Smartphone/tablet CPUs

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

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

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

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

Apple A4 also appeared
in iPhone 4 (2010).

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

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

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

hone/tablet CPUs

2010) was the
ular tablet:
an 15 million sold.

ontains 45nm
4 system-on-chip.

4 contains
RM Cortex-A8 CPU core
rVR SGX 535 GPU.

A8 CPU core (2005)
s ARMv7-A insn set,
g NEON vector insns.

Apple A4 also appeared
in iPhone 4 (2010).

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

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

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

ARM de
supporti
Cortex-A
Cortex-A
Cortex-A
Cortex-A
Cortex-A

Also son

A9, A15
cores are
tries to
compens

t CPUs

s the
t:
ion sold.

5nm
on-chip.

s
x-A8 CPU core
535 GPU.

ore (2005)
A insn set,
ector insns.

Apple A4 also appeared
in iPhone 4 (2010).

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

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

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

ARM designed mo
supporting same A
Cortex-A9 (2007),
Cortex-A5 (2009),
Cortex-A15 (2010
Cortex-A7 (2011),
Cortex-A17 (2014

Also some larger 6

A9, A15, A17, and
cores are “out of o
tries to reorder ins
compensate for du

core

s.

1

Apple A4 also appeared
in iPhone 4 (2010).

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

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

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

2

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

Also some larger 64-bit core

A9, A15, A17, and some 64-
cores are “out of order”: CP
tries to reorder instructions t
compensate for dumb compi

Apple A4 also appeared
in iPhone 4 (2010).

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

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

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

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

Also some larger 64-bit cores.

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

4 also appeared
e 4 (2010).

GHz Samsung Exynos
Samsung Galaxy S (2010)
Cortex-A8 CPU core.

GHz TI OMAP3630 in
a Droid X (2010)
Cortex-A8 CPU core.

00MHz Freescale i.MX50
on Kindle 4 (2011)
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,
fewer ins

eared
).

ung Exynos
Galaxy S (2010)
8 CPU core.

MAP3630 in
(2010)
8 CPU core.

eescale i.MX50
4 (2011)
8 CPU core.

2

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.

3

A5, A7, original A
fewer insns at onc

os
(2010)
re.

in

re.

MX50

re.

2

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.

3

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

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.

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. $\Rightarrow$ Simpler, cheaper, more energy-efficient.

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.

esigned more cores

ng same ARMv7-A insns:
A9 (2007),
A5 (2009),
A15 (2010),
A7 (2011),
A17 (2014), etc.

ne larger 64-bit cores.

, A17, and some 64-bit
e "out of order": CPU
reorder instructions to
sate 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 c

Basic AR
16 32-bi

Optiona
16 128-b

Cortex-A
(and Co
and Qua
and Qua
always h

Cortex-A
sometim

ore cores
ARMv7-A insns:

),

), etc.

64-bit cores.

d some 64-bit
order": CPU
structions to
umb 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

Basic ARM insn se
16 32-bit registers

Optional NEON ex
16 128-bit register

Cortex-A7 and Co
(and Cortex-A15 a
and Qualcomm Sc
and Qualcomm Kr
always have NEON

Cortex-A5 and Co
sometimes have N

insns:

s.

-bit
PU
to
ilers.

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 u
16 128-bit registers: 2048 b

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

Cortex-A5 and Cortex-A9
sometimes have NEON insn

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.

original A8 are in-order,
sns at once. ⇒ Simpler,
more energy-efficient.

an one billion Cortex-A7
have been sold.

in low-cost and mid-range
ones: Mobiistar Buddy,
ar Kool, Mobiistar LAI Z1,
g Galaxy J1 Ace Neo, etc.

d in typical TV boxes,
martWatch 3, Samsung
, 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 Be
"NEON
new Cor
for vario

e.g. Curv
460200 c
498284 c

Compare
cycles or
for NIST
9 million
4.8 milli
3.9 milli

8 are in-order,
e. ⇒ Simpler,
rgy-efficient.

lion Cortex-A7
sold.

st and mid-range
biistar Buddy,
lobiistar LAI Z1,
J1 Ace Neo, etc.

al TV boxes,
3, Samsung
y 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–Sc
"NEON crypto" so
new Cortex-A8 spe
for various crypto

e.g. Curve25519 E
460200 cycles on
498284 cycles on

Compare to OpenS
cycles on Cortex-A
for NIST P-256 E
9 million for Open
4.8 million for Ope
3.9 million for Ope

rder,
pler,
nt.

x-A7

-range
ddy,
AI Z1,
o, etc.

es,
ng
c.

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 record
for various crypto primitives.

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

Compare to OpenSSL
cycles on Cortex-A8-slow
for NIST P-256 ECDH:
9 million for OpenSSL 0.9.8
4.8 million for OpenSSL 1.0
3.9 million for OpenSSL 1.0

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.

crypto

RM insn set uses
t registers: 512 bits.

l NEON extension uses
bit registers: 2048 bits.

A7 and Cortex-A8
rtex-A15 and Cortex-A17
lcomm Scorpion
lcomm Krait)
ave NEON insns.

A5 and Cortex-A9
es 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 i

`4x a = b`
is a vect
`a[0]`
`a[1]`
`a[2]`
`a[3]`

et uses
: 512 bits.

xtension uses
rs: 2048 bits.

rtex-A8
and Cortex-A17
corpion
rait)
N insns.

rtex-A9
EON 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

```
a[0] = b[0] +
a[1] = b[1] +
a[2] = b[2] +
a[3] = b[3] +
```

.

ses

its.

k-A17

s.

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 additi

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

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

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

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

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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

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

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

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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

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

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

ADD $\xrightarrow{2 \text{ cycles}}$ ADD $\xrightarrow{2 \text{ cycles}}$ ADD

rnstein–Schwabe

crypto" software:

tex-A8 speed records

us crypto primitives.

ve25519 ECDH:

cycles on Cortex-A8-fast,

cycles on Cortex-A8-slow.

e to OpenSSL

n Cortex-A8-slow

P-256 ECDH:

n for OpenSSL 0.9.8k.

on for OpenSSL 1.0.1c.

on for OpenSSL 1.0.2j.

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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 $\xrightarrow{2 \text{ cycles}}$ ADD $\xrightarrow{2 \text{ cycles}}$ ADD

`4x a = b`

is a vect

```
a[0]
a[1]
a[2]
a[3]
```

Stage N

Stage N

Stage N

ADD $\xrightarrow{2}$

Also log

chwabe

oftware:

eed records

primitives.

CDH:

Cortex-A8-fast,

Cortex-A8-slow.

SSL

A8-slow

CDH:

SSL 0.9.8k.

enSSL 1.0.1c.

enSSL 1.0.2j.

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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 $\xrightarrow{\text{2 cycles}}$ ADD $\xrightarrow{\text{2 cycles}}$ ADD

`4x a = b - c`

is a vector of 4 32

```
a[0] = b[0] -
a[1] = b[1] -
a[2] = b[2] -
a[3] = b[3] -
```

Stage N1: reads c

Stage N2: reads b

Stage N3: perform

Stage N4: a is rea

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

Also logic insns, sh

s

-fast,

-slow.

k.

.1c.

.2j.

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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 $\xrightarrow{\text{2 cycles}}$ ADD $\xrightarrow{\text{2 cycles}}$ ADD

`4x a = b - c`

is a vector of 4 32-bit subtra

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

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

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

Also logic insns, shifts, etc.

NEON instructions

`4x a = b + c`

is a vector of 4 32-bit additions:

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

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

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

ADD $\xrightarrow{\text{2 cycles}}$ ADD $\xrightarrow{\text{2 cycles}}$ ADD

`4x a = b - c`

is a vector of 4 32-bit subtractions:

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

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

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

Also logic insns, shifts, etc.

nstructions

```
o + c
```

tor of 4 32-bit additions:

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

A8 NEON arithmetic unit
this every cycle.

2: reads b and c.

3: performs addition.

4: a is ready.

2 cycles → ADD → 2 cycles → ADD

`4x a = b - c`

is a vector of 4 32-bit subtractions:

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

Stage N1: reads c.

Stage N2: reads b, negates c.

Stage N3: performs addition.

Stage N4: a is ready.

ADD → (2 or 3 cycles) → SUB

Also logic insns, shifts, etc.

Multiplic

c[0,1]

c[2,3]

Two cyc

Multiply

c[0,1]

c[2,3]

Also two

Stage N

Stage N

Stage N

:

Stage N

s

-bit additions:

c[0];
c[1];
c[2];
c[3].

arithmetic unit
cycle.

and c.
ns addition.
dy.

$$\text{DD} \xrightarrow{2\text{ cycles}} \text{ADD}$$

4x a = b - c

is a vector of 4 32-bit subtractions:

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

Stage N1: reads c.

Stage N2: reads b, negates c.

Stage N3: performs addition.

Stage N4: a is ready.

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

Also logic insns, shifts, etc.

Multiplication insn

c[0,1] = a[0] si
c[2,3] = a[1] si

Two cycles on Cor

Multiply-accumula

c[0,1] += a[0] s
c[2,3] += a[1] s

Also two cycles on

Stage N1: reads b
Stage N2: reads a
Stage N3: reads c
:
Stage N8: c is rea

ons:

c unit

n.

es
→ ADD

`4x a = b - c`

is a vector of 4 32-bit subtractions:

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

Stage N1: reads `c`.

Stage N2: reads `b`, negates `c`.

Stage N3: performs addition.

Stage N4: `a` is ready.

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

Also logic insns, shifts, etc.

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A

Stage N1: reads `b`.

Stage N2: reads `a`.

Stage N3: reads `c` if accumu

⋮

Stage N8: `c` is ready.

`4x a = b - c`

is a vector of 4 32-bit subtractions:

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

Stage N1: reads `c`.

Stage N2: reads `b`, negates `c`.

Stage N3: performs addition.

Stage N4: `a` is ready.

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

Also logic insns, shifts, etc.

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A8.

Stage N1: reads `b`.

Stage N2: reads `a`.

Stage N3: reads `c` if accumulate.

⋮

Stage N8: `c` is ready.

b - c

tor of 4 32-bit subtractions:

= b[0] - c[0];

= b[1] - c[1];

= b[2] - c[2];

= b[3] - c[3].

1: reads c.

2: reads b, negates c.

3: performs addition.

4: a is ready.

or 3 cycles → SUB

ic insns, shifts, etc.

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A8.

Stage N1: reads b.

Stage N2: reads a.

Stage N3: reads c if accumulate.

⋮

Stage N8: c is ready.

Typical s

```
c[0,1]
c[2,3]

c[0,1]
c[2,3]

c[0,1]
c[2,3]
```

Cortex-A

Reads c

-bit subtractions:

```
 c[0];
 c[1];
 c[2];
 c[3].
```

.

, negates c.

ns addition.

dy.

→ SUB

ifts, etc.

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A8.

Stage N1: reads b.

Stage N2: reads a.

Stage N3: reads c if accumulate.

⋮

Stage N8: c is ready.

Typical sequence

```
c[0,1] = a[0] si
c[2,3] = a[1] si

c[0,1] += e[2] s
c[2,3] += e[3] s

c[0,1] += g[0] s
c[2,3] += g[1] s
```

Cortex-A8 recogni

Reads c in N6 inst

actions:

c.

n.

8

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A8.

Stage N1: reads `b`.

Stage N2: reads `a`.

Stage N3: reads `c` if accumulate.

⋮

Stage N8: `c` is ready.

9

Typical sequence of three in

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

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

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

Cortex-A8 recognizes this pa

Reads `c` in N6 instead of N3

Multiplication insn:

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

Two cycles on Cortex-A8.

Multiply-accumulate insn:

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

Also two cycles on Cortex-A8.

Stage N1: reads `b`.

Stage N2: reads `a`.

Stage N3: reads `c` if accumulate.

⋮

Stage N8: `c` is ready.

Typical sequence of three insns:

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

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

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

Cortex-A8 recognizes this pattern.

Reads `c` in N6 instead of N3.

cation insn:

```
= a[0] signed* b[0];
= a[1] signed* b[1]
```

cles on Cortex-A8.

-accumulate insn:

```
+= a[0] signed* b[0];
+= a[1] signed* b[1]
```

o cycles on Cortex-A8.

1: reads b.

2: reads a.

3: reads c if accumulate.

8: c is ready.

Typical sequence of three insns:

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

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

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

Cortex-A8 recognizes this pattern.

Reads c in N6 instead of N3.

Time	N1
1	*b*
2	
3	*f*
4	
5	*h*
6	
7	
8	
9	
10	
11	
12	

n:

igned* b[0];

igned* b[1]

rtex-A8.

nte insn:

signed* b[0];

signed* b[1]

n Cortex-A8.

.

.

: if accumulate.

ady.

Typical sequence of three insns:

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

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

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

Cortex-A8 recognizes this pattern.

Reads c in N6 instead of N3.

Time	N1	N2	N3	N4
1	*b*			
2		*a*		
3	*f*		×	
4		*e*		×
5	*h*		×	
6		*g*		×
7			×	
8				×
9				
10				
11				
12				

```
[0];
[1]

[0];
[1]

8.

ulate.
```

Typical sequence of three insns:

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

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

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

Cortex-A8 recognizes this pattern.

Reads c in N6 instead of N3.

Time	N1	N2	N3	N4	N5	N6	N
1	*b*						
2		*a*					
3	*f*		×				
4		*e*		×			
5	*h*		×		×		
6		*g*		×		×	
7			×		×		
8				×		×	
9					×		+
10						×	
11							+
12							

Typical sequence of three insns:

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

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

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

Cortex-A8 recognizes this pattern.

Reads c in N6 instead of N3.

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

sequence of three insns:

```
= a[0] signed* b[0];
= a[1] signed* b[1]

+= e[2] signed* f[2];
+= e[3] signed* f[3]

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

A8 recognizes this pattern.
in N6 instead of N3.

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

NEON a
and perm
r = s[1]

Cortex-A
NEON l
that run
NEON a

Arithme
most im
can ofte
to hide l

Cortex-A
handling

of three insns:

```
igned* b[0];
igned* b[1]

signed* f[2];
signed* f[3]

signed* h[2];
signed* h[3]
```

zes this pattern.

tead of N3.

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

NEON also has loa
and permutation i
`r = s[1] t[2] r[`

Cortex-A8 has a se
NEON load/store
that runs in paralle
NEON arithmetic

Arithmetic is typic
most important bo
can often schedule
to hide loads/store

Cortex-A7 is differ
handling all NEON

sns:

```
[0];
[1]

[2];
[3]

[2];
[3]
```

attern.

3.

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

NEON also has load/store in
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 u
handling all NEON insns.

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

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

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

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

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

	N2	N3	N4	N5	N6	N7	N8
	a						
		$\times$					
	e		$\times$				
		$\times$		$\times$			
	g		$\times$		$\times$		
		$\times$		$\times$			
			$\times$		$\times$		c
				$\times$		$+$	
					$\times$		c
						$+$	
							c

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

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

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

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

Curve25

Radix 2^2

$(f_0, f_1, f_2$

to repres

$f = f_0 +$

$2^{102} f_4 +$

$2^{204} f_8 +$

Unscaled

f is valu

$f_0 t^0 + 2$

$f_4 t^4 + 2$

$f_8 t^8 + 2$

4	N5	N6	N7	N8
	×			
		×		
	×			
		×		c
	×		+	
		×		c
			+	
				c

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

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

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

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

Curve25519 on NE

Radix $2^{25.5}$: Use s
$(f_0, f_1, f_2, f_3, f_4, f_5,$
to represent the in
$f = f_0 + 2^{26} f_1 + 2$
$2^{102} f_4 + 2^{128} f_5 + 2$
$2^{204} f_8 + 2^{230} f_9$ mo

Unscaled polynom
f is value at $2^{25.5}$
$f_0 t^0 + 2^{0.5} f_1 t^1 + f$
$f_4 t^4 + 2^{0.5} f_5 t^5 + f$
$f_8 t^8 + 2^{0.5} f_9 t^9$.

7 | N8

c

c

c

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

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

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

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

Curve25519 on NEON

Radix $2^{25.5}$: Use small integ
$(f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9$
to represent the integer
$f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^7$
$2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^1$
$2^{204} f_8 + 2^{230} f_9$ modulo 2^{255}

Unscaled polynomial view:
f is value at $2^{25.5}$ of the pol
$f_0 t^0 + 2^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5}$
$f_4 t^4 + 2^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5}$
$f_8 t^8 + 2^{0.5} f_9 t^9$.

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

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

lso has load/store insns
mutation insns: e.g.,
] t[2] r[2,3]

A8 has a separate
oad/store unit
s in parallel with
rithmetic unit.

tic is typically
portant bottleneck:
n schedule insns
oads/stores/perms.

A7 is different: one unit
g all NEON insns.

Curve25519 on NEON

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

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

$h \equiv f g$

$h_0 = f_0 g_0$
$h_1 = f_0 g_1$
$h_2 = f_0 g_2$
$h_3 = f_0 g_3$
$h_4 = f_0 g_4$
$h_5 = f_0 g_5$
$h_6 = f_0 g_6$
$h_7 = f_0 g_7$
$h_8 = f_0 g_8$
$h_9 = f_0 g_9$

Proof: r

ad/store insns

nsns: e.g.,

2,3]

eparate

unit

el with

unit.

cally

ottleneck:

e insns

es/perms.

rent: one unit

N insns.

Curve25519 on NEON

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

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

$h \equiv f g \pmod{2^{25}}$

$h_0 = f_0 g_0 + 38 f_1 g_9 + 19$

$h_1 = f_0 g_1 + f_1 g_0 + 19$

$h_2 = f_0 g_2 + 2 f_1 g_1 +$

$h_3 = f_0 g_3 + f_1 g_2 +$

$h_4 = f_0 g_4 + 2 f_1 g_3 +$

$h_5 = f_0 g_5 + f_1 g_4 +$

$h_6 = f_0 g_6 + 2 f_1 g_5 +$

$h_7 = f_0 g_7 + f_1 g_6 +$

$h_8 = f_0 g_8 + 2 f_1 g_7 +$

$h_9 = f_0 g_9 + f_1 g_8 +$

Proof: multiply po

nsns

unit

Curve25519 on NEON

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

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

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

$h_0 = f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g$
$h_1 = f_0 g_1 + f_1 g_0 + 19 f_2 g_9 + 19 f_3 g$
$h_2 = f_0 g_2 + 2 f_1 g_1 + f_2 g_0 + 38 f_3 g$
$h_3 = f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g$
$h_4 = f_0 g_4 + 2 f_1 g_3 + f_2 g_2 + 2 f_3 g$
$h_5 = f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g$
$h_6 = f_0 g_6 + 2 f_1 g_5 + f_2 g_4 + 2 f_3 g$
$h_7 = f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g$
$h_8 = f_0 g_8 + 2 f_1 g_7 + f_2 g_6 + 2 f_3 g$
$h_9 = f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g$

Proof: multiply polys mod t

Curve25519 on NEON

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

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

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

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

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

519 on NEON

$^{25.5}$: Use small integers

$_2, f_3, f_4, f_5, f_6, f_7, f_8, f_9)$

sent the integer

$- 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 +$

$2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 +$

$2^{230} f_9$ modulo $2^{255} - 19$.

d polynomial view:

e at $2^{25.5}$ of the poly

$^{0.5} f_1 t^1 + f_2 t^2 + 2^{0.5} f_3 t^3 +$

$^{0.5} f_5 t^5 + f_6 t^6 + 2^{0.5} f_7 t^7 +$

$^{0.5} f_9 t^9$.

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

$$
\begin{aligned}
h_0 &= f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g_7 + 19 f_4 g_6 + 38 f_5 g_5 + 19 f \\
h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_9 + 19 f_3 g_8 + 19 f_4 g_7 + 19 f_5 g_6 + 19 f \\
h_2 &= f_0 g_2 + 2 f_1 g_1 + f_2 g_0 + 38 f_3 g_9 + 19 f_4 g_8 + 38 f_5 g_7 + 19 f \\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_9 + 19 f_5 g_8 + 19 f \\
h_4 &= f_0 g_4 + 2 f_1 g_3 + f_2 g_2 + 2 f_3 g_1 + f_4 g_0 + 38 f_5 g_9 + 19 f \\
h_5 &= f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + f_5 g_0 + 19 f \\
h_6 &= f_0 g_6 + 2 f_1 g_5 + f_2 g_4 + 2 f_3 g_3 + f_4 g_2 + 2 f_5 g_1 + f \\
h_7 &= f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + f_5 g_2 + f \\
h_8 &= f_0 g_8 + 2 f_1 g_7 + f_2 g_6 + 2 f_3 g_5 + f_4 g_4 + 2 f_5 g_3 + f \\
h_9 &= f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + f_5 g_4 + f
\end{aligned}
$$

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

EON

small integers

$f_6, f_7, f_8, f_9)$

nteger

$2^{51} f_2 + 2^{77} f_3 +$

$2^{153} f_6 + 2^{179} f_7 +$

odulo $2^{255} - 19$.

ial view:

of the poly

$f_2 t^2 + 2^{0.5} f_3 t^3 +$

$f_6 t^6 + 2^{0.5} f_7 t^7 +$

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

$$
\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_9 + 19f_2g_8 + 38f_3g_7 + 19f_4g_6 + 38f_5g_5 + 19f_6g_4 + 38f_7g_3 \\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_9 + 19f_3g_8 + 19f_4g_7 + 19f_5g_6 + 19f_6g_5 + 19f_7g_4 \\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 \\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 \\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 \\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + 19f_6g_9 + 19f_7g_8 \\
h_6 &= f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + 2f_5g_1 + f_6g_0 + 38f_7g_9 \\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 \\
h_8 &= f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + 2f_5g_3 + f_6g_2 + 2f_7g_1 \\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2
\end{aligned}
$$

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

gers

$g_9)$

$^7 f_3 +$

$^{79} f_7 +$

$- 19.$

ly

$f_3 t^3 +$

$f_7 t^7 +$

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

$$
\begin{aligned}
h_0 &= f_0 g_0 + 38 f_1 g_9 + 19 f_2 g_8 + 38 f_3 g_7 + 19 f_4 g_6 + 38 f_5 g_5 + 19 f_6 g_4 + 38 f_7 g_3 + 19 f_8 g_2 + 3 \\
h_1 &= f_0 g_1 + f_1 g_0 + 19 f_2 g_9 + 19 f_3 g_8 + 19 f_4 g_7 + 19 f_5 g_6 + 19 f_6 g_5 + 19 f_7 g_4 + 19 f_8 g_3 + 1 \\
h_2 &= f_0 g_2 + 2 f_1 g_1 + f_2 g_0 + 38 f_3 g_9 + 19 f_4 g_8 + 38 f_5 g_7 + 19 f_6 g_6 + 38 f_7 g_5 + 19 f_8 g_4 + 3 \\
h_3 &= f_0 g_3 + f_1 g_2 + f_2 g_1 + f_3 g_0 + 19 f_4 g_9 + 19 f_5 g_8 + 19 f_6 g_7 + 19 f_7 g_6 + 19 f_8 g_5 + 1 \\
h_4 &= f_0 g_4 + 2 f_1 g_3 + f_2 g_2 + 2 f_3 g_1 + f_4 g_0 + 38 f_5 g_9 + 19 f_6 g_8 + 38 f_7 g_7 + 19 f_8 g_6 + 3 \\
h_5 &= f_0 g_5 + f_1 g_4 + f_2 g_3 + f_3 g_2 + f_4 g_1 + f_5 g_0 + 19 f_6 g_9 + 19 f_7 g_8 + 19 f_8 g_7 + 1 \\
h_6 &= f_0 g_6 + 2 f_1 g_5 + f_2 g_4 + 2 f_3 g_3 + f_4 g_2 + 2 f_5 g_1 + f_6 g_0 + 38 f_7 g_9 + 19 f_8 g_8 + 3 \\
h_7 &= f_0 g_7 + f_1 g_6 + f_2 g_5 + f_3 g_4 + f_4 g_3 + f_5 g_2 + f_6 g_1 + f_7 g_0 + 19 f_8 g_9 + 1 \\
h_8 &= f_0 g_8 + 2 f_1 g_7 + f_2 g_6 + 2 f_3 g_5 + f_4 g_4 + 2 f_5 g_3 + f_6 g_2 + 2 f_7 g_1 + f_8 g_0 + 3 \\
h_9 &= f_0 g_9 + f_1 g_8 + f_2 g_7 + f_3 g_6 + f_4 g_5 + f_5 g_4 + f_6 g_3 + f_7 g_2 + f_8 g_1 +
\end{aligned}
$$

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

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

$$
\begin{aligned}
h_0 &= f_0g_0+38f_1g_9+19f_2g_8+38f_3g_7+19f_4g_6+38f_5g_5+19f_6g_4+38f_7g_3+19f_8g_2+38f_9g_1,\\
h_1 &= f_0g_1+\ \ f_1g_0+19f_2g_9+19f_3g_8+19f_4g_7+19f_5g_6+19f_6g_5+19f_7g_4+19f_8g_3+19f_9g_2,\\
h_2 &= f_0g_2+\ 2f_1g_1+\ \ f_2g_0+38f_3g_9+19f_4g_8+38f_5g_7+19f_6g_6+38f_7g_5+19f_8g_4+38f_9g_3,\\
h_3 &= f_0g_3+\ \ f_1g_2+\ \ f_2g_1+\ \ f_3g_0+19f_4g_9+19f_5g_8+19f_6g_7+19f_7g_6+19f_8g_5+19f_9g_4,\\
h_4 &= f_0g_4+\ 2f_1g_3+\ \ f_2g_2+\ 2f_3g_1+\ \ f_4g_0+38f_5g_9+19f_6g_8+38f_7g_7+19f_8g_6+38f_9g_5,\\
h_5 &= f_0g_5+\ \ f_1g_4+\ \ f_2g_3+\ \ f_3g_2+\ \ f_4g_1+\ \ f_5g_0+19f_6g_9+19f_7g_8+19f_8g_7+19f_9g_6,\\
h_6 &= f_0g_6+\ 2f_1g_5+\ \ f_2g_4+\ 2f_3g_3+\ \ f_4g_2+\ 2f_5g_1+\ \ f_6g_0+38f_7g_9+19f_8g_8+38f_9g_7,\\
h_7 &= f_0g_7+\ \ f_1g_6+\ \ f_2g_5+\ \ f_3g_4+\ \ f_4g_3+\ \ f_5g_2+\ \ f_6g_1+\ \ f_7g_0+19f_8g_9+19f_9g_8,\\
h_8 &= f_0g_8+\ 2f_1g_7+\ \ f_2g_6+\ 2f_3g_5+\ \ f_4g_4+\ 2f_5g_3+\ \ f_6g_2+\ 2f_7g_1+\ \ f_8g_0+38f_9g_9,\\
h_9 &= f_0g_9+\ \ f_1g_8+\ \ f_2g_7+\ \ f_3g_6+\ \ f_4g_5+\ \ f_5g_4+\ \ f_6g_3+\ \ f_7g_2+\ \ f_8g_1+\ \ f_9g_0.
\end{aligned}
$$

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

(mod $2^{255} - 19$) where

$$
\begin{aligned}
&+38f_1g_9+19f_2g_8+38f_3g_7+19f_4g_6+38f_5g_5+19f_6g_4+38f_7g_3+19f_8g_2+38f_9g_1,\\
&+\ f_1g_0+19f_2g_9+19f_3g_8+19f_4g_7+19f_5g_6+19f_6g_5+19f_7g_4+19f_8g_3+19f_9g_2,\\
&+\ 2f_1g_1+\ f_2g_0+38f_3g_9+19f_4g_8+38f_5g_7+19f_6g_6+38f_7g_5+19f_8g_4+38f_9g_3,\\
&+\ f_1g_2+\ f_2g_1+\ f_3g_0+19f_4g_9+19f_5g_8+19f_6g_7+19f_7g_6+19f_8g_5+19f_9g_4,\\
&+\ 2f_1g_3+\ f_2g_2+\ 2f_3g_1+\ f_4g_0+38f_5g_9+19f_6g_8+38f_7g_7+19f_8g_6+38f_9g_5,\\
&+\ f_1g_4+\ f_2g_3+\ f_3g_2+\ f_4g_1+\ f_5g_0+19f_6g_9+19f_7g_8+19f_8g_7+19f_9g_6,\\
&+\ 2f_1g_5+\ f_2g_4+\ 2f_3g_3+\ f_4g_2+\ 2f_5g_1+\ f_6g_0+38f_7g_9+19f_8g_8+38f_9g_7,\\
&+\ f_1g_6+\ f_2g_5+\ f_3g_4+\ f_4g_3+\ f_5g_2+\ f_6g_1+\ f_7g_0+19f_8g_9+19f_9g_8,\\
&+\ 2f_1g_7+\ f_2g_6+\ 2f_3g_5+\ f_4g_4+\ 2f_5g_3+\ f_6g_2+\ 2f_7g_1+\ f_8g_0+38f_9g_9,\\
&+\ f_1g_8+\ f_2g_7+\ f_3g_6+\ f_4g_5+\ f_5g_4+\ f_6g_3+\ f_7g_2+\ f_8g_1+\ f_9g_0.
\end{aligned}
$$

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

Each h_i
products
of $2f_1, 2$
$19g_1, 19$

Each h_i
under re
sizes of

(Analyze
bugs can
See 201
Barbosa
several r

$h_0, h_1, \ldots$
for subse

$^{55} - 19)$ where

$$
\begin{aligned}
&9f_2g_8 + 38f_3g_7 + 19f_4g_6 + 38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,\\
&9f_2g_9 + 19f_3g_8 + 19f_4g_7 + 19f_5g_6 + 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,\\
&\ f_2g_0 + 38f_3g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,\\
&\ f_2g_1 + \ \ f_3g_0 + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,\\
&\ f_2g_2 + \ 2f_3g_1 + \ \ f_4g_0 + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,\\
&\ f_2g_3 + \ \ f_3g_2 + \ \ f_4g_1 + \ \ f_5g_0 + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,\\
&\ f_2g_4 + \ 2f_3g_3 + \ \ f_4g_2 + \ 2f_5g_1 + \ \ f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,\\
&\ f_2g_5 + \ \ f_3g_4 + \ \ f_4g_3 + \ \ f_5g_2 + \ \ f_6g_1 + \ \ f_7g_0 + 19f_8g_9 + 19f_9g_8,\\
&\ f_2g_6 + \ 2f_3g_5 + \ \ f_4g_4 + \ 2f_5g_3 + \ \ f_6g_2 + \ 2f_7g_1 + \ \ f_8g_0 + 38f_9g_9,\\
&\ f_2g_7 + \ \ f_3g_6 + \ \ f_4g_5 + \ \ f_5g_4 + \ \ f_6g_3 + \ \ f_7g_2 + \ \ f_8g_1 + \ \ f_9g_0.
\end{aligned}
$$

olys mod $t^{10} - 19$.

Each h_i is a sum o
products after prec
of $2f_1, 2f_3, 2f_5, 2f_7$
$19g_1, 19g_2, \ldots, 19$

Each h_i fits into 6
under reasonable l
sizes of $f_1, g_1, \ldots,$

(Analyze this very
bugs can slip past
See 2011 Brumley
Barbosa–Vercaute
several recent Ope

$h_0, h_1, \ldots$ are too
for subsequent mu

where

$$
\begin{aligned}
&g_7 + 19f_4g_6 + 38f_5g_5 + 19f_6g_4 + 38f_7g_3 + 19f_8g_2 + 38f_9g_1,\\
&g_8 + 19f_4g_7 + 19f_5g_6 + 19f_6g_5 + 19f_7g_4 + 19f_8g_3 + 19f_9g_2,\\
&g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,\\
&g_0 + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,\\
&g_1 + f_4g_0 + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,\\
&g_2 + f_4g_1 + f_5g_0 + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,\\
&g_3 + f_4g_2 + 2f_5g_1 + f_6g_0 + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,\\
&g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_9 + 19f_9g_8,\\
&g_5 + f_4g_4 + 2f_5g_3 + f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_9,\\
&g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\end{aligned}
$$

$t^{10} - 19$.

Each h_i is a sum of ten products after precomputati
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 test See 2011 Brumley–Page–Barbosa–Vercauteren and several recent OpenSSL bug

$h_0, h_1, \ldots$ are too large for subsequent multiplicatio

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

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

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

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

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

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

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

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

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

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

Carry h_0
replace (
(h_0 mod
This ma

Similarly
Eventua

We actu
Slightly
(given d
but mor

Some th

- Mix si
- Interle

$+19f_8g_2+38f_9g_1,$

$+19f_8g_3+19f_9g_2,$

$+19f_8g_4+38f_9g_3,$

$+19f_8g_5+19f_9g_4,$

$+19f_8g_6+38f_9g_5,$

$+19f_8g_7+19f_9g_6,$

$+19f_8g_8+38f_9g_7,$

$+19f_8g_9+19f_9g_8,$

$+\quad f_8g_0+38f_9g_9,$

$+\quad f_8g_1+\quad f_9g_0.$

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

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

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

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

Carry $h_0 \to h_1$: i.e replace (h_0, h_1) wi $(h_0 \bmod 2^{26}, h_1 +$ This makes h_0 sm

Similarly for other Eventually all h_i a

We actually use si Slightly more expe (given details of in but more room for

Some things we ha

- Mix signed, unsi
- Interleave reduct

$8f_9g_1$,
$9f_9g_2$,
$8f_9g_3$,
$9f_9g_4$,
$8f_9g_5$,
$9f_9g_6$,
$8f_9g_7$,
$9f_9g_8$,
$8f_9g_9$,
f_9g_0.

15

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.

16

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

Similarly for other h_i.
Eventually all h_i are small e

We actually use signed coeff
Slightly more expensive carr
(given details of insn set)
but more room for $ab + c^2$

Some things we haven't trie
- Mix signed, unsigned carri
- Interleave reduction, carry

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

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

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

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

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

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

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

Some things we haven't tried yet:

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

is a sum of ten

s after precomputation

$f_3, 2f_5, 2f_7, 2f_9,$

$g_2, \ldots, 19g_9$.

fits into 64 bits

asonable limits on

$f_1, g_1, \ldots, f_9, g_9$.

e this very carefully:

n slip past most tests!

1 Brumley–Page–

–Vercauteren and

ecent OpenSSL bugs.)

. . are too large

equent multiplication.

Carry $h_0 \to h_1$: i.e.,

replace (h_0, h_1) with

$(h_0 \bmod 2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$.

This makes h_0 small.

Similarly for other h_i.

Eventually all h_i are small enough.

We actually use signed coeffs.

Slightly more expensive carries

(given details of insn set)

but more room for $ab + c^2$ etc.

Some things we haven't tried yet:

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

Minor ch

Result o

used unt

Find an

for the C

while th

Sometim

higher-le

Example

$h_2 \to h_3$

$h_7 \to h_8$

have lon

of ten
computation
$, 2f_9,$
g_9.

4 bits
imits on
f_9, g_9.

carefully:
most tests!
–Page–
ren and
enSSL bugs.)

large
ltiplication.

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

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

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

Some things we haven't tried yet:

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

Minor challenge: p
Result of each insn
used until a few cy

Find an independe
for the CPU to sta
while the first insn

Sometimes helps t
higher-level compu

Example: carries h
$h_2 \to h_3 \to h_4 \to$
$h_7 \to h_8 \to h_9 \to$
have long chain of

on

s!

s.)

n.

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

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

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

Some things we haven't tried yet:

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

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

Find an independent insn
for the CPU to start workin
while the first insn is in prog

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 dependen

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

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

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

Some things we haven't tried yet:

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

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

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

Sometimes helps to adjust
higher-level computations.

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

$\to h_1$: i.e.,
(h_0, h_1) with
$2^{26}, h_1 + \lfloor h_0/2^{26} \rfloor)$.
kes h_0 small.

for other h_i.
lly all h_i are small enough.

ally use signed coeffs.
more expensive carries
etails of insn set)
e room for $ab + c^2$ etc.

ings we haven't tried yet:
gned, unsigned carries.
ave 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.

Alternat
$h_0 \to h_1$
$h_1 \to h_2$
$h_2 \to h_3$
$h_3 \to h_4$
$h_4 \to h_5$
$h_5 \to h_6$

12 carrie
but later

Now mu
to find i
for CPU

e.,
th
$\lfloor h_0/2^{26} \rfloor$).
all.

h_i.
re small enough.

gned coeffs.
ensive carries
nsn set)
r $ab + c^2$ etc.

aven't tried yet:
gned carries.
tion, carrying.

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

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

Sometimes helps to adjust
higher-level computations.

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

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

12 carries instead
but latency is muc

Now much easier
to find independen
for CPU to handle

).

nough.

fs.

ies

etc.

d yet:

es.

ing.

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 paralle

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.

hallenge: pipelining.
f each insn cannot be
til a few cycles later.

independent insn
CPU to start working on
e first insn is in progress.

nes helps to adjust
evel computations.

e: carries $h_0 \to h_1 \to$
$\to h_4 \to h_5 \to h_6 \to$
$\to h_9 \to h_0 \to h_1$
g chain of dependencies.

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

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

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

Major ch

e.g. `4x a`
does 4 a
but need
of inputs

On Cort
occasion
run in pa
but frequ
would be

On Cort
every op

pipelining.
n cannot be
ycles later.

ent insn
art working on
n is in progress.

o adjust
utations.

$h_0 \to h_1 \to$
$h_5 \to h_6 \to$
$h_0 \to h_1$
f dependencies.

Alternative: carry

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

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

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

Major challenge: v

e.g. `4x a = b + c`
does 4 additions a
but needs particula
of inputs and outp

On Cortex-A8,
occasional permut
run in parallel with
but frequent perm
would be a bottlen

On Cortex-A7,
every operation co

pe
.

g on
gress.

→
→

ncies.

Alternative: carry

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

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

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

Major challenge: vectorizati

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

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

On Cortex-A7,
every operation costs cycles.

Alternative: carry

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

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

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

Major challenge: vectorization.

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

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

On Cortex-A7,
every operation costs cycles.

ive: carry

and $h_5 \to h_6$;

and $h_6 \to h_7$;

and $h_7 \to h_8$;

and $h_8 \to h_9$;

and $h_9 \to h_0$;

and $h_0 \to h_1$.

es instead of 11,

ncy is much smaller.

ch easier

ndependent insns

to handle in parallel.

Major challenge: vectorization.

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

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

On Cortex-A7,
every operation costs cycles.

Often hi

do a pai

$h = f\,g$;

Vectoriz

Merge f_0

and f_0', f

into vect

Similarly

Then co

Computa

into NE

`c[0,1]`

`c[2,3]`

$\to h_6$;
$\to h_7$;
$\to h_8$;
$\to h_9$;
$\to h_0$;
$\to h_1$.

of 11,
ch smaller.

nt insns
e in parallel.

Major challenge: vectorization.

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

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

On Cortex-A7,
every operation costs cycles.

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

Vectorize across th
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 $(h_i$

Computation fits n
into NEON insns:
`c[0,1] = a[0] si`
`c[2,3] = a[1] si`

Major challenge: vectorization.

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

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

On Cortex-A7,
every operation costs cycles.

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

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

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

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

Major challenge: vectorization.

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

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

On Cortex-A7,
every operation costs cycles.

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

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

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

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

hallenge: vectorization.

`a = b + c`

dditions at once,

ds particular arrangement

s and outputs.

ex-A8,

nal permutations

arallel with arithmetic,

uent permutations

e a bottleneck.

ex-A7,

peration costs cycles.

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

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

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

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

Example
$C = X_1$
inside po
for Edwa

vectorization.

t once,

ar arrangement

outs.

tations

n arithmetic,

utations

heck.

osts cycles.

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

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

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

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

Example: Recall
$C = X_1 \cdot X_2$; $D =$
inside point-additi
for Edwards curves

on.

ment

ic,

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

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

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

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


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

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

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

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

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


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

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

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

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

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


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

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

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

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

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

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


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

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

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

gher-level operations
r of mults in parallel:
$h' = f'g'$.

e across those mults.
$_0, f_1, \ldots, f_9$
$f'_1, \ldots, f'_9$
tors (f_i, f'_i).
(g_i, g'_i).
mpute (h_i, h'_i).

ation fits naturally
ON insns: e.g.,

```
= a[0] signed* b[0];
= a[1] signed* b[1]
```


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

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

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

Example
with a b
can vect

operations
in parallel:

hose mults.
$_9$

).

$, h_i')$.

naturally
e.g.,

```
igned* b[0];
igned* b[1]
```


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

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

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

Example: A busy
with a backlog of
can vectorize acro

s
l:
s.

```
0];
1]
```

21

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

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

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

22

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

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

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

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

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

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

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

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

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

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.

e: Recall
$\cdot X_2$; $D = Y_1 \cdot Y_2$
oint-addition formulas
ards curves.

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

e: Typical algorithms
-base scalarmult
ny 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 fie
inside a

Example
convert
$Z^{-1}X \in$

Easy, co
$11\mathbf{M} + 2$

```
z2 = z1
z8 = z2
z9 = z1
z11 = z2
z22 = z
z_5_0 =
z_10_5 =
```

$Y_1 \cdot Y_2$

on formulas

s.

mpute

$7P$ as

$4P = 2P + 2P$;

$6P = 3P + 3P$

P.

algorithms

armult

l 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
inside a single scal

Example: At end
convert fraction (X
$Z^{-1}X \in \{0, 1, \ldots$

Easy, constant tim
$11\mathbf{M} + 254\mathbf{S}$ for p

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2
```

as

$+ 2P$;

$+ 3P$

s

ds.

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 p
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: $Z^{-1} =$
$11\mathbf{M} + 254\mathbf{S}$ for $p = 2^{255} -$

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```

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

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

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

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

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

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

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```

e: A busy server
acklog of scalarmults
orize across them.

a disadvantage of
ng across two mults:
f, f', g, g', h, h'
at least 1536 bits,
very little room
porary registers.

some loads and stores
ectorized `mulmul`.
nvisible on Cortex-A8,
ger issue on Cortex-A7.

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

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

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

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```

```
z_10_0 =
z_20_10
z_20_0 =
z_40_20
z_40_0 =
z_50_10
z_50_0 =
z_100_5
z_100_0
z_200_1
z_200_0
z_250_5
z_250_0
z_255_5
z_255_2
```

server
scalarmults
ss them.

ntage of
two mults:
h, h'
36 bits,
room
sters.

s and stores
ulmul.
n Cortex-A8,
n Cortex-A7.

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

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

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

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```

```
z_10_0 = z_10_5*
z_20_10 = z_10_0
z_20_0 = z_20_10
z_40_20 = z_20_0
z_40_0 = z_40_20
z_50_10 = z_40_0
z_50_0 = z_50_10
z_100_50 = z_50_
z_100_0 = z_100_
z_200_100 = z_10
z_200_0 = z_200_
z_250_50 = z_200
z_250_0 = z_250_
z_255_5 = z_250_
z_255_21 = z_255
```

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

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

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

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```



```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```

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

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

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

```
z2 = z1^2^1
z8 = z2^2^2
z9 = z1*z8
z11 = z2*z9
z22 = z11^2^1
z_5_0 = z9*z22
z_10_5 = z_5_0^2^5
```



```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```

eld ops are hard to pair
single scalarmult.

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

nstant time: $Z^{-1} = Z^{p-2}$.
254**S** for $p = 2^{255} - 19$:

```
^2^1
^2^2
*z8
2*z9
11^2^1
 z9*z22
= z_5_0^2^5
```



```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```

Can still
inside a

Strategy

50 mul i

$(f_0, 2f_1), (f_2$

$(f_1, f_8), (f_3, f$

$(g_0, g_1), (g_2$

$(g_0, 19g_1), ($

$(19g_2, 19g_3$

$(19g_2, g_3), ($

Change

e.g., $(h_0$

e hard to pair
larmult.

of ECDH,
$X : Z$) into
$, p-1\}$.

ne: $Z^{-1} = Z^{p-2}$.
$= 2^{255} - 19$:

^5


```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```

Can still vectorize
inside a single fiel

Strategy in our so

50 mul insns starti

$(f_0,2f_1),(f_2,2f_3),(f_4,2f_5),$

$(f_1,f_8),(f_3,f_0),(f_5,f_2),(f_7,$

$(g_0,g_1),(g_2,g_3),(g_4,g_5),($

$(g_0,19g_1),(g_2,19g_3),(g_4,$

$(19g_2,19g_3),(19g_4,19g_5),$

$(19g_2,g_3),(19g_4,g_5),(19g$

Change carry patt

e.g., $(h_0, h_4) \rightarrow (h$

pair

o

$= Z^{p-2}$.

19:

```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```


Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

$(g_0,19g_1),(g_2,19g_3),(g_4,19g_5),(g_6,19g$

$(19g_2,19g_3),(19g_4,19g_5),(19g_6,19g_7),($

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

Change carry pattern to vec

e.g., $(h_0, h_4) \to (h_1, h_5)$.

```
z_10_0 = z_10_5*z_5_0
z_20_10 = z_10_0^2^10
z_20_0 = z_20_10*z_10_0
z_40_20 = z_20_0^2^20
z_40_0 = z_40_20*z_20_0
z_50_10 = z_40_0^2^10
z_50_0 = z_50_10*z_10_0
z_100_50 = z_50_0^2^50
z_100_0 = z_100_50*z_50_0
z_200_100 = z_100_0^2^100
z_200_0 = z_200_100*z_100_0
z_250_50 = z_200_0^2^50
z_250_0 = z_250_50*z_50_0
z_255_5 = z_250_0^2^5
z_255_21 = z_255_5*z11
```


Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

```
= z_10_5*z_5_0
 = z_10_0^2^10
= z_20_10*z_10_0
 = z_20_0^2^20
= z_40_20*z_20_0
 = z_40_0^2^10
= z_50_10*z_10_0
0 = z_50_0^2^50
 = z_100_50*z_50_0
00 = z_100_0^2^100
 = z_200_100*z_100_0
0 = z_200_0^2^50
 = z_250_50*z_50_0
 = z_250_0^2^5
1 = z_255_5*z11
```


Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

Core arit
on mul i
Squaring

Some lo

ECDH: 2

More de
356019 c

≈78% o
Cortex-A

Still som

```
z_5_0
^2^10
*z_10_0
^2^20
*z_20_0
^2^10
*z_10_0
0^2^50
50*z_50_0
0_0^2^100
100*z_100_0
_0^2^50
50*z_50_0
0^2^5
_5*z11
```


Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

Core arithmetic: 1
on mul insns for e
Squarings are som

Some loss for carr

ECDH: ≈10 field

More detailed ana
356019 cycles on
≈78% of software
Cortex-A8-fast cyc
Still some room fo

Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

Core arithmetic: 100 cycles
on mul insns for each field m
Squarings are somewhat fast

Some loss for carries etc.

ECDH: ≈10 field muls · 255

More detailed analysis:
356019 cycles on arithmetic
≈78% of software's total
Cortex-A8-fast cycles for EC
Still some room for improve

Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

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

Some loss for carries etc.

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

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

Can still vectorize
inside a single field op.

Strategy in our software:

50 mul insns starting from

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

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

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

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

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

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

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

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

Some loss for carries etc.

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

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

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

vectorize
single field op.

in our software:

nsns starting from

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

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

$,g_3),(g_4,g_5),(g_6,g_7)$;

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

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

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

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

carry pattern to vectorize,

$, h_4) \to (h_1, h_5)$.

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

Some loss for carries etc.

ECDH: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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 m

https:/

benchma
2137 pu
hundred
39 DH p
56 signa
304 auth

d op.

ftware:

ing from

$(f_6,2f_7),(f_8,2f_9)$;

$f_4),(f_9,f_6)$;

$g_6,g_7)$;

$19g_5),(g_6,19g_7),(g_8,19g_9)$;

$(19g_6,19g_7),(19g_8,19g_9)$;

$_6,g_7),(19g_8,g_9)$.

ern to vectorize,

$h_1, h_5)$.

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

Some loss for carries etc.

ECDH: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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 t

https://bench.c
benchmarks for (c
2137 public imple
hundreds of crypto
39 DH primitives,
56 signature primi
304 authenticated

f_9);

g_7),(g_8,19g_9);

19g_8,19g_9);

$_9$).

torize,

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

Some loss for carries etc.

ECDH: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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
hundreds of crypto primitive
39 DH primitives,
56 signature primitives,
304 authenticated ciphers, e

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

Some loss for carries etc.

ECDH: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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: ≈10 field muls · 255 bits.

More detailed analysis:
356019 cycles on arithmetic;
≈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.

Exercise: Make them faster!