Reverse Engineering the M1

Stan Skowronek

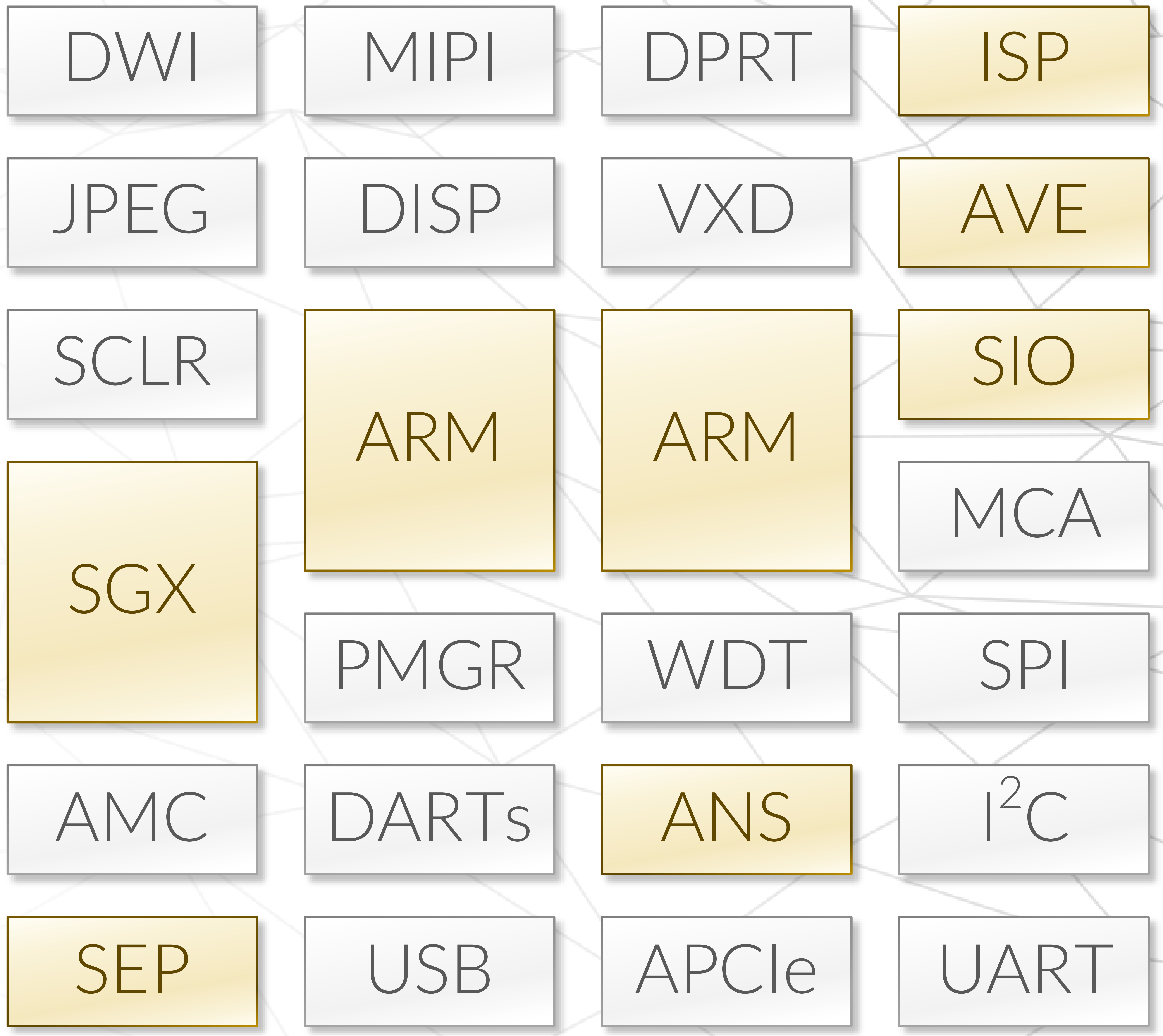
CORELLIUM

About me

- Electronic Engineer
- GPU logic designer
- Cryptographic / security HW
- Reverse Engineering for fun and profit
- Co-founded Corellium

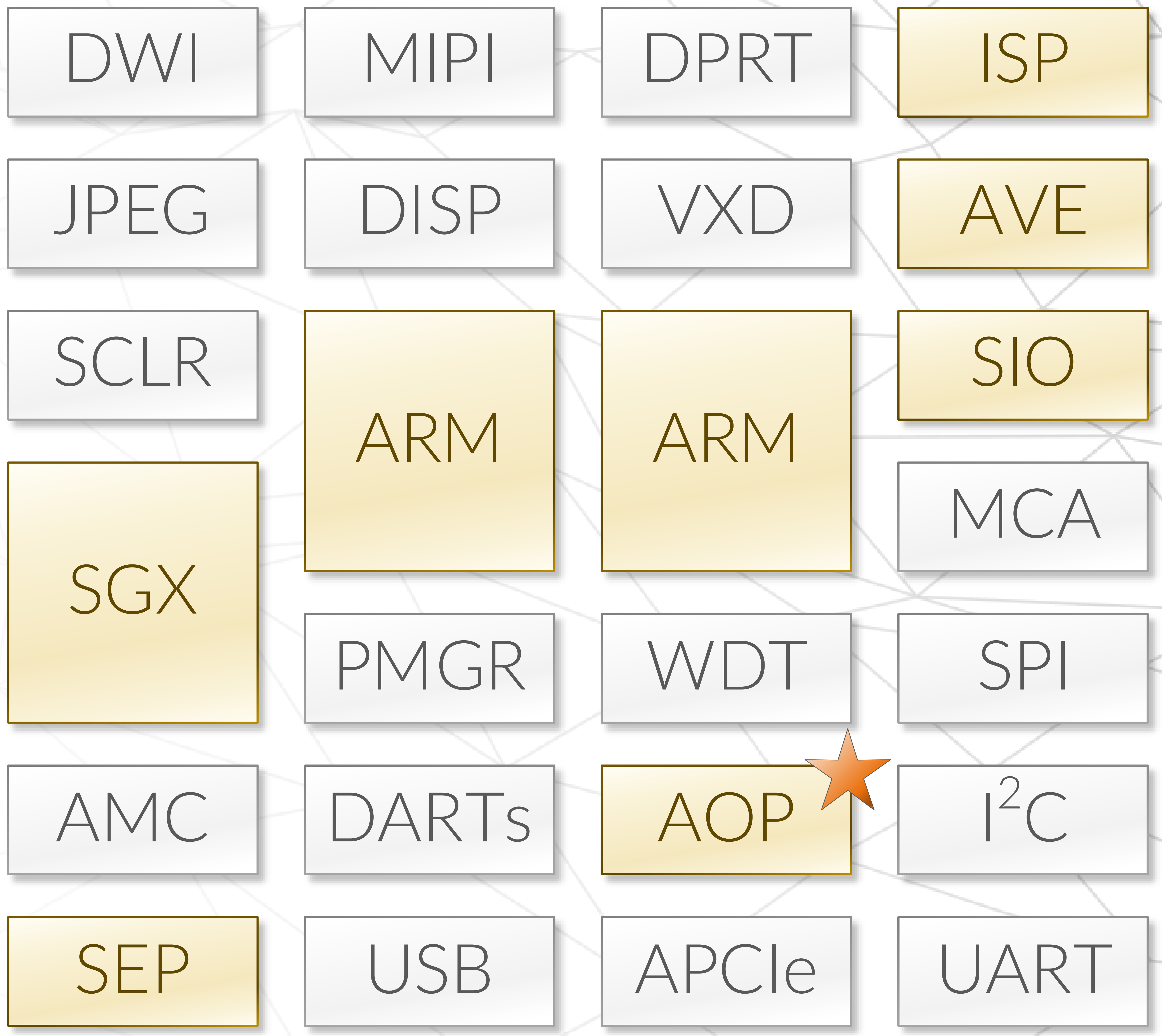
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, SEP



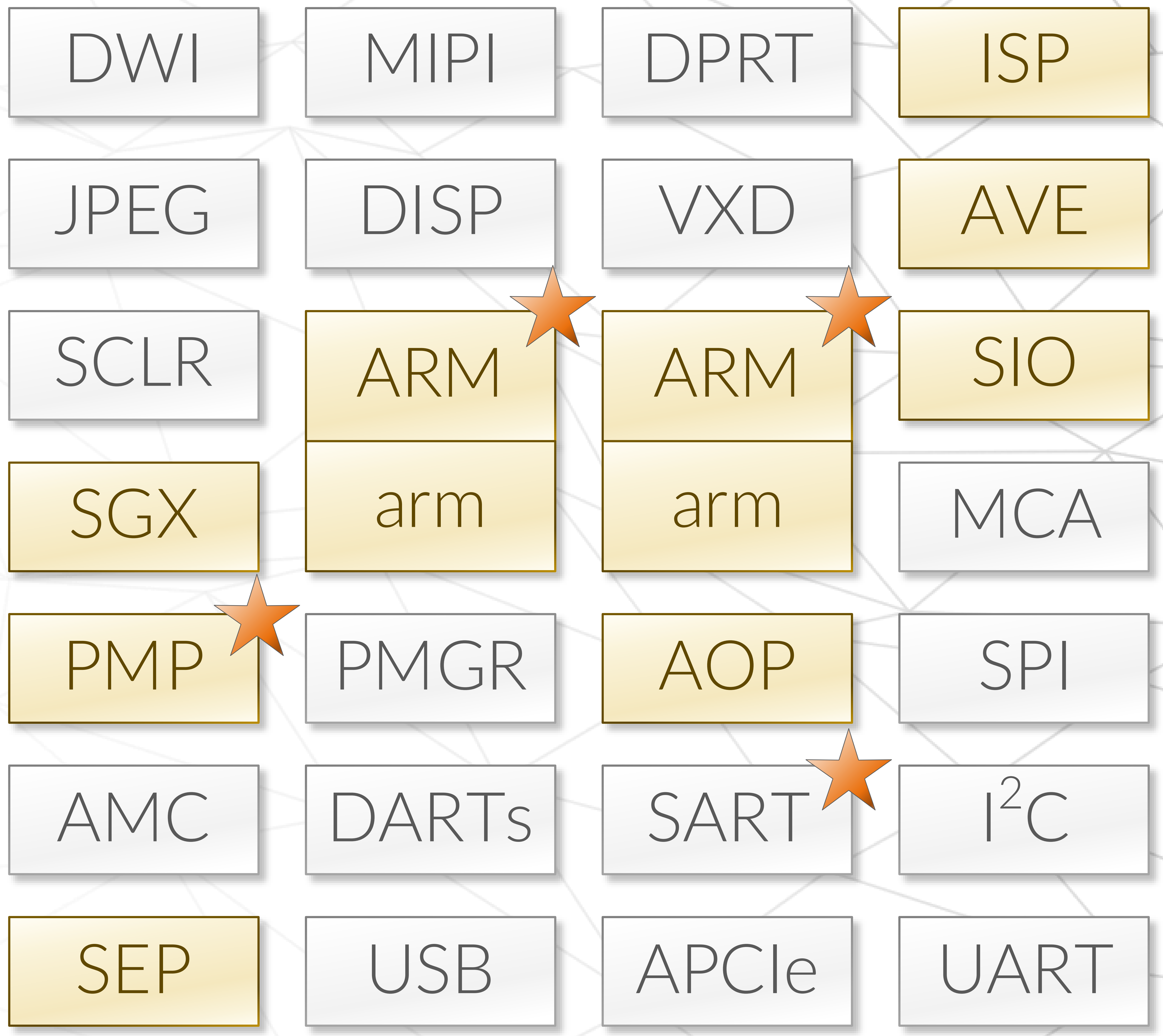
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP



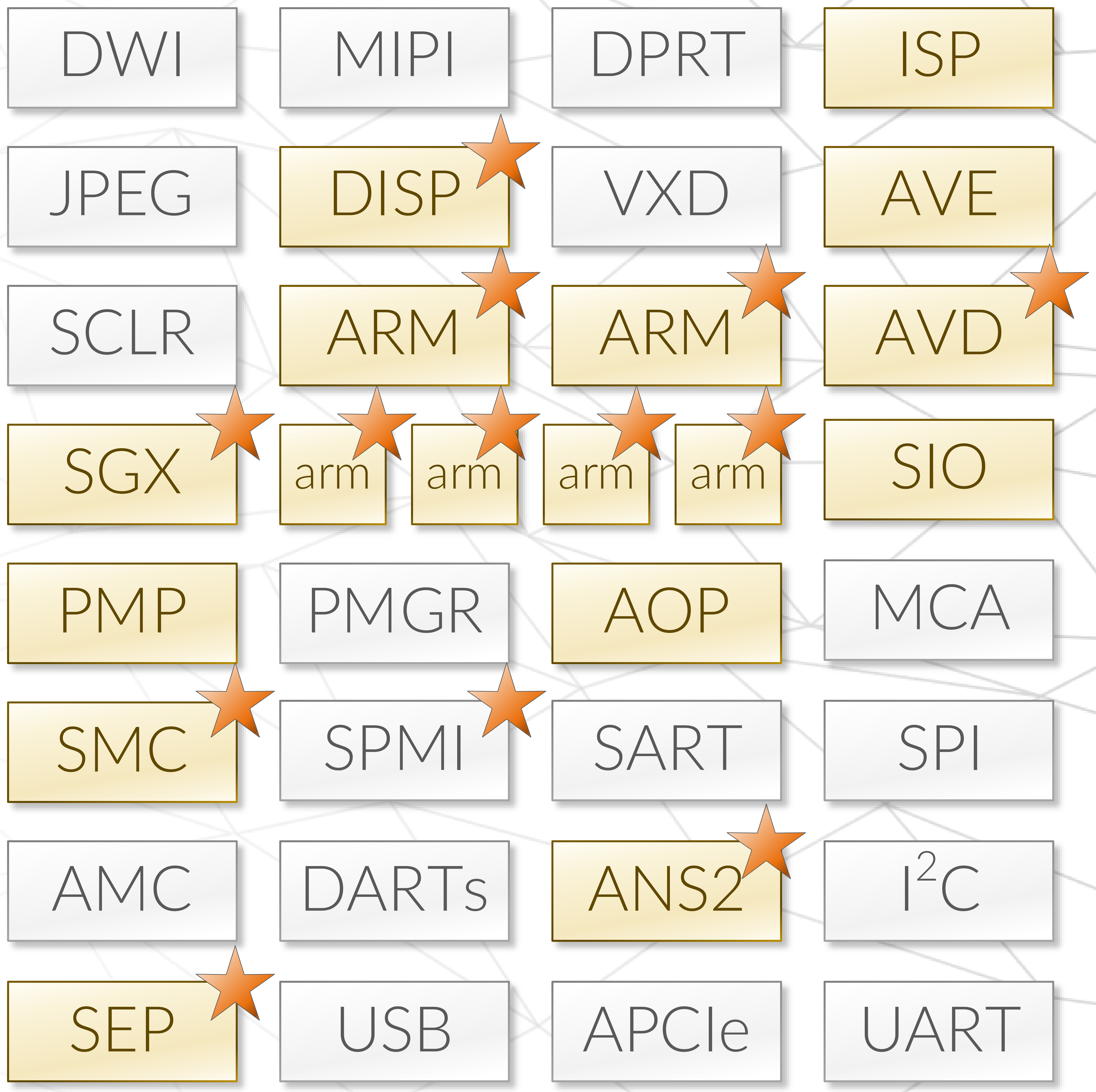
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, no EL3



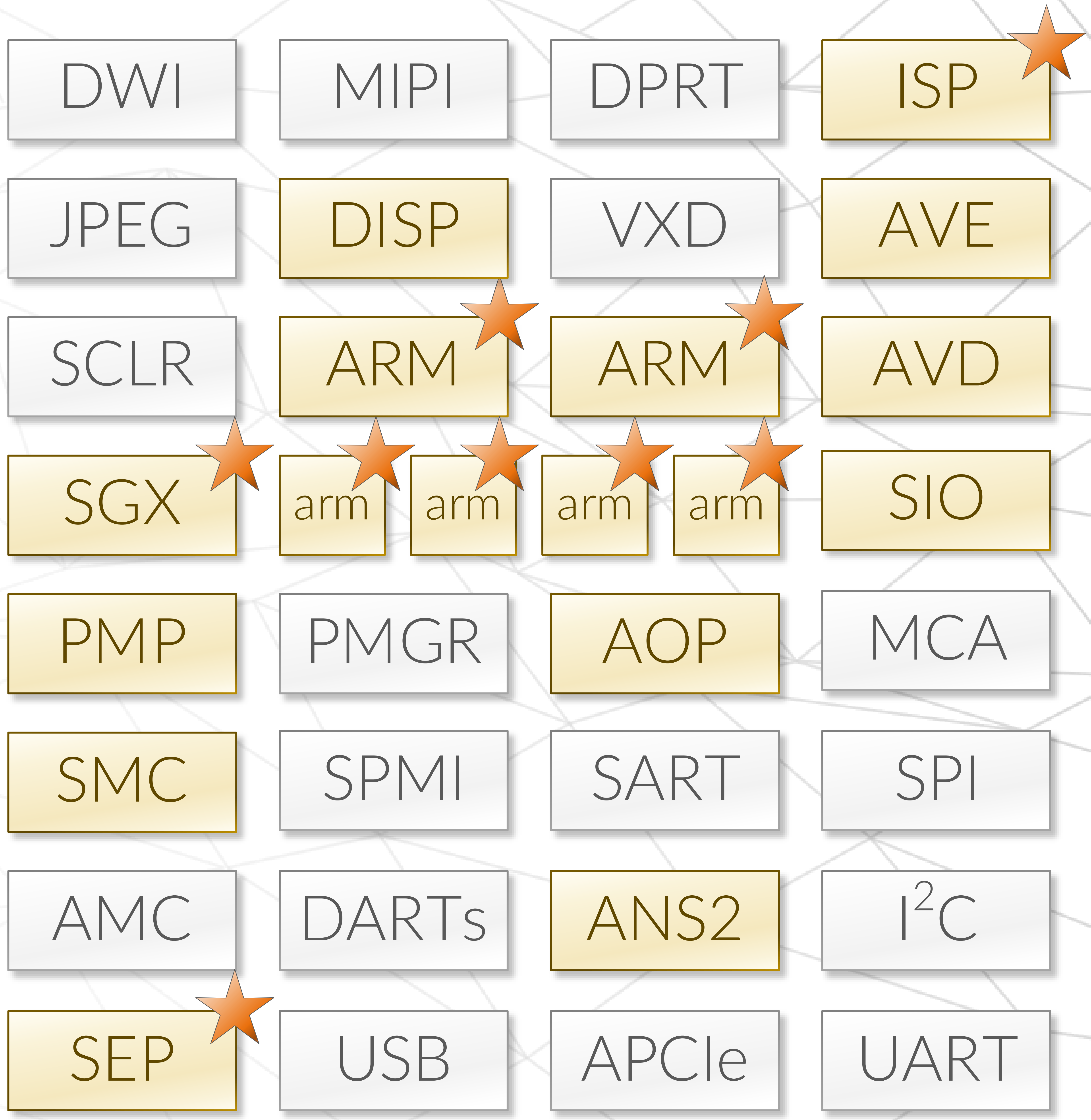
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, SART, no EL3
- A11: 6-core, SMC, ANS2



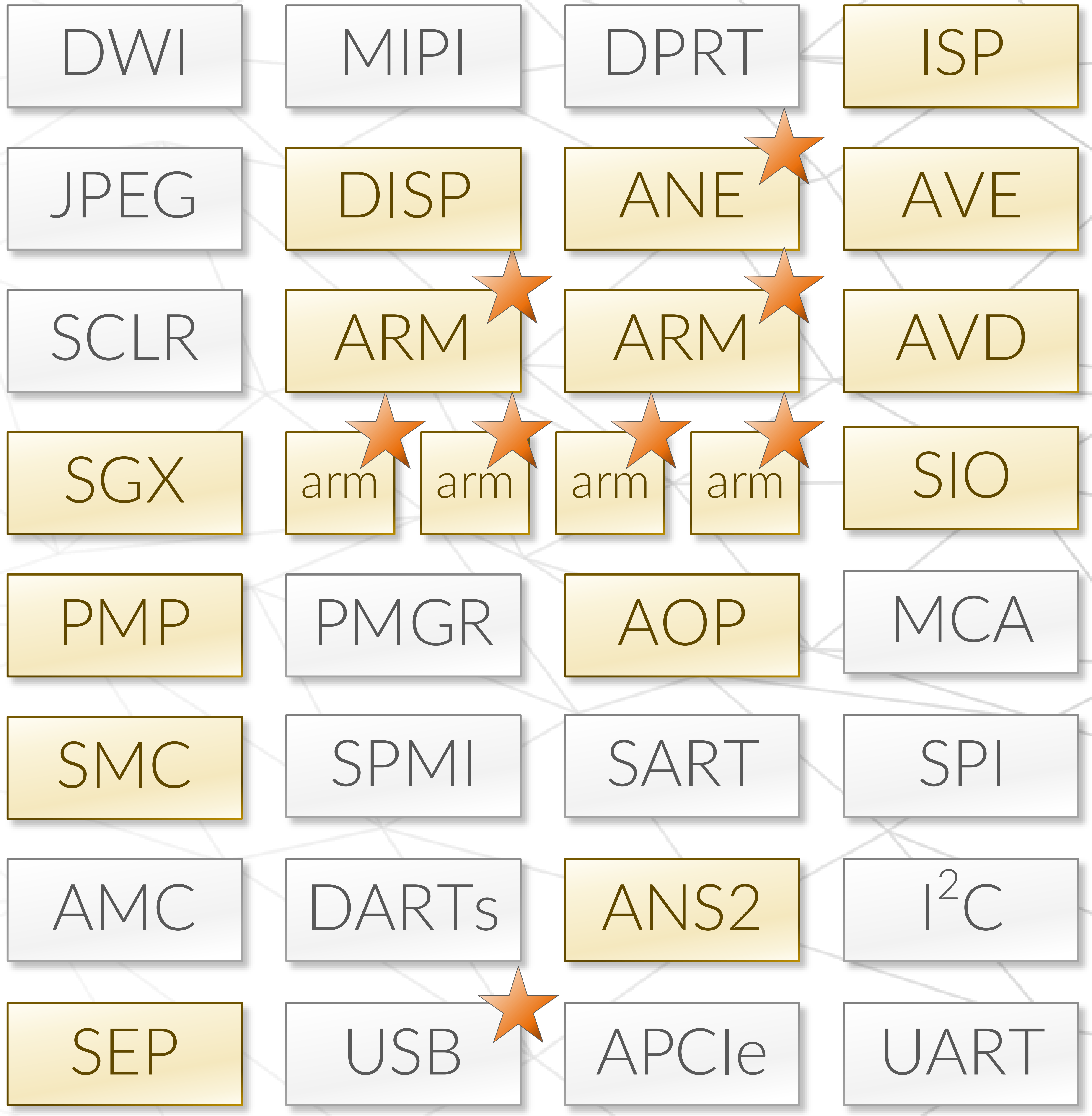
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, SART, no EL3
- A11: 6-core, SMC, ANS2
- A12: ARMv8.3-PAuth



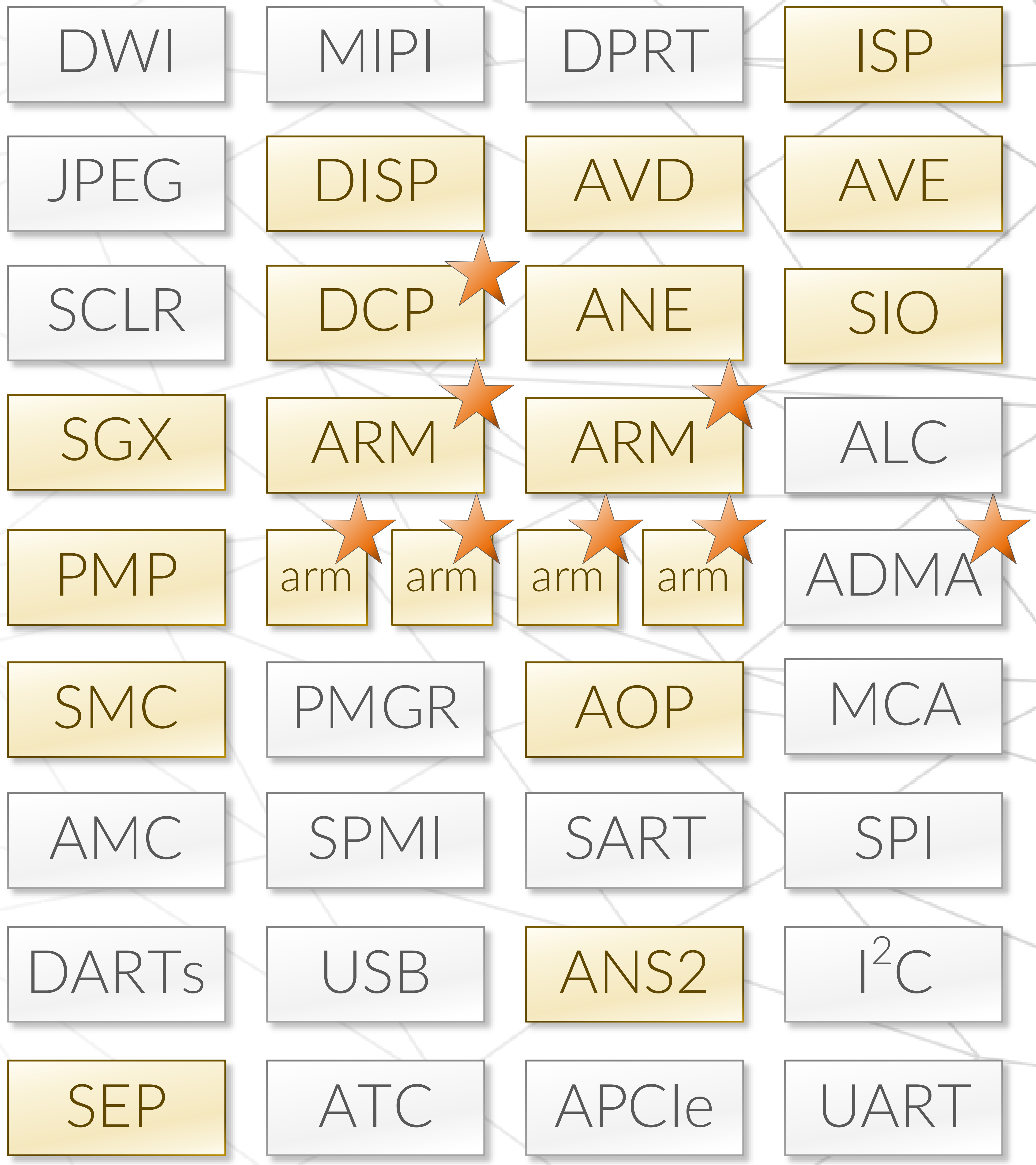
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, SART, no EL3
- A11: 6-core, SMC, ANS2
- A12: ARMv8.3-Pauth
- A13: PPL, ANE



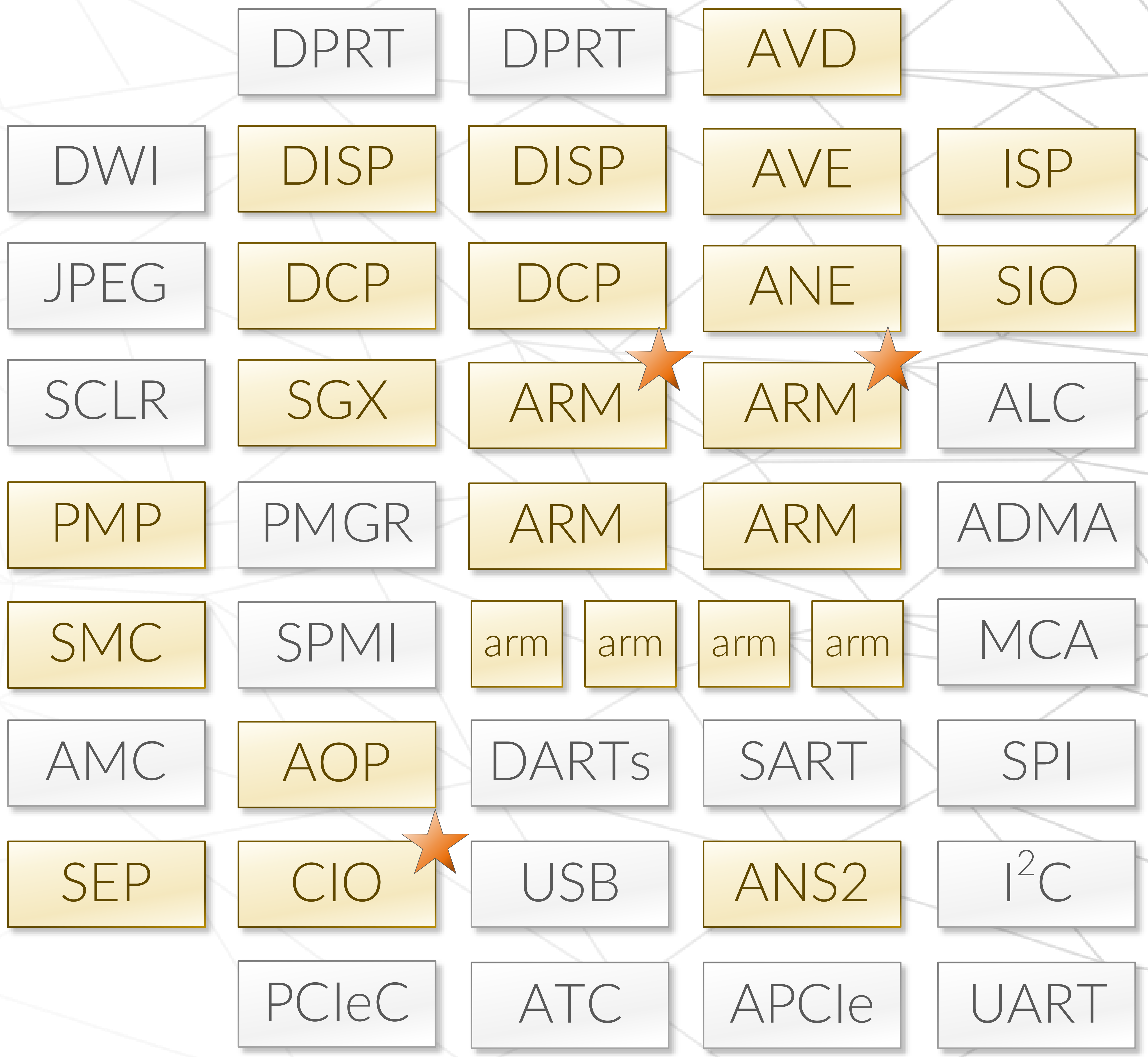
Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, SART, no EL3
- A11: 6-core, SMC, ANS2
- A12: ARMv8.3-Pauth
- A13: PPL, ANE
- A14: AMX, ADMA, DCP



Apple SoCs

- A4-A6: 32-bit
- A7-A8: 64-bit, EL3, SEP
- A9: NVMe, AOP
- A10: PMP, SART, no EL3
- A11: 6-core, SMC, ANS2
- A12: ARMv8.3-Pauth
- A13: PPL, ANE
- A14: AMX, ADMA, DCP
- M1: 8-core, USB4



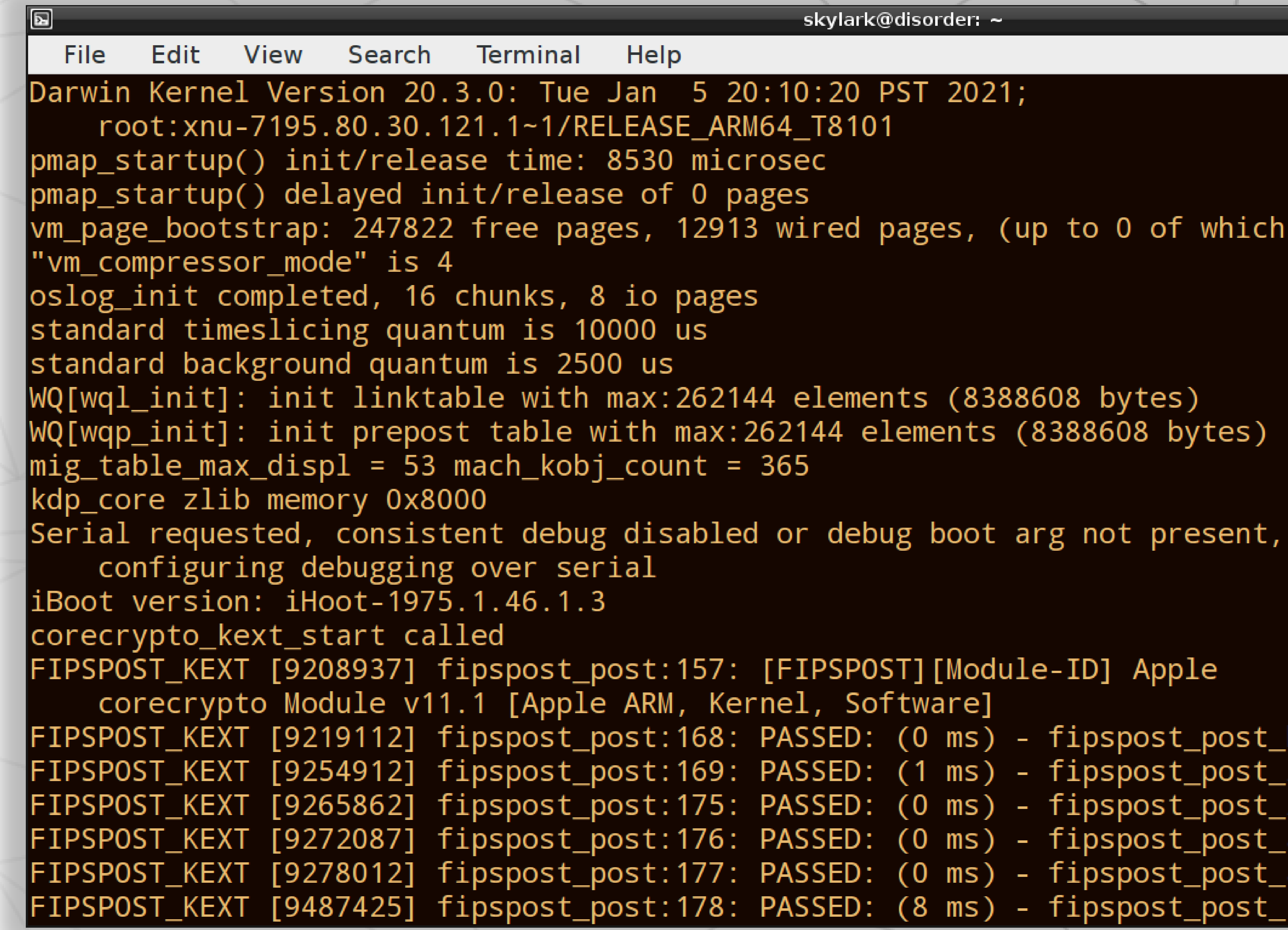
Modeling for RE

Corellium models

- We try to model hardware at register level
- Needs significant RE work
- Learn new things from every firmware update

Internal M1 model

- Offshoot of our commercial A14 model
- Added USB4 just to reverse engineer it



```
Darwin Kernel Version 20.3.0: Tue Jan  5 20:10:20 PST 2021;
    root:xnu-7195.80.30.121.1~1/RELEASE_ARM64_T8101
pmap_startup() init/release time: 8530 microsec
pmap_startup() delayed init/release of 0 pages
vm_page_bootstrap: 247822 free pages, 12913 wired pages, (up to 0 of which
"vm_compressor_mode" is 4
oslog_init completed, 16 chunks, 8 io pages
standard timeslicing quantum is 10000 us
standard background quantum is 2500 us
WQ[wql_init]: init linktable with max:262144 elements (8388608 bytes)
WQ[wqp_init]: init prepost table with max:262144 elements (8388608 bytes)
mig_table_max_displ = 53 mach_kobj_count = 365
kdp_core zlib memory 0x8000
Serial requested, consistent debug disabled or debug boot arg not present,
    configuring debugging over serial
iBoot version: iHoot-1975.1.46.1.3
corecrypto_kext_start called
FIPSPOST_KEXT [9208937] fipspost_post:157: [FIPSPOST][Module-ID] Apple
    corecrypto Module v11.1 [Apple ARM, Kernel, Software]
FIPSPOST_KEXT [9219112] fipspost_post:168: PASSED: (0 ms) - fipspost_post_
FIPSPOST_KEXT [9254912] fipspost_post:169: PASSED: (1 ms) - fipspost_post_
FIPSPOST_KEXT [9265862] fipspost_post:175: PASSED: (0 ms) - fipspost_post_
FIPSPOST_KEXT [9272087] fipspost_post:176: PASSED: (0 ms) - fipspost_post_
FIPSPOST_KEXT [9278012] fipspost_post:177: PASSED: (0 ms) - fipspost_post_
FIPSPOST_KEXT [9487425] fipspost_post:178: PASSED: (8 ms) - fipspost_post_
```

Modeling for RE

How to validate models?

- Test against different versions of firmware

    … or…

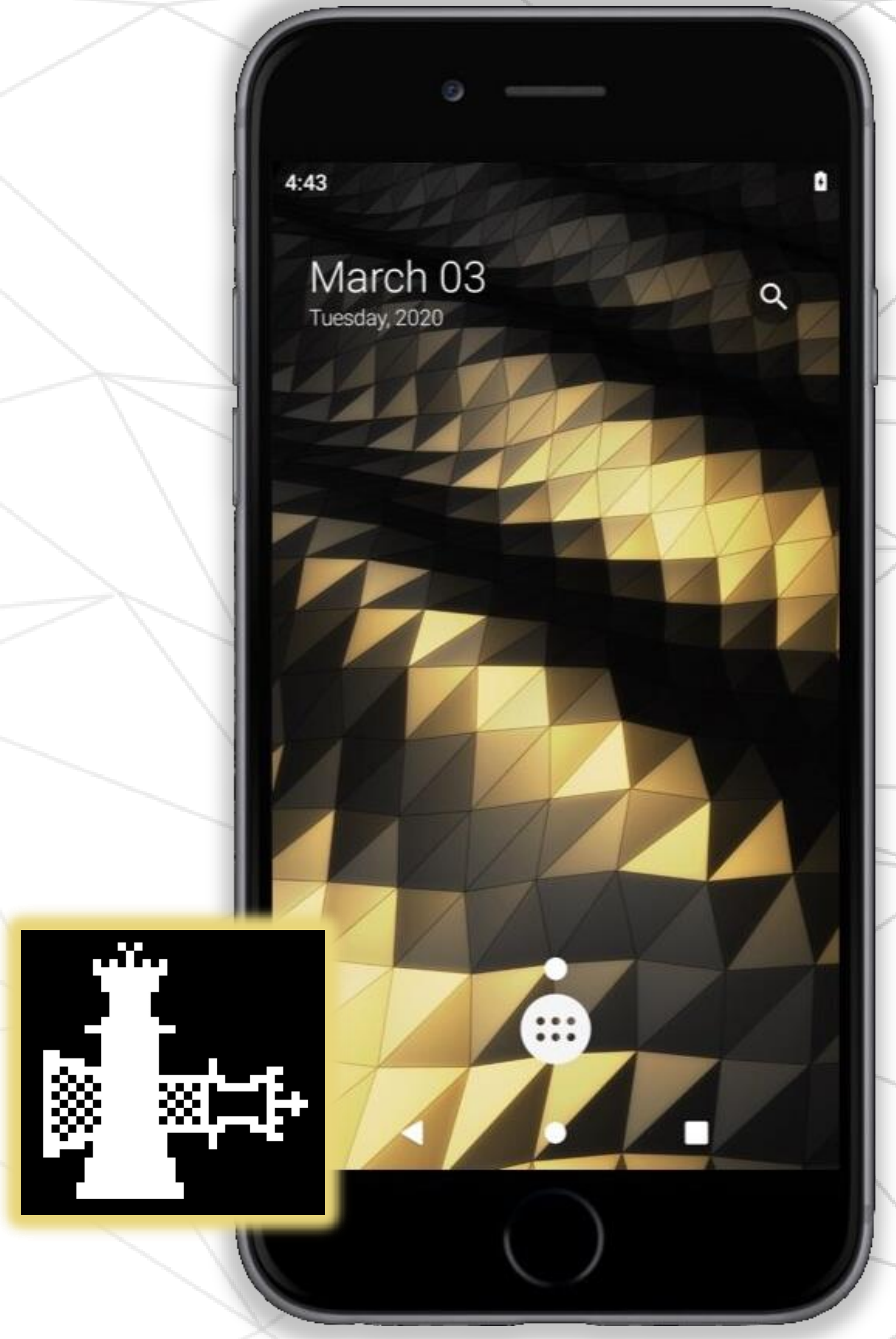
- Write software based on your model
- Run it on real hardware

Our chance: checkra1n

- First time we could run our own kernel
- Immediately started on Linux port to A10

Apple's newest SoC: M1

- Unprecedented open bootloader on top-of-the-line



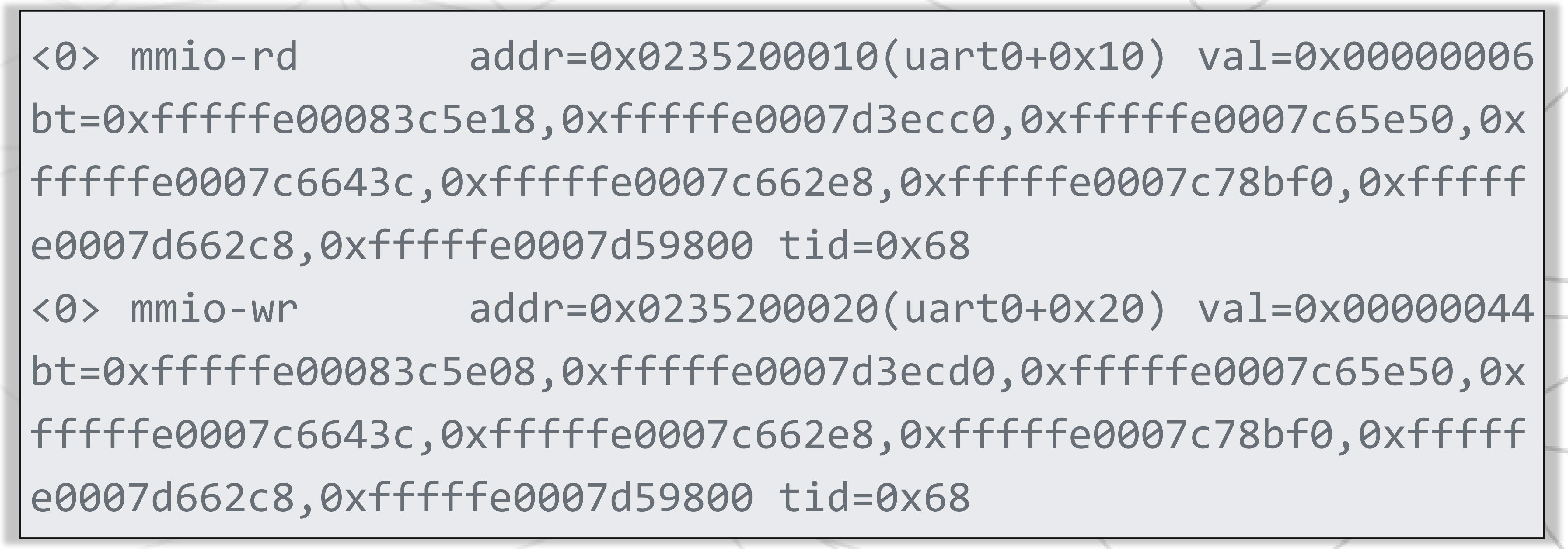
Get started

Gather information

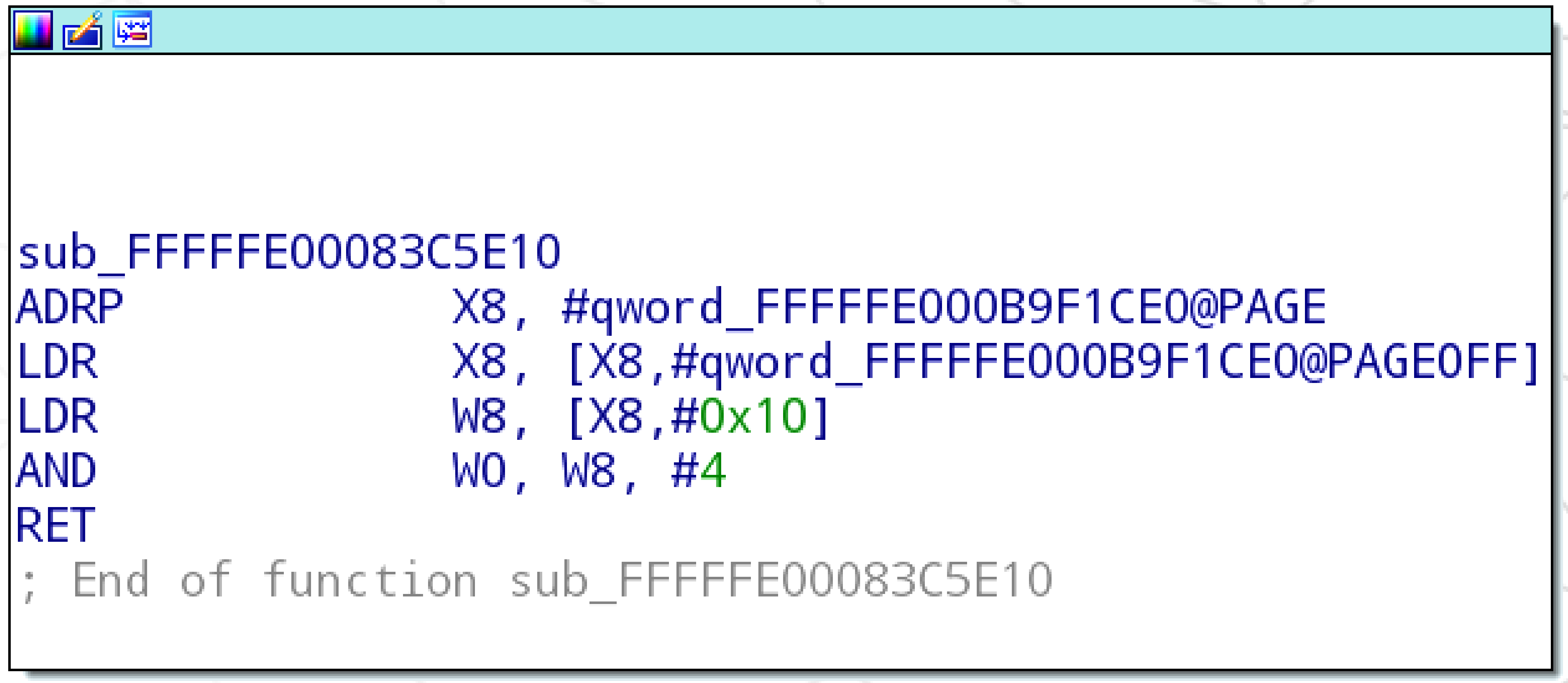
- Device tree
- `ioreg -w0 -l`
- Kernel strings

Generate stub model

- MMIO range for every device in tree
- Print / log accesses
- Add call backtrace to each access
- … maybe also a thread ID



```
<0> mmio-rd       addr=0x0235200010(uart0+0x10) val=0x00000006
bt=0xfffffe00083c5e18,0xfffffe0007d3ecc0,0xfffffe0007c65e50,0x
fffffe0007c6643c,0xfffffe0007c662e8,0xfffffe0007c78bf0,0xfffff
e0007d662c8,0xfffffe0007d59800 tid=0x68
<0> mmio-wr       addr=0x0235200020(uart0+0x20) val=0x00000044
bt=0xfffffe00083c5e08,0xfffffe0007d3ecd0,0xfffffe0007c65e50,0x
fffffe0007c6643c,0xfffffe0007c662e8,0xfffffe0007c78bf0,0xfffff
e0007d662c8,0xfffffe0007d59800 tid=0x68
```



```
sub_FFFFFE00083C5E10
ADRP                X8, #qword_FFFFFE000B9F1CE0@PAGE
LDR                 X8, [X8,#qword_FFFFFE000B9F1CE0@PAGEOFF]
LDR                 W8, [X8,#0x10]
AND                 W0, W8, #4
RET
; End of function sub_FFFFFE00083C5E10
```

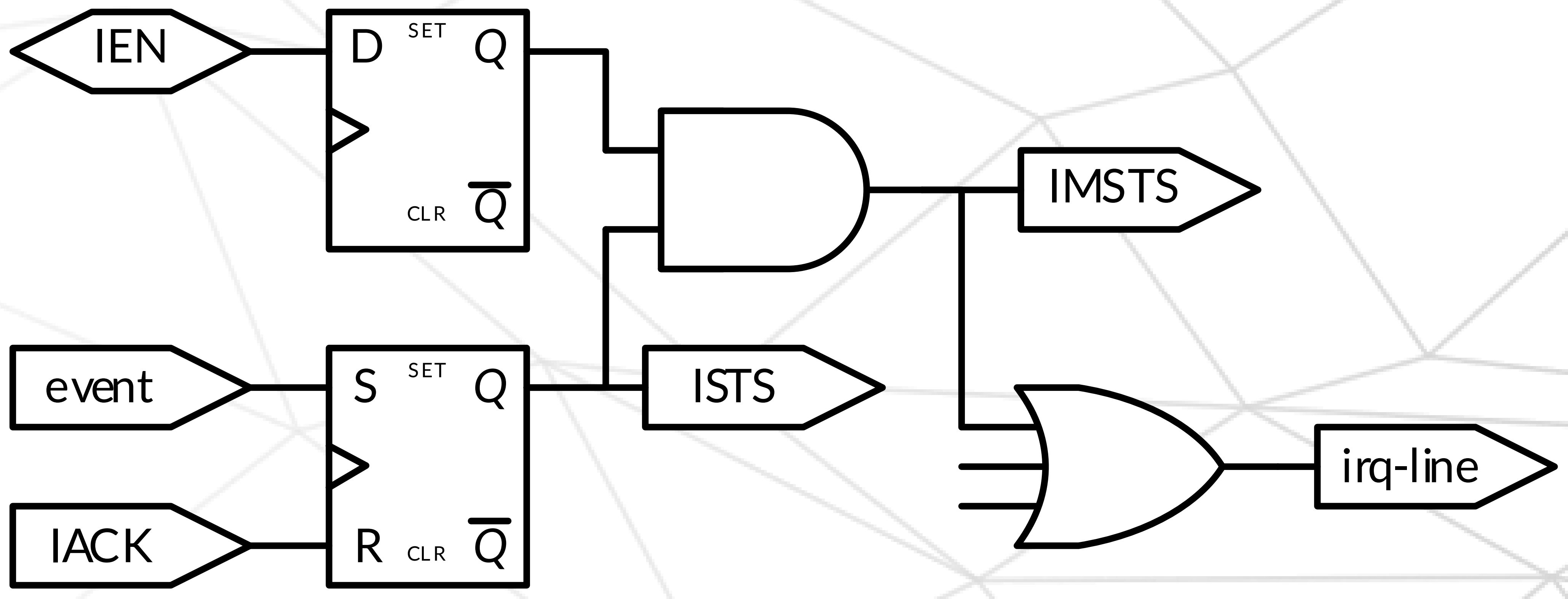
IRQ handling

Hardware resources

- Interrupt status register (ISTS)
- Interrupt enable / mask register (IEN / IMSK)
- Optional acknowledge (IACK)
- Optional masked status (IMSTS)
- Hardware IRQline

Reverse engineer it

- Assert the IRQ line
- First access will be ISTS or IMSTS
- If status is always written back, it's IACK
- If sw clears / sets bits incrementally, it's IEN / IMSK



```
<1>   mmio-rd              addr=0x023b102004(aic_0+0x2004)
val=0x00010267  bt=0xfffffe0008d5aef8,0xfffffe0008d5a5ec
```
^ read status in top level interrupt controller

```
<3>   mmio-rd                   addr=0x0235104008(spi1+0x8)
val=0x00008003  bt=0xfffffe00090987d0,0xfffffe0009097c54
```
^ read ISTS register

```
<3>   mmio-wr                   addr=0x0235104008(spi1+0x8)
val=0x00008003  bt=0xfffffe00090987dc,0xfffffe0009097c90
```
^ write IACK – frequently same register as ISTS

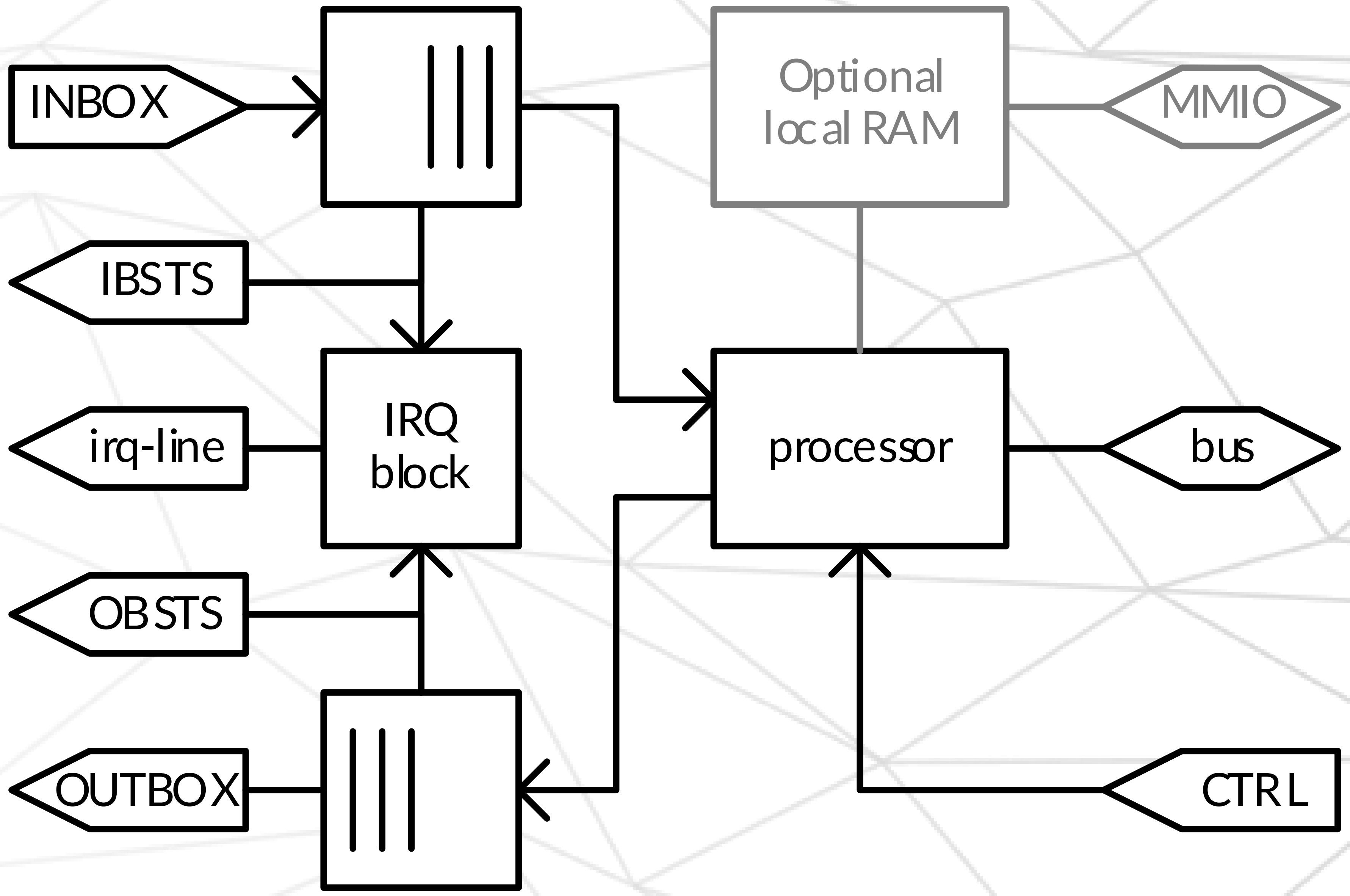
Coprocessors

Hardware resources

- Inbox FIFO / register
- Outbox FIFO / register
- Firmware load and run / stop
- Interrupt controller

Reverse engineer it

- Which register and bit is polled around writing to the inbox?
- Which register and bit is checked when outbox IRQ happens?



```
<1> mmio-rd addr=0x023bc00044(pmp_0+0x44) val=0x00000000
<1> mmio-wr addr=0x023bc00044(pmp_0+0x44) val=0x00000010
          ^ start processor
<1> mmio-rd addr=0x023bc08114(pmp_0+0x8114) val=0x00100001
<1> mmio-rd addr=0x023bc08830(pmp_0+0x8830)
          val=0x00100000000b000b,0x0010000000000000
          ^ read message
<1> mmio-wr  addr=0x023bc08800(pmp_0+0x8800)
          val=0x0020001000b000b,0x0000000000000000
<1> mmio-rd  addr=0x023bc08110(pmp_0+0x8110) val=0x00020000
          ^ write message
```

Coprocessors

Logical view

- Endpoints
- RTBuddy initialization
- Request-response conventions
- Unsolicited events
- RTKit patch bay

Reverse engineer it

- Read ioreg carefully: endpoint ID to name
- RTBuddy protocol is pretty stable
- Trace endpoint traffic

```
[a10-pmp] start
[a10-pmp] out 00100000000b000b 0 <- EP 0 = RTBuddy
[a10-pmp] msg 0020000100b000b 0
     ^ RTBuddy version negotiation (here, 11.11)
[a10-pmp] out 00800000000000ff 0 <- EP 0..31
[a10-pmp] out 0080000100000001 0 <- EP 32..63, last
     ^ coprocessor lists endpoints
[a10-pmp] msg 0050000100000002 0
[a10-pmp] msg 0050000200000002 0
[a10-pmp] msg 0050000300000002 0
     ^ kernel starts endpoints
...
[a10-pmp] msg 0001000000004000 20 <- EP 32 = payload
[a10-pmp] out 2002000000000000 20
     ^ kernel sends transfer buffer address, gets ack
```

Coprocessors

RTBuddy endpoints

- 0 – management
- 1 – crash log
- 2 – system log
- 3 – debug
- 4 – IOKit report

Detailed crash log

- Assertions tell you what you should do
- Has a mailbox message log
- Thread stack backtraces

```
Crashlog (14704 bytes):
    tag 'Cstr', 112 bytes
        assert  failed:  [7446]:duplicate  tag  error  for  tag  0,
host_id 0x0. status_reg: 0x8000000
                                 ...
    tag 'Ccst', 160 bytes
        [ 017390 016db8 00a314 01d2d8 01d30c 026304 025dc8 011588
00aa84 008df0 129908 12a098 12a098 1272a4 11fec8 041d6c 00e358 ]
                                 ...
    tag 'Cmbx', 3120 bytes
 2000111111101:0000000000000000 t=0000000000000000
        01:0010800000000000 t=000000000648bfd8
                                 ...
    tag 'Crtk', 1728 bytes
        rtk_ep_work                [ 00dc14 00e918 0176c0 00e358 ]
        smc_mbox_work_loop         [ 00dc14 00e918 0176c0 00e358 ]
                                 ...
NVME 0
  CMD status: 0x08000000, info: 0x00000000
  CC: 0x00460001, CSTS: 0x00000001
  Admin SQ: 0x80399c000, size: 32, tail/head: 7/7
                                 ...
```

IOMMUs

That was a weird address

- There’s no RAM at 0x4000 physical
- So what was that?
- DART in front of almost every DMA device
- DARTs keep getting tweaked

IOMMUs are nice

- They make the system a bit more resistant
- They make RE *easier!*
- Advance notification of relevant memory areas
- Cleaned up after you’re supposed to be done
- Lots of information leaked there – look at this table of MMIO

```
[a10-pmp] msg 0001000000004000 20 <- EP 32 = payload
[a10-pmp] out 2002000000000000 20
     ^ kernel sends transfer buffer address, gets ack
```

```
(gdb) mon iommu pmp
Dump of IOMMU `pmp' stream 0:
  0000000000004000-0000000000007fff -> 080ed38000 rw
  0000000000008000-000000000000bfff -> 080ed34000 rw
  000000000000c000-000000000000ffff -> 080ed30000 rw
  0000000000010000-0000000000013fff -> 080ed2c000 rw
  00000000c0000000-00000000c0003fff -> 0200024000 rw
  00000000c0004000-00000000c0007fff -> 0200064000 rw
  00000000c0008000-00000000c000bfff -> 02000a4000 rw
  00000000c000c000-00000000c000ffff -> 02000e4000 rw
  00000000c0010000-00000000c0013fff -> 0200158000 rw
  00000000c0014000-00000000c0017fff -> 0200204000 rw
  00000000c0018000-00000000c001bfff -> 0200244000 rw
  00000000c001c000-00000000c001ffff -> 0200284000 rw
  00000000c0020000-00000000c0023fff -> 02002c4000 rw
...
```

IOMMUs

There's a lot of IOMMUs

- A8: 7 DARTs
- A11: 11 DARTs
- M1: 24 DARTs
- … and GPU has its own GART

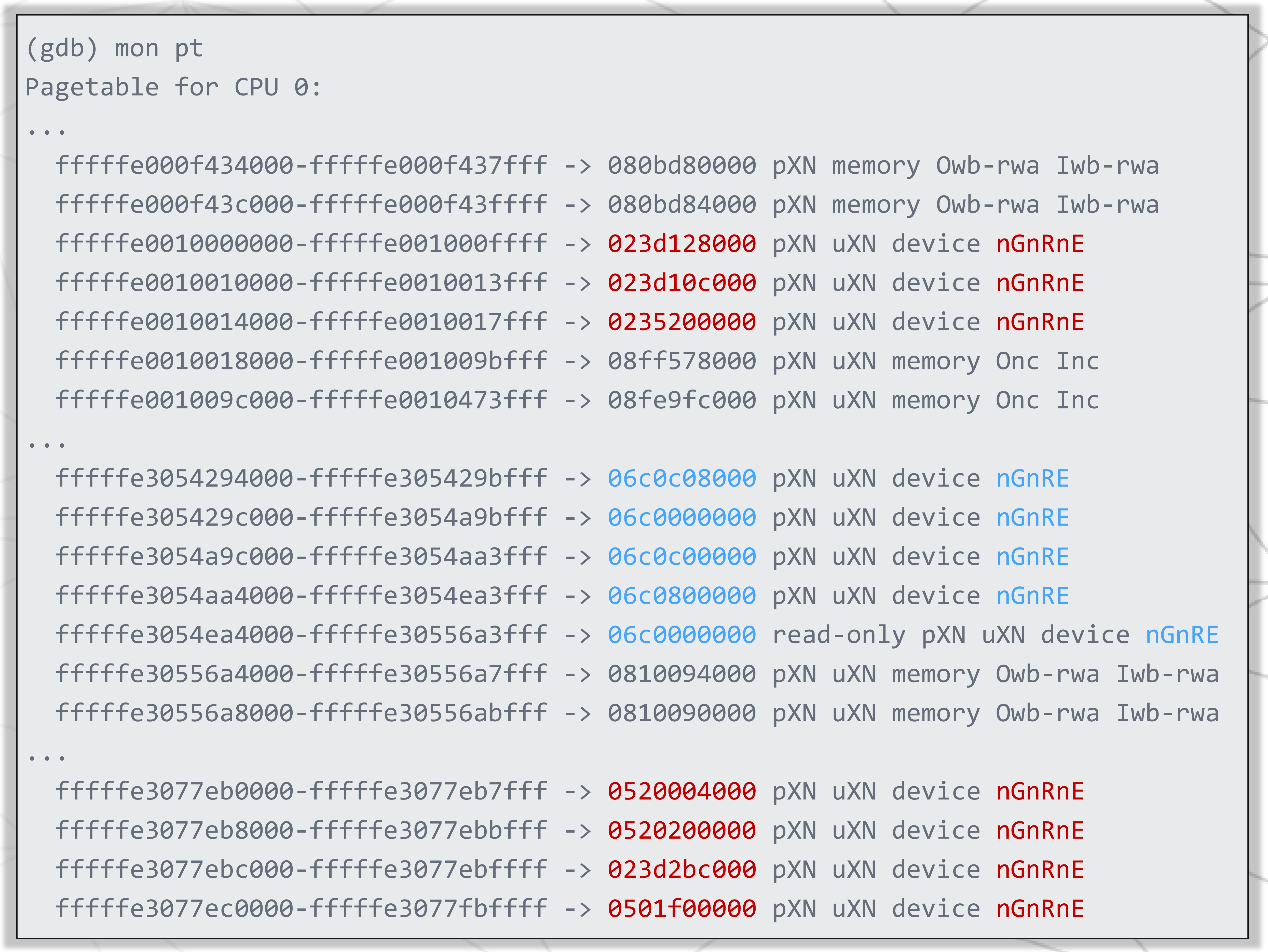
Some IOMMUs have extras

- Multiple translation paths
- Extra address filters
- Bypass path for some streams
- Different page sizes on some chips

```
(gdb) mon iommu
List of IOMMUs in system:
    apciec1                 dart-16k        15
    acio1                   dart-16k        0,1,15
    usb1                    dart-16k        0,1,3
    apciec0                 dart-16k        15
    acio0                   dart-16k        0,1,15
    usb0                    dart-16k        0,1,3
    ave                     dart-16k        0,15
    avd                     dart-16k        0,15
    jpeg1                   dart-16k        0
    jpeg0                   dart-16k        0
    scaler                  dart-16k        0
    dispext                 dart-16k        0,4,12,15
    dcpext                  dart-16k        0,12,15
    disp                    dart-16k        0,4,12,15
    dcp                     dart-16k        0,12,15
    apcie2                  dart-16k        1,2
    apcie1                  dart-16k        1,2
    apcie0                  dart-16k        1,2
    sep                     dart-16k        0,15
    ane                     dart-16k        0,13,15
    isp                     dart-16k        0,15
    pmp                     dart-16k        0,15
    sio                     dart-16k        0,1,15
    aop                     dart-16k        0,15
```

Memory types

- **ARM defines device memory types**
  - nGnRnE: most restrictive, requires write-ack
  - nGnRE: less restrictive, posted writes
  - ... and others
- **Apple chips *really care!***
  - If you use nGnRE for on-chip I/O (Linux default), writes get silently ignored
  - If you use nGnRnE for PCIe I/O, writes also get silently ignored
  - (PCIe uses posted writes to MMIO)
  - Look at XNU pagetable on the right

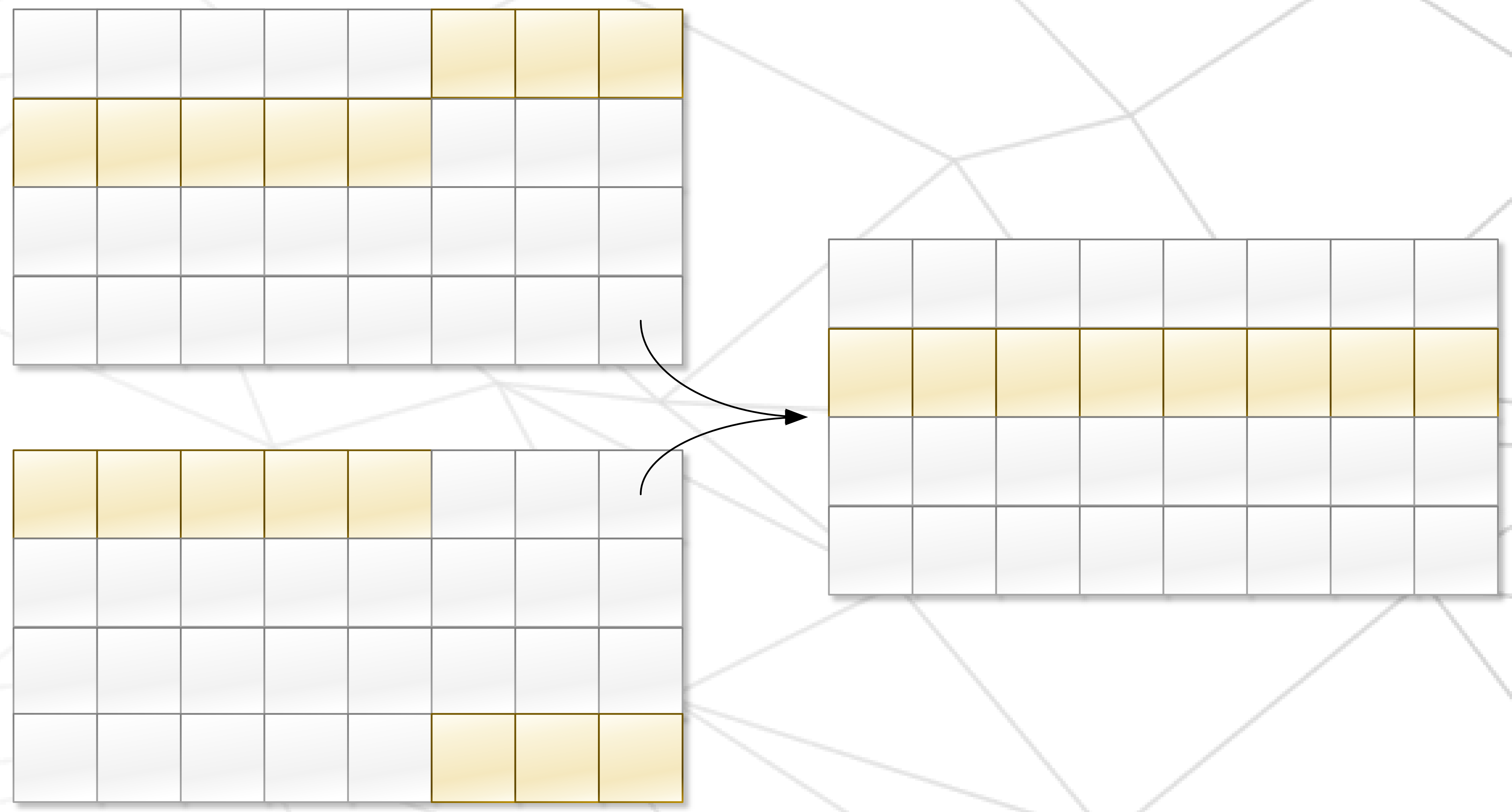


```
(gdb) mon pt
Pagetable for CPU 0:
...
  fffffe000f434000-fffffe000f437fff -> 080bd80000 pXN memory Owb-rwa Iwb-rwa
  fffffe000f43c000-fffffe000f43ffff -> 080bd84000 pXN memory Owb-rwa Iwb-rwa
  fffffe0010000000-fffffe001000ffff -> 023d128000 pXN uXN device nGnRnE
  fffffe0010010000-fffffe0010013fff -> 023d10c000 pXN uXN device nGnRnE
  fffffe0010014000-fffffe0010017fff -> 0235200000 pXN uXN device nGnRnE
  fffffe0010018000-fffffe001009bfff -> 08ff578000 pXN uXN memory Onc Inc
  fffffe001009c000-fffffe0010473fff -> 08fe9fc000 pXN uXN memory Onc Inc
...
  fffffe3054294000-fffffe305429bfff -> 06c0c08000 pXN uXN device nGnRE
  fffffe305429c000-fffffe3054a9bfff -> 06c0000000 pXN uXN device nGnRE
  fffffe3054a9c000-fffffe3054aa3fff -> 06c0c00000 pXN uXN device nGnRE
  fffffe3054aa4000-fffffe3054ea3fff -> 06c0800000 pXN uXN device nGnRE
  fffffe3054ea4000-fffffe30556a3fff -> 06c0000000 read-only pXN uXN device nGnRE
  fffffe30556a4000-fffffe30556a7fff -> 0810094000 pXN uXN memory Owb-rwa Iwb-rwa
  fffffe30556a8000-fffffe30556abfff -> 0810090000 pXN uXN memory Owb-rwa Iwb-rwa
...
  fffffe3077eb0000-fffffe3077eb7fff -> 0520004000 pXN uXN device nGnRnE
  fffffe3077eb8000-fffffe3077ebbfff -> 0520200000 pXN uXN device nGnRnE
  fffffe3077ebc000-fffffe3077ebffff -> 023d2bc000 pXN uXN device nGnRnE
  fffffe3077ec0000-fffffe3077fbffff -> 0501f00000 pXN uXN device nGnRnE
```

AMX

AMX vector extensions

- Invalid opcode space, most encoded in Xreg
- Corellium needed it for A14 sound output
- Unusual “ring vector” architecture
- Lazy state save / restore in kernel



Reverse engineer it

- Didn’t have an M1 yet (wasn’t out as a product)
- iPhone 12 *Accelerate.framework* uses it
- Write vDSP single-function tests, run in a VM on A14 model and dump AMX instruction streams
- FFT butterflies got us started

```
OP 0x00201188   AMXFMA 0x8000000009e00000    z30 = x0 * y0
OP 0x00201188   AMXFMA 0x8000000009f00040    z31 = x0 * y1
OP 0x00201188   AMXFMA 0x800000000be20080    z62 = x2 * y2
OP 0x00201188   AMXFMA 0x800000000bf200c0    z63 = x2 * y3
OP 0x002011a8   AMXFMS 0x8000000001e10040    z30 -= x1 * y1
OP 0x00201188   AMXFMA 0x8000000001f10000    z31 += x1 * y0
OP 0x002011a8   AMXFMS 0x8000000003e300c0    z62 -= x3 * y3
OP 0x00201188   AMXFMA 0x8000000003f30080    z63 += x3 * y2
```

Since then, Dougall Johnson’s work emerged. Have a look if you can find it!

NVMe IOMMU

IOMMU design tradeoffs

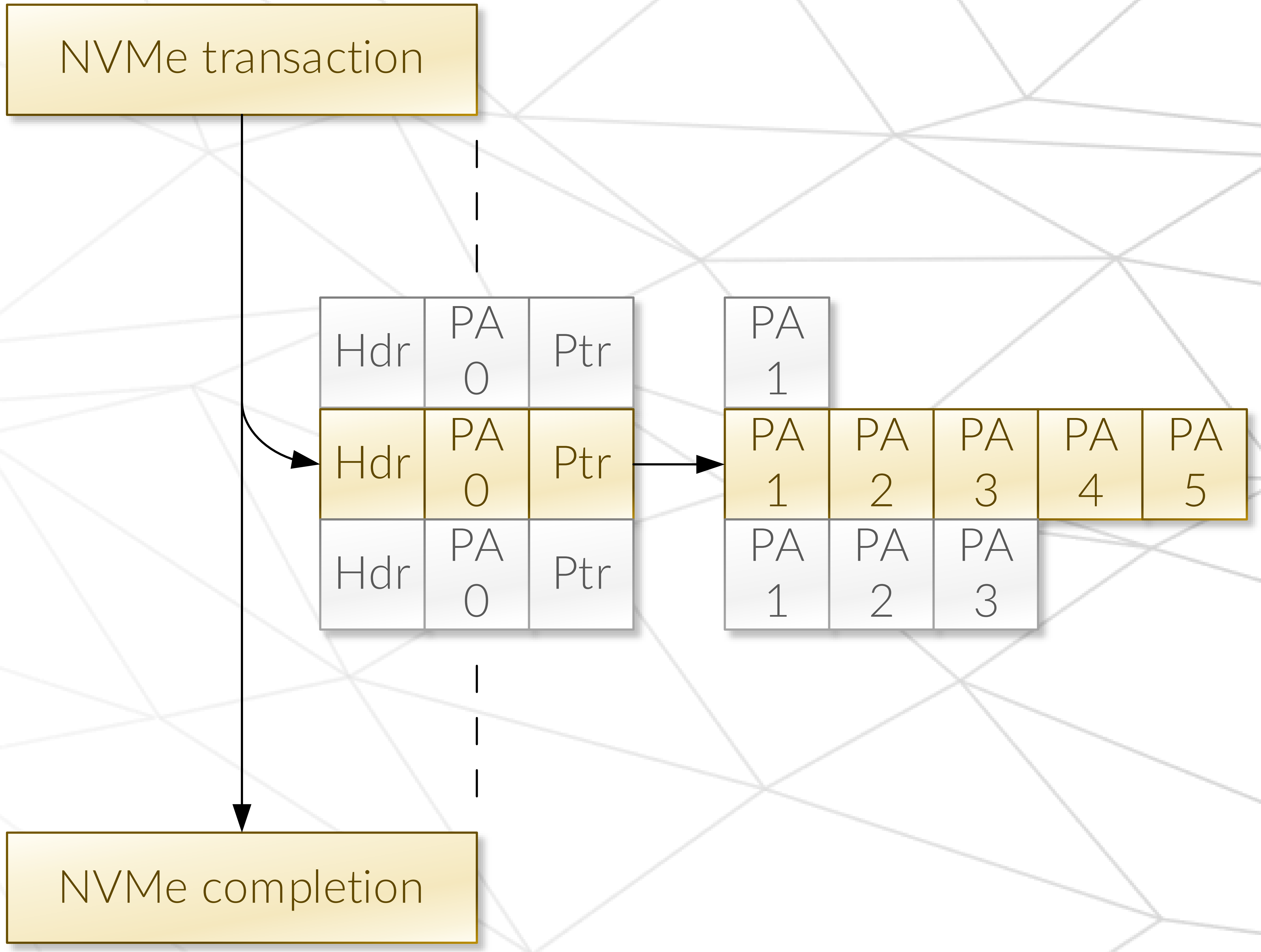
- Pagetable-style (DART) requires address space allocation, pagetable rewrites and flushes
- NAND access over NVMe happens in long blocks
- NVMMU uses NVMe transaction ID as slot index
- Each slot has a list of physical pages to map

Reverse engineer it

- Mostly by watching writes into the NVMMU table
- Read-write flag correlated with NVMe access type
- Still has a DART for non-data accesses

One more thing *(thanks, Apple)*

- A14/M1 use a non-standard NVMe queue format



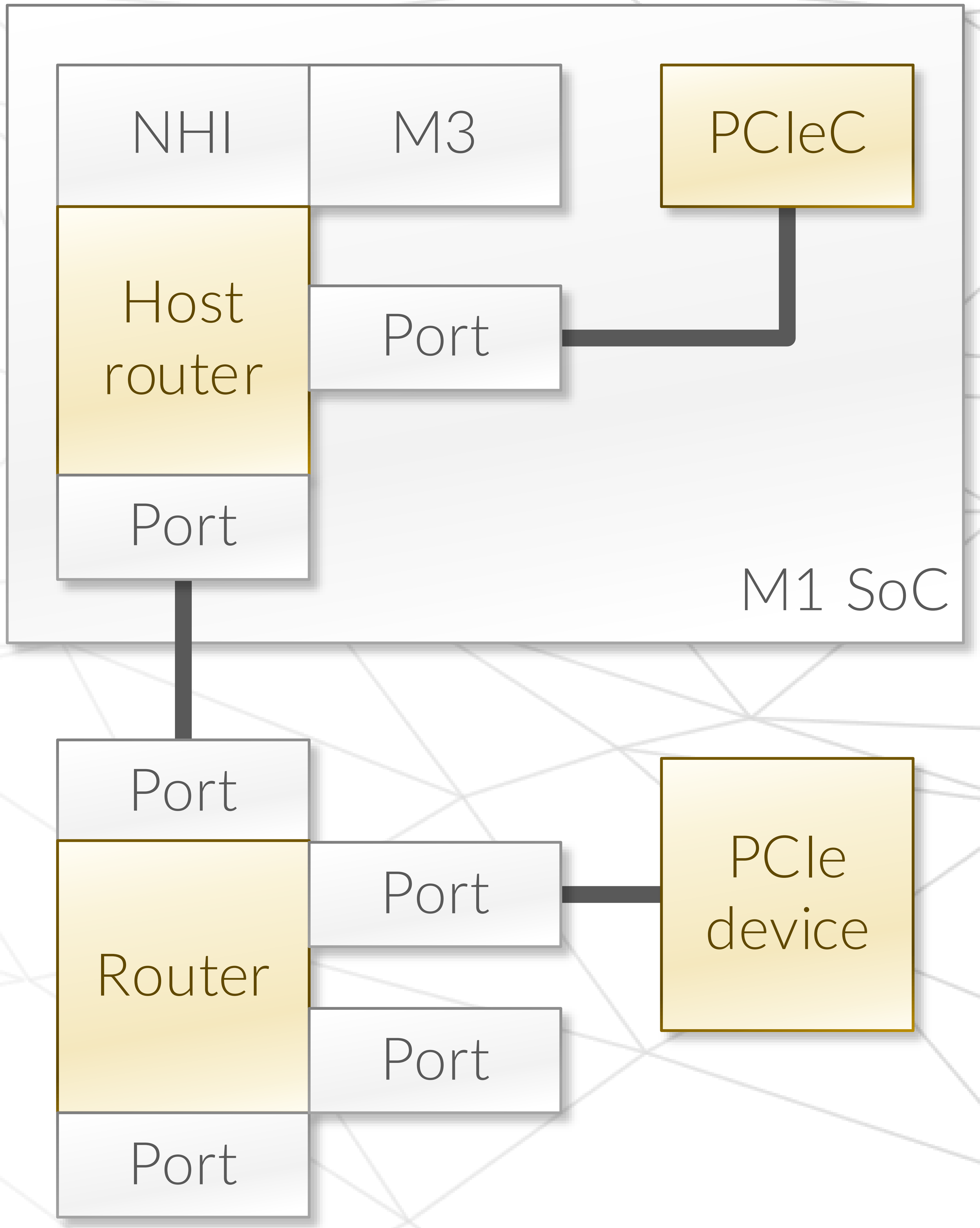
USB 4

New generation of Thunderbolt

- Same basic ideas
- High-speed link with packet routers
- PCIe, DisplayPort, USB tunneled
- Market debut in the M1

Reverse engineer it

- Pretty much required writing a model
- USB-IF released partial USB 4 spec
- Intel contributed USB 4 support to Linux based on Andreas' Noever's work
- … but Apple made it their own, as usual



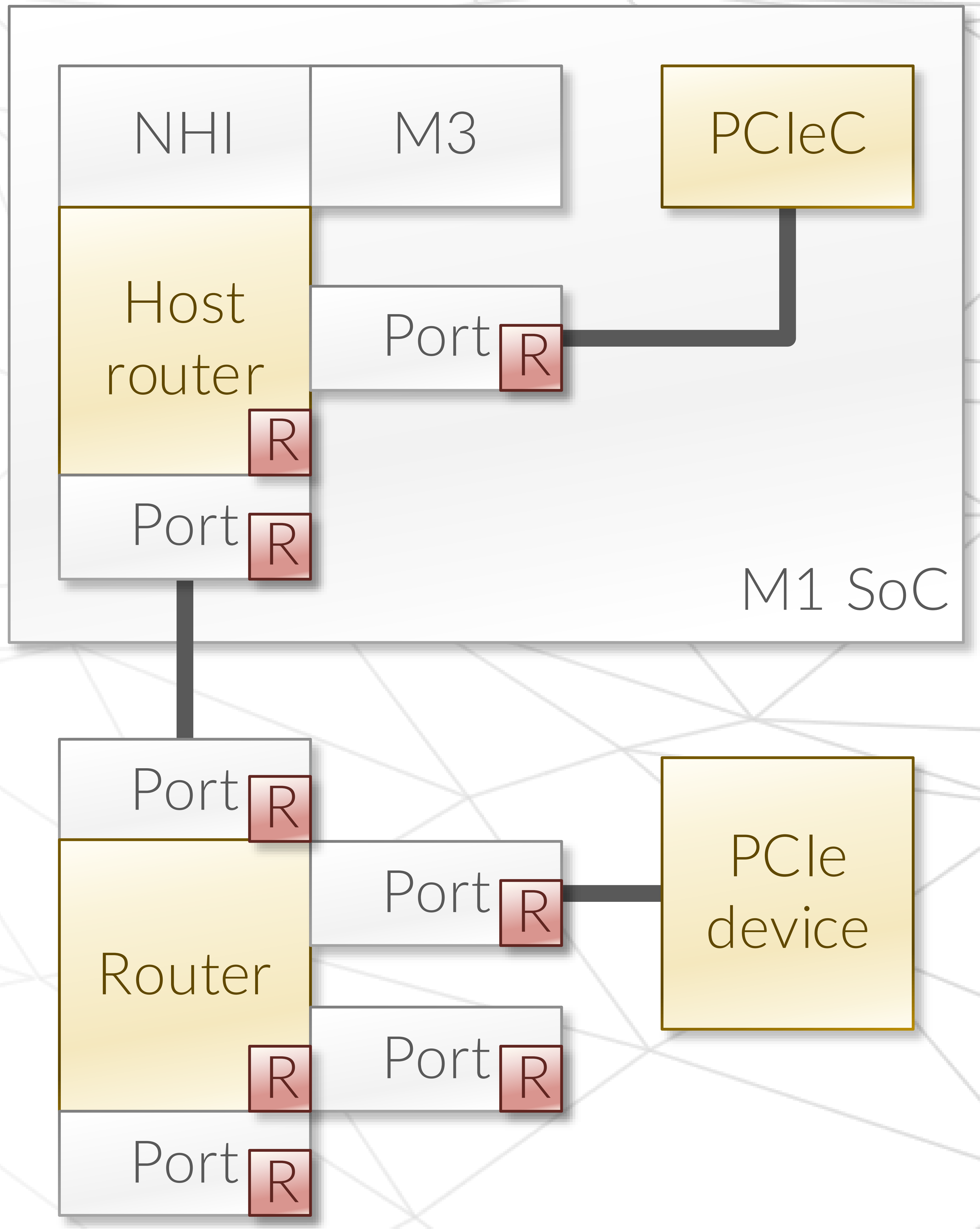
USB 4

NHI

- Ring-based interface to submit and receive messages
- Most messages are control, but has a data plane
- Separate layer, does not know much about topology, routers, ports etc.

Register spaces

- Routers, ports, links, etc. have register spaces
- Used to configure topology
- Defined in USB 4 spec
- … which left space for vendor extensions
- Everyone uses vendor extensions



USB 4

Incompatible NHI

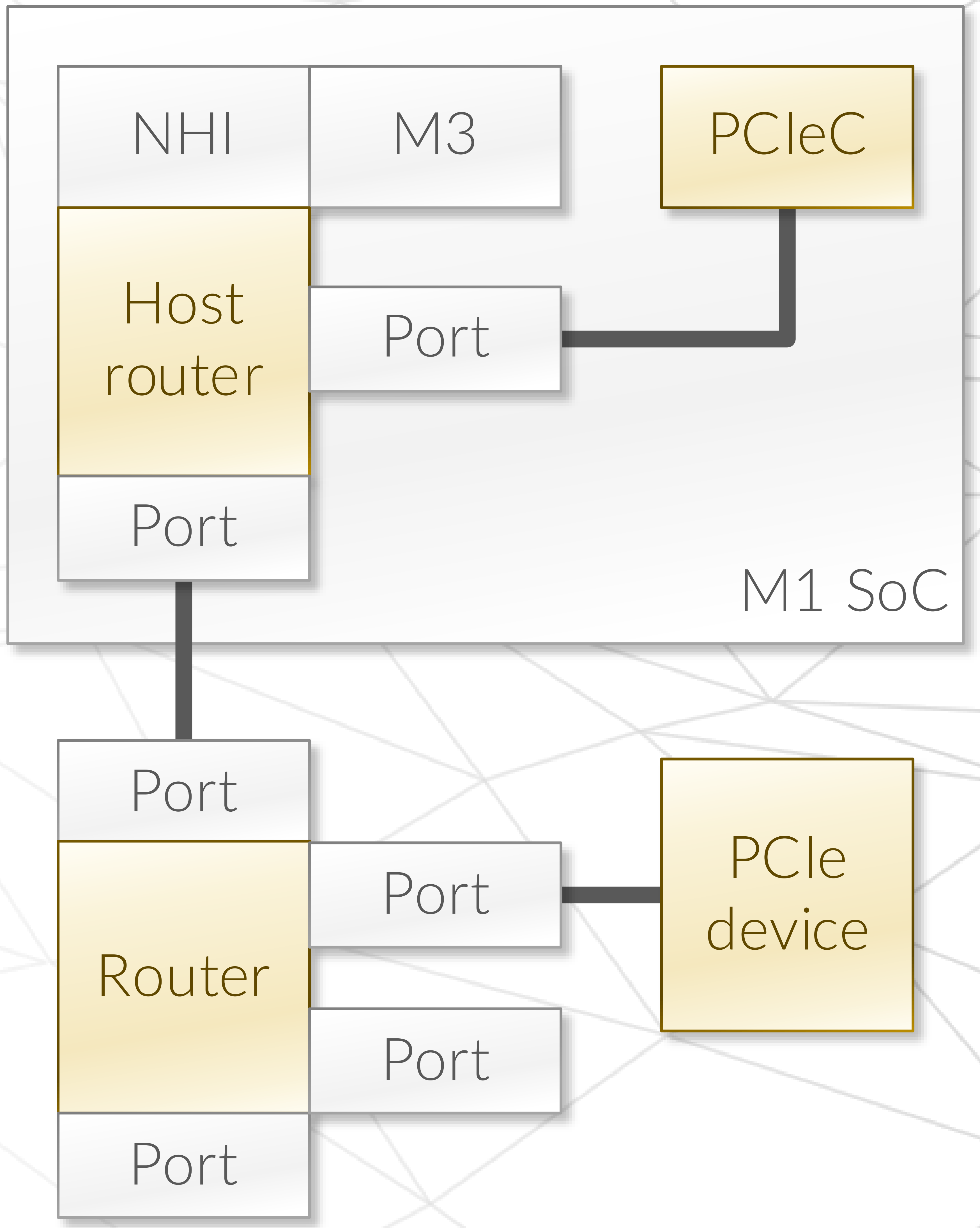
- Not PCI based, unlike USB 4 standard
- Requires booting a Cortex-M3 CPU first
- Have to set up PHY on external port
- Magic router register to wake host router

Special PCIeC controller

- Slightly different from regular APCIe
- A few new registers to trigger connection to USB 4

Type-C PHY

- Used for USB 3, USB 4 and DisplayPort
- Universal but hard to set up with tunables



M1 Linux

Native port

- No virtual machines, hypervisors, etc.
- Loads from iBoot using a preloader
- Preloader builds a Linux device tree with tunables

Peripheral exploration

- Keyboard and touchpad similar to older MacBooks
- Touchbar has a dedicated display controller
- Exact DCP (display coprocessor) protocol changes version to version; *very annoying*
- Bluetooth has a new PCIe ring-based protocol
- USB 4 is pretty cool, but PCIeC limitations seem to preclude GPUs ☹
- Still, nice to have 10G fiber to the Mini

M1 Linux does not have a cool logo or name

THANK YOU

Black Hat 2021