult
have many parallel point adds.
Often higher-level operations do a pair of mults in parallel:
\[ h = f g; \quad h' = f' g'. \]

Vectorize across those mults.

Mere across those mults.

\[ f_0, f_1, \ldots, f_9 \quad f'_0, f'_1, \ldots, f'_9 \]

Form vectors \((f_i, f'_i)\).

\((g_i, g'_i)\).

Then compute \((h_i, h'_i)\).

Computation fits naturally into NEON insns: e.g.,
\[
\begin{align*}
c[0,1] &= a[0] \text{ signed} \ast b[0]; \\
c[2,3] &= a[1] \text{ signed} \ast b[1].
\end{align*}
\]

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

Example: Can compute
\[
\begin{align*}
2P, 3P, 4P, 5P, 6P, 7P & \quad \text{as} \\
2P &= P + P; \\
3P &= 2P + P \quad \text{and} \quad 4P = 2P + 2P; \\
5P &= 4P + P \quad \text{and} \quad 6P = 3P + 3P \quad \text{and} \quad 7P = 4P + 3P.
\end{align*}
\]

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

Example: A busy server with a backlog of scalarmults can vectorize across them.
Often higher-level operations do a pair of mults in parallel:

\[ h = f \cdot g; \quad h' = f' \cdot g'. \]

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.,

\[ \text{c}[0,1] = \text{a}[0] \times \text{b}[0]; \]
\[ \text{c}[2,3] = \text{a}[1] \times \text{b}[1]; \]

Example: Recall \( C = X_1 \cdot X_2; \quad D = Y_1 \cdot Y_2 \)

inside point-addition formulas for Edwards curves.

Example: Can compute \( 2P, 3P, 4P, 5P, 6P, 7P \) as

\[ 2P = P + P; \]
\[ 3P = 2P + P \quad \text{and} \quad 4P = 2P + 2P; \]
\[ 5P = 4P + P \quad \text{and} \quad 6P = 3P + 3P \quad \text{and} \quad 7P = 4P + 3P. \]

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

Example: A busy server with a backlog of scalarmults can vectorize across them.
Often higher-level operations do a pair of mults in parallel:
\[ h = f \circ g; \quad h' = f' \circ g'. \]

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.,
\[
\begin{align*}
c[0,1] &= a[0] \text{ signed} \ast b[0]; \\
c[2,3] &= a[1] \text{ signed} \ast b[1].
\end{align*}
\]

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
\[
\begin{align*}
2P &= P + P; \\
3P &= 2P + P \quad \text{and} \quad 4P = 2P + 2P; \\
5P &= 4P + P \quad \text{and} \quad 6P = 3P + 3P \\
\text{and} \quad 7P &= 4P + 3P.
\end{align*}
\]

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

Example: A busy server with a backlog of scalarmults can vectorize across them.
Example: Recall
\[ C = X_1 \cdot X_2; \quad D = Y_1 \cdot Y_2 \]
inside point-addition formulas for Edwards curves.

Example: Can compute
\[ 2P, 3P, 4P, 5P, 6P, 7P \]
as
\[ 2P = P + P; \]
\[ 3P = 2P + P \text{ and } 4P = 2P + 2P; \]
\[ 5P = 4P + P \text{ and } 6P = 3P + 3P \]
and \[ 7P = 4P + 3P. \]

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

Example: A busy server with a backlog of scalar multiplications can vectorize across them.
Example: Recall
\[ C = X_1 \cdot X_2; \quad D = Y_1 \cdot Y_2 \]
inside point-addition formulas for Edwards curves.

Example: Can compute
\( 2P, 3P, 4P, 5P, 6P, 7P \) as
\[ 2P = P + P; \]
\[ 3P = 2P + P \text{ and } 4P = 2P + 2P; \]
\[ 5P = 4P + P \text{ and } 6P = 3P + 3P \text{ and } 7P = 4P + 3P. \]

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.

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

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

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

Some field ops are hard to pair
inside a single scalar mulmul.

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

Easy, constant time:
$$Z^{-1} = Z^{p-2}.$$  

$$z_2 = z_1^2,$$
$$z_8 = z_2^2,$$
$$z_9 = z_1z_8,$$
$$z_{11} = z_2z_9,$$
$$z_{22} = z_{11}^2,$$
$$z_{5_0} = z_9z_{22},$$
$$z_{10_5} = z_{5_0}^2.$$
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 $4P = 2P + 2P$; $6P = 3P + 3P$; $7P = 4P + 3P$. 

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.

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: $Z^{-1} = Z^{p-2}$.

$11M + 254S$ for $p = 2^{255} - 19$.

$z_2 = z_1^{2^1}$
$z_8 = z_2^{2^2}$
$z_9 = z_1 * z_8$
$z_11 = z_2 * z_9$
$z_22 = z_{11}^{2^1}$
$z_{5_0} = z_9 * z_{22}$
$z_{10_5} = z_{5_0}^{2^5}$
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: $Z^{-1} = Z^{p-2}$.

$11M + 254S$ for $p = 2^{255} - 19$.

$z_2 = z_1^2^1$
$z_8 = z_2^2^2$
$z_9 = z_1 * z_8$
$z_{11} = z_2 * z_9$
$z_{22} = z_{11}^2^1$
$z_{5_0} = z_9 * z_{22}$
$z_{10_5} = z_{5_0}^2^5$
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: $Z^{-1} = Z^{p-2}$.

$11M + 254S$ for $p = 2^{255} - 19$:

\[
\begin{align*}
z_2 &= z_1^{2^1} \\
z_8 &= z_2^{2^2} \\
z_9 &= z_1 \times z_8 \\
z_{11} &= z_2 \times z_9 \\
z_{22} &= z_{11}^{2^1} \\
z_{5_0} &= z_9 \times z_{22} \\
z_{10_5} &= z_{5_0}^{2^5} 
\end{align*}
\]
Example: A busy server with a backlog of scalar multiplies can vectorize across them. Beware a disadvantage of vectorizing across two MulMul: 
\[ f, f', g, g', h, h' \]
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 scalar multiply.
Example: At end of ECDH, convert fraction \((X : Z)\) into \(Z^{-1}X \in \{0, 1, \ldots, p - 1\}\).

Easy, constant time: \(Z^{-1} = Z^{p-2}\).
\(11M + 254S\) for \(p = 2^{255} - 19\):
\[
\begin{align*}
z_2 &= z_1^2^1 \\
z_8 &= z_2^2^2 \\
z_9 &= z_1*z_8 \\
z_{11} &= z_2*z_9 \\
z_{22} &= z_{11}^2^1 \\
z_{5_0} &= z_9*z_{22} \\
z_{10_5} &= z_{5_0}^2^5 \\
z_{10_0} &= z_{10_5}*z_{5_0} \\
z_{20_10} &= z_{10_0}^2^{10} \\
z_{20_0} &= z_{20_10}*z_{10_0} \\
z_{40_20} &= z_{20_0}^2^{20} \\
z_{40_0} &= z_{40_20}*z_{20_0} \\
z_{50_10} &= z_{40_0}^2^{10} \\
z_{50_0} &= z_{50_10}*z_{10_0} \\
z_{100_50} &= z_{50_0}^2^{50} \\
z_{100_0} &= z_{100_50}*z_{50_0} \\
z_{200_100} &= z_{100_0}^2^{100} \\
z_{200_0} &= z_{200_100}*z_{100_0} \\
z_{200_0} &= z_{200_100}*z_{100_0} \\
z_{250_50} &= z_{200_0}^2^{50} \\
z_{250_0} &= z_{250_50}*z_{50_0} \\
z_{255_5} &= z_{250_0}^2^{5} \\
z_{255_21} &= z_{255_5}*z_{11}
\end{align*}
\]
Example: A busy server with a backlog of scalarmults can vectorize across them.

Beware a disadvantage of vectorizing across two multiplies: $256$-bit inputs $h, h'$ occupy at least $1536$ bits, leaving very little room for registers.

We use some loads and stores inside vectorized multiplier operations.

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: $Z^{-1} = Z^{p-2}$.

11M + 254S for $p = 2^{255} - 19$:

\[
\begin{align*}
z_2 & = z_1^2 \\
z_8 & = z_2^2 \\
z_9 & = z_1z_8 \\
z_{11} & = z_2z_9 \\
z_{22} & = z_{11}^2 \\
z_{5_0} & = z_9z_{22} \\
z_{10_5} & = z_{5_0}^2 \\
z_{10_0} & = z_{10_5}z_{5_0} \\
z_{20_10} & = z_{10_0}^2 \\
z_{20_0} & = z_{20_10}z_{10_0} \\
z_{40_20} & = z_{20_0}^2 \\
z_{40_0} & = z_{40_20}z_{20_0} \\
z_{50_10} & = z_{40_0}^2 \\
z_{50_0} & = z_{50_10}z_{10_0} \\
z_{100_50} & = z_{50_0}^2 \\
z_{100_0} & = z_{100_50}z_{50_0} \\
z_{200_100} & = z_{100_0}^2 \\
z_{200_0} & = z_{200_100}z_{100_0} \\
z_{250_50} & = z_{200_0}^2 \\
z_{250_0} & = z_{250_50}z_{50_0} \\
z_{255_5} & = z_{250_0}^2 \\
z_{255_21} & = z_{255_5}z_{11}
\end{align*}
\]
Some field ops are hard to pair inside a single scalar mult.

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

Easy, constant time: \(Z^{-1} = Z^{p-2}\).

**11M + 254S** for \(p = 2^{255} - 19\):

\[
\begin{align*}
z_2 &= z_1^{2^1} \\
z_8 &= z_2^{2^2} \\
z_9 &= z_1 z_8 \\
z_{11} &= z_2 z_9 \\
z_{22} &= z_{11}^{2^1} \\
z_{5_0} &= z_9 z_{22} \\
z_{10_5} &= z_{5_0}^{2^5} \\
z_{10_0} &= z_{10_5} z_{5_0} \\
z_{20_10} &= z_{10_0}^{2^{10}} \\
z_{20_0} &= z_{20_10} z_{10_0} \\
z_{40_20} &= z_{20_0}^{2^{20}} \\
z_{40_0} &= z_{40_20} z_{20_0} \\
z_{50_10} &= z_{40_0}^{2^{10}} \\
z_{50_0} &= z_{50_10} z_{10_0} \\
z_{100_50} &= z_{50_0}^{2^{50}} \\
z_{100_0} &= z_{100_50} z_{50_0} \\
z_{200_100} &= z_{100_0}^{2^{100}} \\
z_{200_0} &= z_{200_100} z_{100_0} \\
z_{250_50} &= z_{200_0}^{2^{50}} \\
z_{250_0} &= z_{250_50} z_{50_0} \\
z_{255_5} &= z_{250_0}^{2^5} \\
z_{255_21} &= z_{255_5} z_{11}
\end{align*}
\]
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: \(Z^{-1} = Z^{p-2}\).

11M + 254S for \(p = 2^{255} - 19\):

\[
\begin{align*}
z_2 &= z_1^2^1 \\
z_8 &= z_2^2^2 \\
z_9 &= z_1 \cdot z_8 \\
z_{11} &= z_2 \cdot z_9 \\
z_{22} &= z_{11}^2^1 \\
z_{5_0} &= z_9 \cdot z_{22} \\
z_{10_5} &= z_{5_0}^2^5 \\
z_{10_0} &= z_{10_5} \cdot z_{5_0} \\
z_{20_10} &= z_{10_0}^2^10 \\
z_{20_0} &= z_{20_10} \cdot z_{10_0} \\
z_{40_20} &= z_{20_0}^2^20 \\
z_{40_0} &= z_{40_20} \cdot z_{20_0} \\
z_{50_10} &= z_{40_20}^2^10 \\
z_{50_0} &= z_{50_10} \cdot z_{10_0} \\
z_{100_50} &= z_{50_0}^2^50 \\
z_{100_0} &= z_{100_50} \cdot z_{50_0} \\
z_{200_100} &= z_{100_0}^2^100 \\
z_{200_0} &= z_{200_100} \cdot z_{100_0} \\
z_{250_50} &= z_{200_0}^2^50 \\
z_{250_0} &= z_{250_50} \cdot z_{50_0} \\
z_{255_5} &= z_{250_0}^2^5 \\
z_{255_21} &= z_{255_5} \cdot z_{11}
\end{align*}
\]
Some field ops are hard to pair inside a single scalar mult.

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

Constant time: \(Z^{-1} = Z^{p-2}\).

254S for \(p = 2^{255} - 19:\)

\[
\begin{align*}
z_{10 \cdot 0} &= z_{10 \cdot 5} \cdot z_{5 \cdot 0} \\
z_{20 \cdot 10} &= z_{10 \cdot 0} \cdot 2 \cdot 10 \\
z_{20 \cdot 0} &= z_{20 \cdot 10} \cdot z_{10 \cdot 0} \\
z_{40 \cdot 20} &= z_{20 \cdot 0} \cdot 2 \cdot 20 \\
z_{40 \cdot 0} &= z_{40 \cdot 20} \cdot z_{20 \cdot 0} \\
z_{50 \cdot 10} &= z_{40 \cdot 0} \cdot 2 \cdot 10 \\
z_{50 \cdot 0} &= z_{50 \cdot 10} \cdot z_{10 \cdot 0} \\
z_{100 \cdot 50} &= z_{50 \cdot 0} \cdot 2 \cdot 50 \\
z_{100 \cdot 0} &= z_{100 \cdot 50} \cdot z_{50 \cdot 0} \\
z_{200 \cdot 100} &= z_{100 \cdot 0} \cdot 2 \cdot 100 \\
z_{200 \cdot 0} &= z_{200 \cdot 100} \cdot z_{100 \cdot 0} \\
z_{250 \cdot 50} &= z_{200 \cdot 0} \cdot 2 \cdot 50 \\
z_{250 \cdot 0} &= z_{250 \cdot 50} \cdot z_{50 \cdot 0} \\
z_{255 \cdot 5} &= z_{250 \cdot 0} \cdot 2 \cdot 5 \\
z_{255 \cdot 21} &= z_{255 \cdot 5} \cdot z_{11}
\end{align*}
\]

Can still vectorize inside a single field op.

Strategy in our software:

50 mul insns starting from 
\((f_{0}, 2f_{1}), (f_{2}, 2f_{3}), (f_{4}, 2f_{5}), (f_{6}, 2f_{7}), (f_{8}, 2f_{9})\); 
\((g_{0}, g_{1}), (g_{2}, g_{3}), (g_{4}, g_{5}), (g_{6}, g_{7}), (g_{8}, g_{9})\); 
\((19g_{2}, 19g_{3}), (19g_{4}, 19g_{5}), (19g_{6}, 19g_{7}), (19g_{8}, 19g_{9})\).

Change carry pattern to vectorize, e.g., \((h_{0}, h_{4}) \rightarrow (h_{1}, h_{5})\).
Some field ops are hard to pair inside a single scalar mult.
Example: At end of ECDH, convert fraction \((X : Z)\) into
\[Z^{-1} X \in \{0, 1, \ldots, p - 1\}\].
Here: \(Z^{-1} = Z^{p-2}\).
\(p = 2^{255} - 19:\)
\[z_{10, 0} = z_{10, 5} \times z_{5, 0}\]
\[z_{20, 10} = z_{10, 0} \times 2^{10}\]
\[z_{20, 0} = z_{20, 10} \times z_{10, 0}\]
\[z_{40, 20} = z_{20, 0} \times 2^{20}\]
\[z_{40, 0} = z_{40, 20} \times z_{20, 0}\]
\[z_{50, 10} = z_{40, 0} \times 2^{10}\]
\[z_{50, 0} = z_{50, 10} \times z_{10, 0}\]
\[z_{100, 50} = z_{50, 0} \times 2^{50}\]
\[z_{100, 0} = z_{100, 50} \times z_{50, 0}\]
\[z_{200, 100} = z_{100, 0} \times 2^{100}\]
\[z_{200, 0} = z_{200, 100} \times z_{100, 0}\]
\[z_{250, 50} = z_{200, 0} \times 2^{50}\]
\[z_{250, 0} = z_{250, 50} \times z_{50, 0}\]
\[z_{255, 5} = z_{250, 0} \times 2^{5}\]
\[z_{255, 21} = z_{255, 5} \times z_{11}\]
Can still vectorize inside a single field op.
Strategy in our software:
50 mul insns starting from
\[(f_0, 2f_1), (f_2, 2f_3), (f_4, 2f_5), \ldots\]
\[(f_1, f_8), (f_3, f_0), (f_5, f_2), \ldots\]
\[(g_0, g_1), (g_2, g_3), (g_4, g_5), \ldots\]
\[(g_0, 19g_1), (g_2, 19g_3), (g_4, 19g_5), \ldots\]
\[(19g_2, 19g_3), (19g_4, 19g_5), \ldots\]
\[(19g_2, g_3), (19g_4, g_5), (19g_0, 19g_1)\].
Change carry pattern to vectorize,
e.g., \((h_0, h_4) \rightarrow (h_1, h_5)\).
Some field ops are hard to pair inside a single scalarmult.

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

Easy, constant time:
\[Z^{-1} = Z \cdot p^{-2} .\]

\[z_{10.0} = z_{10.5} \cdot z_{5.0} \]
\[z_{20.10} = z_{10.0} \cdot 2 \cdot 10 \]
\[z_{20.0} = z_{20.10} \cdot z_{10.0} \]
\[z_{40.20} = z_{20.0} \cdot 2 \cdot 20 \]
\[z_{40.0} = z_{40.20} \cdot z_{20.0} \]
\[z_{50.10} = z_{40.0} \cdot 2 \cdot 10 \]
\[z_{50.0} = z_{50.10} \cdot z_{10.0} \]
\[z_{100.50} = z_{50.0} \cdot z_{22.1} \]
\[z_{100.0} = z_{100.50} \cdot z_{5.0} \]
\[z_{200.10} = z_{100.0} \cdot 2 \cdot 10 \]
\[z_{200.0} = z_{200.10} \cdot z_{10.0} \]
\[z_{250.50} = z_{200.0} \cdot 2 \cdot 50 \]
\[z_{250.0} = z_{250.50} \cdot z_{5.0} \]
\[z_{255.5} = z_{250.0} \cdot 2 \cdot 5 \]
\[z_{255.21} = z_{255.5} \cdot z_{11} \]

Can still vectorize inside a single field op.

Strategy in our software:
50 mul insns starting from
\[(f_0, 2f_1), (f_2, 2f_3), (f_4, 2f_5), (f_6, 2f_7), (f_8, 2f_9),
(f_1, f_8), (f_3, f_0), (f_5, f_2), (f_7, f_4), (f_9, f_6);
(g_0, g_1), (g_2, g_3), (g_4, g_5), (g_6, g_7);
(g_0, 19g_1), (g_2, 19g_3), (g_4, 19g_5), (g_6, 19g_7);
(19g_2, 19g_3), (19g_4, 19g_5), (19g_6, 19g_7), (19g_8, 19g_9);
(19g_2, g_3), (19g_4, g_5), (19g_6, g_7), (19g_8, g_9).\]

Change carry pattern to vectorize, e.g., \((h_0, h_4) \rightarrow (h_1, h_5)\).
\begin{align*}
z_{10,0} &= z_{10,5} \times z_{5,0} \\
z_{20,10} &= z_{10,0}^{2^{10}} \\
z_{20,0} &= z_{20,10} \times z_{10,0} \\
z_{40,20} &= z_{20,0}^{2^{20}} \\
z_{40,0} &= z_{40,20} \times z_{20,0} \\
z_{50,10} &= z_{40,0}^{2^{10}} \\
z_{50,0} &= z_{50,10} \times z_{10,0} \\
z_{100,50} &= z_{50,0}^{2^{50}} \\
z_{100,0} &= z_{100,50} \times z_{50,0} \\
z_{200,100} &= z_{100,0}^{2^{100}} \\
z_{200,0} &= z_{200,100} \times z_{100,0} \\
z_{250,50} &= z_{200,0}^{2^{50}} \\
z_{250,0} &= z_{250,50} \times z_{50,0} \\
z_{255,5} &= z_{250,0}^{2^{5}} \\
z_{255,21} &= z_{255,5} \times z11
\end{align*}

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., \((h_0, h_4) \rightarrow (h_1, h_5)\).
= z_{10,5} \cdot z_{5,0}
= z_{10,0}^{2^10}
= z_{20,10} \cdot z_{10,0}
= z_{20,0}^{2^20}
= z_{40,20} \cdot z_{20,0}
= z_{40,0}^{2^20}
= z_{50,10} \cdot z_{10,0}
= z_{50,0}^{2^30}
= z_{100,50} \cdot z_{50,0}
= z_{100,0}^{2^50}
= z_{200,100} \cdot z_{100,0}
= z_{200,0}^{2^100}
= z_{250,50} \cdot z_{50,0}
= z_{250,0}^{2^50}
= z_{255,5} \cdot z_{11,0}
= z_{255,21} \cdot z_{11,1}

Can still vectorize inside a single field op.

Strategy in our software:

50 mul insns starting from

( f_0, 2 f_1, f_2, 2 f_3, f_4, 2 f_5, f_6, 2 f_7, f_8, 2 f_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, 19 g_1, g_2, 19 g_3, g_4, 19 g_5, g_6, 19 g_7, g_8, 19 g_9 ) 
( 19 g_2, 19 g_3, 19 g_4, 19 g_5, 19 g_6, 19 g_7, 19 g_8, 19 g_9 ) 
( 19 g_2, g_3, 19 g_4, g_5, 19 g_6, g_7, 19 g_8, g_9 )

Change carry pattern to vectorize, e.g., ( h_0, h_4 ) \rightarrow ( 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 \text{ bits}.
More detailed analysis:
356019 cycles on arithmetic;
\approx 78\% \text{ of software's total Cortex-A8-fast cycles for ECDH}.
Still some room for improvement.
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., \((h_0, h_4) \rightarrow (h_1, h_5)\).

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.
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., \((h_0, h_4) \rightarrow (h_1, h_5).\)

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.
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., \((h_0, h_4) \rightarrow (h_1, h_5)\).

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.
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., \((h_0, h_4) \rightarrow (h_1, h_5)\).

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.

Each CPU is a new adventure.

e.g. Could it be better to use Cortex-A7 FPU with radix \(2^{21.25}\)?
Can still vectorize inside a single field op.

Strategy in our software:

50 mul insns starting from

\((f_0, 2f_3), (f_4, 2f_5), (f_6, 2f_7), (f_8, 2f_9); (f_0, f_5), (f_2, f_7), (f_4, f_6); (g_0, g_3), (g_4, g_5), (g_6, g_7); (g_2, 19g_3), (g_4, 19g_5), (g_6, 19g_7), (g_8, 19g_9); (19g_4, 19g_5), (19g_6, 19g_7), (19g_8, 19g_9); (19g_4, g_5), (19g_6, g_7), (19g_8, g_9).\)

Change carry pattern to vectorize,

\((h_0, h_4) \rightarrow (h_1, h_5).\)

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.

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.

E.g. Could it be better to use Cortex-A7 FPU with radix \(2^{21.25}\)?
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_0; f_8); (f_2; f_0); (f_4; f_6); (f_6; f_7); (g_0; g_1);
(g_2; g_3); (g_4; g_5); (g_6; g_7); (g_8; g_9);
(19g_5); (g_6,19g_7); (g_8,19g_9);
(19g_6,19g_7); (19g_8,19g_9);
(g_6; g_7); (19g_8; g_9).

Pattern to vectorize,
( \( g_1, h_5 \).

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.

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

Some loss for carries etc.

ECDH: \(\approx10\) field muls \(\cdot\) 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
\(\approx78\%\) 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.
Core arithmetic: 100 cycles on mul insns for each field mul. Squarings are somewhat faster.

Some loss for carries etc.

ECDH: \( \approx 10 \text{ field muls} \cdot 255 \text{ bits} \).

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.

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