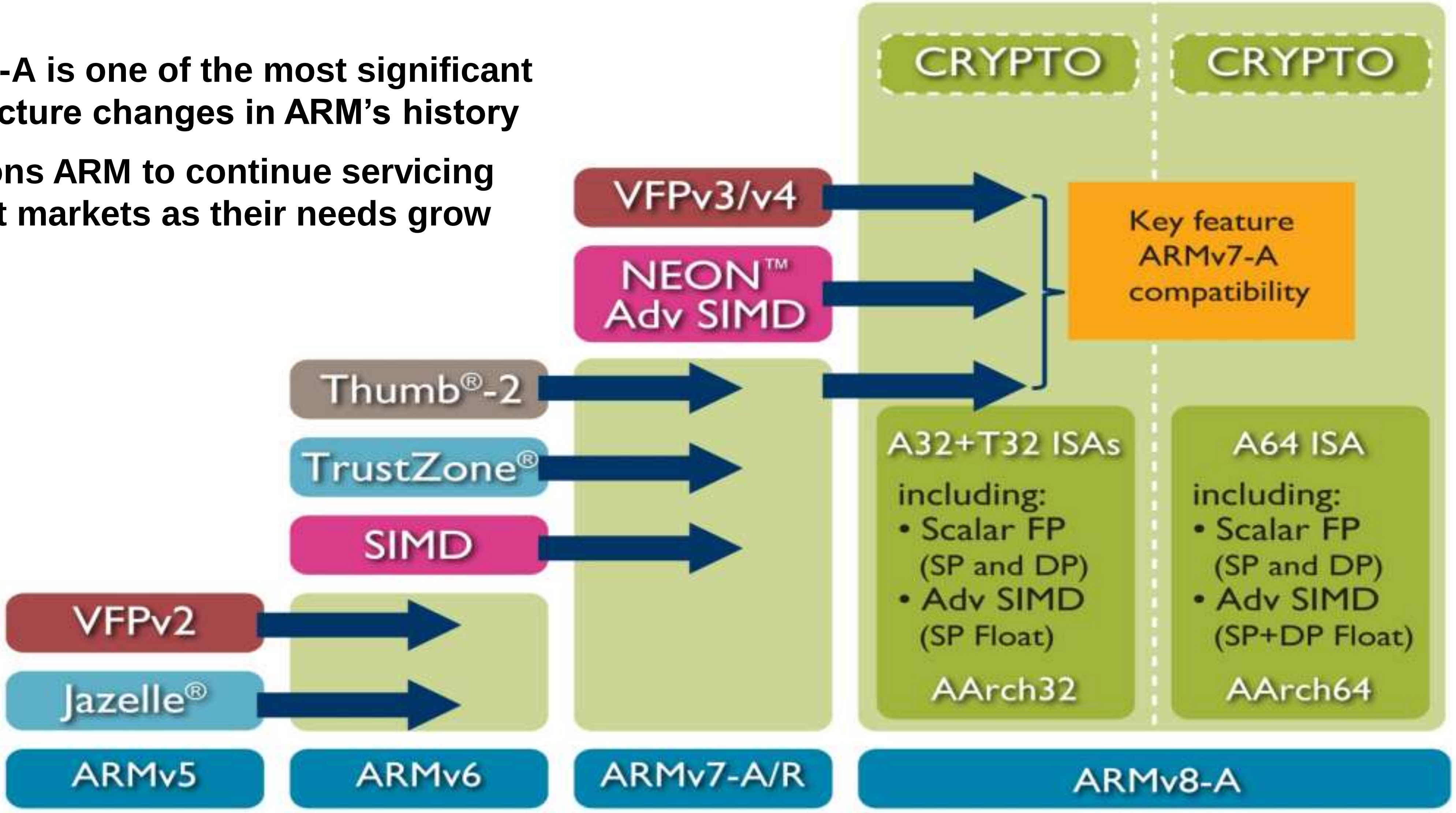
ARMv8-A Architecture Overview

Chris Shore – ARM Training Manager

- **With ARM for 16 years**
- **Managing customer training for 15 years**
  - Worldwide customer training delivery
  - Approved Training Centers
  - Active Assist onsite project services
- **Background**
  - MA Physics & Computer Science, Cambridge University, 1986
  - Background as an embedded software consultant for 17 years
  - Software Engineer
  - Project Manager
  - Technical Director
  - Engineering Manager
  - Training Manager
- **Regular conference speaker and trainer**

Development of the ARM Architecture

- **ARMv8-A is one of the most significant architecture changes in ARM's history**
- **Positions ARM to continue servicing current markets as their needs grow**


CRYPTO
CRYPTO
VFPv3/v4
NEON™ Adv SIMD
Key feature ARMv7-A compatibility
Thumb®-2
TrustZone®
SIMD
A32+T32 ISAs including: • Scalar FP (SP and DP) • Adv SIMD (SP Float)
A64 ISA including: • Scalar FP (SP and DP) • Adv SIMD (SP+DP Float)
AArch32
AArch64
VFPv2
Jazelle®
ARMv5
ARMv6
ARMv7-A/R
ARMv8-A

What's new in ARMv8-A?

- **ARMv8-A introduces two execution states: AArch32 and AArch64**

- **AArch32**
  - Evolution of ARMv7-A
  - A32 (ARM) and T32 (Thumb) instruction sets
    - ARMv8-A adds some new instructions
  - Traditional ARM exception model
  - Virtual addresses stored in 32-bit registers

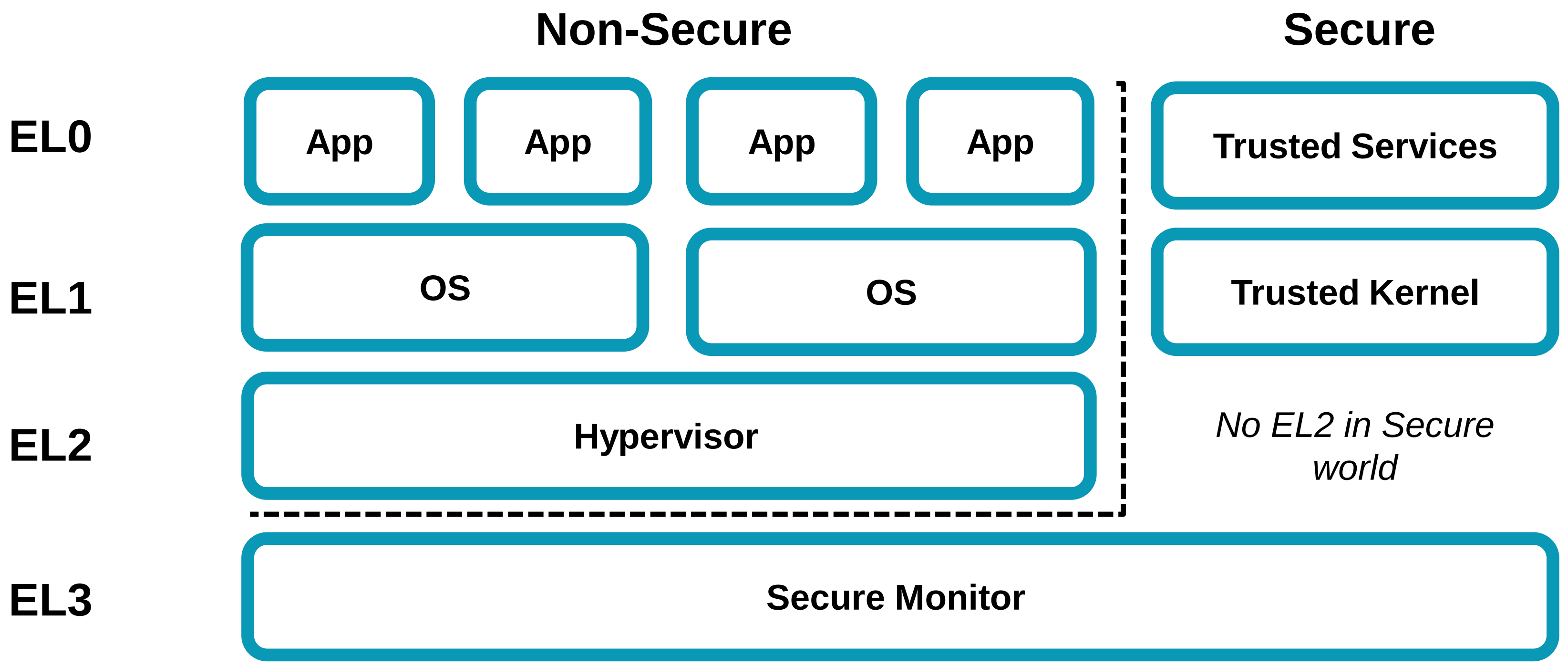
- **AArch64**
  - New 64-bit general purpose registers (X0 to X30)
  - New instructions – A64, fixed length 32-bit instruction set
    - Includes SIMD, floating point and crypto instructions
  - New exception model
  - Virtual addresses now stored in 64-bit registers

Agenda

- Architecture versions
- **Privilege levels**
- AArch64 Registers
- A64 Instruction Set
- AArch64 Exception Model
- AArch64 Memory Model

AArch64 privilege model

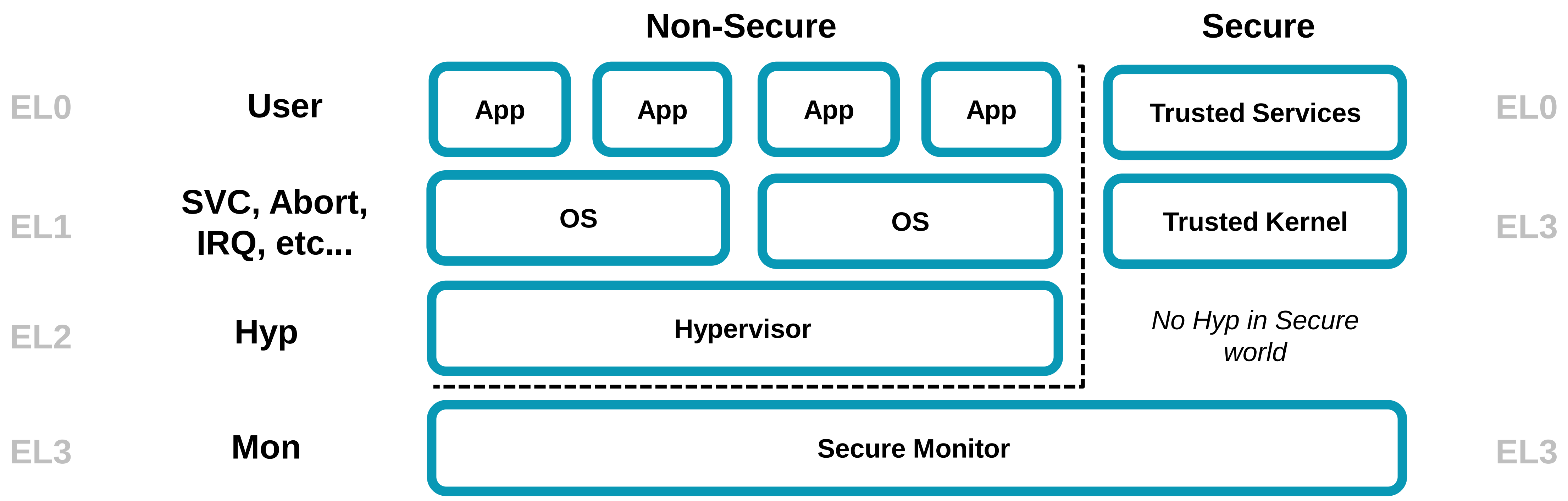
- **AArch64 has four exception levels, and two security states**
  - EL0 = least privileged, EL3 = most privileged
  - Secure state and non-secure (or Normal) state



- **EL2 and EL3 are optional**
  - A processor may not implement EL2/3 if Security or Virtualization are not required

AArch32 privilege model

- **The privilege model in AArch32 is similar to ARMv7-A:**

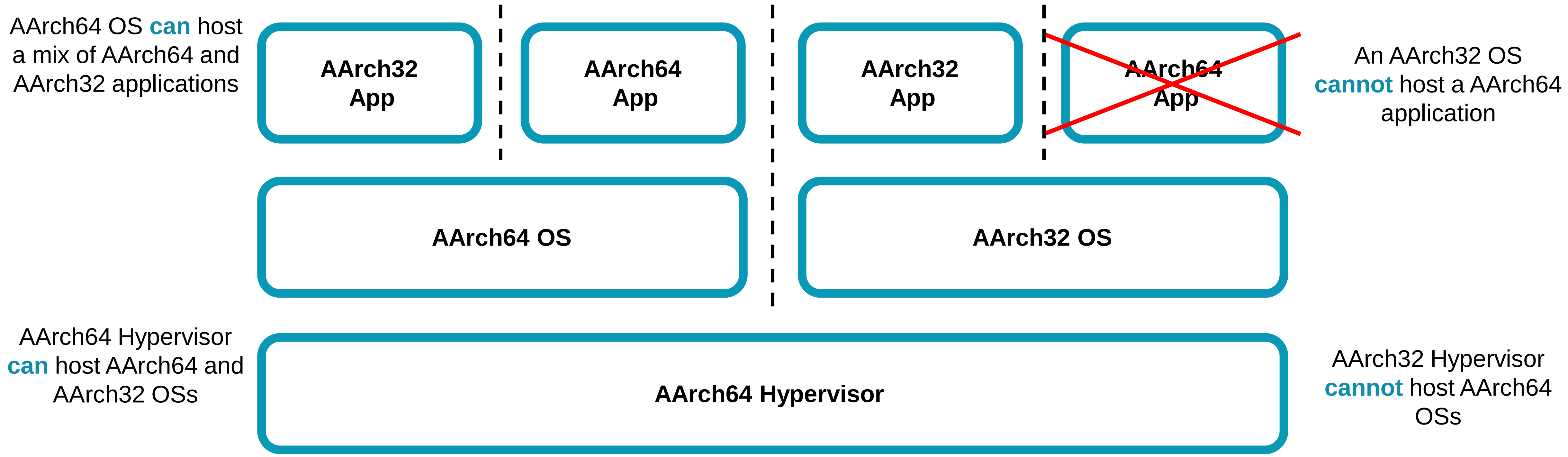


- **When EL3 is using AArch32, in the Secure world the EL1 modes are treated as EL3**
  - No effect on the Normal world

ARM®

Moving between AArch32 & AArch64

- **Execution state can only change on exception entry or return**
  - Moving to a lower EL, execution state can stay the same *or switch to AArch32*
  - Moving to a higher EL, execution state can stay the same *or switch to AArch64*

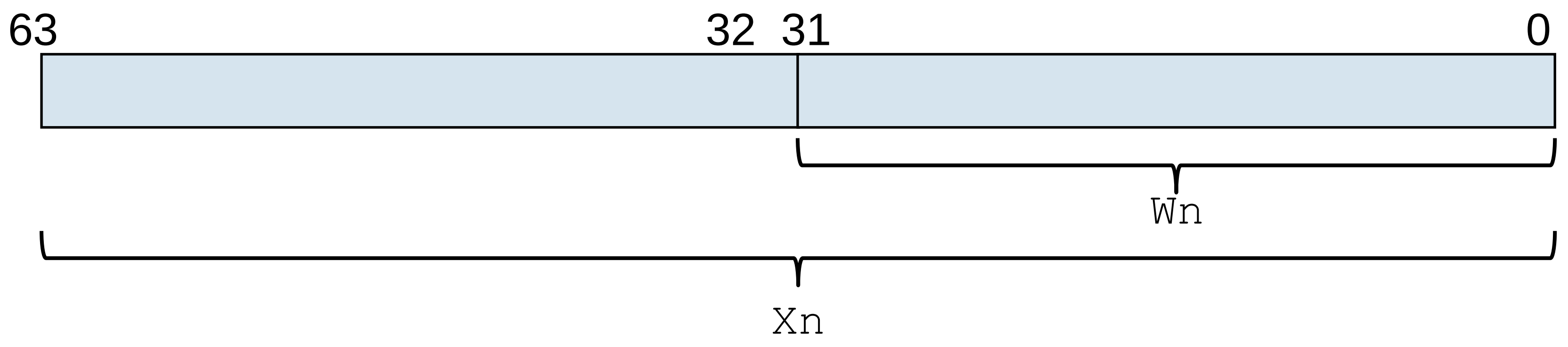

AArch64 OS **can** host a mix of AArch64 and AArch32 applications
AArch32 App
AArch64 App
AArch32 App
AArch64 App
An AArch32 OS **cannot** host a AArch64 application
AArch64 OS
AArch32 OS
AArch64 Hypervisor **can** host AArch64 and AArch32 OSs
AArch64 Hypervisor
AArch32 Hypervisor **cannot** host AArch64 OSs

Agenda

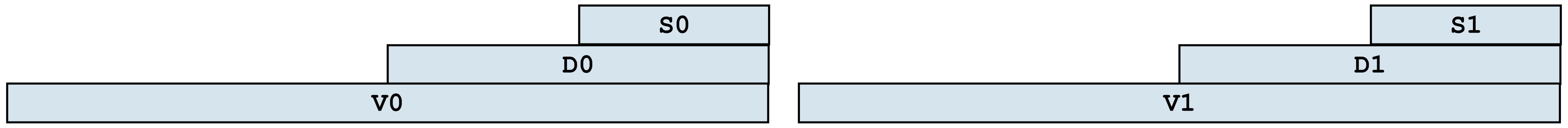
- Architecture versions
- Privilege levels
- **AArch64 Registers**
- A64 Instruction Set
- AArch64 Exception Model
- AArch64 Memory Model

Register banks

- **AArch64 provides 31 general purpose registers**
  - Each register has a 32-bit (w0-w30) and 64-bit (x0-x30) form



- **Separate register file for floating point, SIMD and crypto operations - `Vn`**
  - 32 registers, each 128-bits
    - Can also be accessed in 32-bit (Sn) or 64-bit (Dn) forms



Other registers

- **AArch64 introduces the "zero" register – `XZR` and `WZR`**
  - Reads as 0, writes are ignored

- **The PC is not a general purpose register, cannot be directly referenced**

- **There are separate link registers for function calls and exceptions**
  - **`X30`** – Updated by branch with link instructions (**`BL`** & **`BLR`**)
    Use **`RET`** instruction to return from sub-routines
  - **`ELR_ELn`** – Updated on exception entry
    Use **`ERET`** instruction to return from exceptions

- **Each exception level has its own stack pointer**
  - **`SP_EL0`**, **`SP_EL1`**, **`SP_EL2`** and **`SP_EL3`**
  - The SPs are not general purpose registers
  - Stack pointers must always be 128-bit aligned (bits 3:0 = b0000)
    - Hardware checking of SP alignment can be enabled

Processor state

- **AArch64 does not have a direct equivalent of the AArch32 CPSR**
  - Setting previously held in the CPSR are referred to as Processor State (or PSTATE) fields, and can be accessed individually

Fields	Description
N, Z, C and V	ALU flags
Q	Sticky overflow (AArch32 only)
DAIF	Exception mask bits
SPSel	SP selection (EL0 or ELn), not applicable to EL0
CurrentEL	The current exception level
E	Data endianness (AArch32 only)
IL	Illegal flag. When set, all instructions treated as UNDEFINED
SS	Software stepping bit

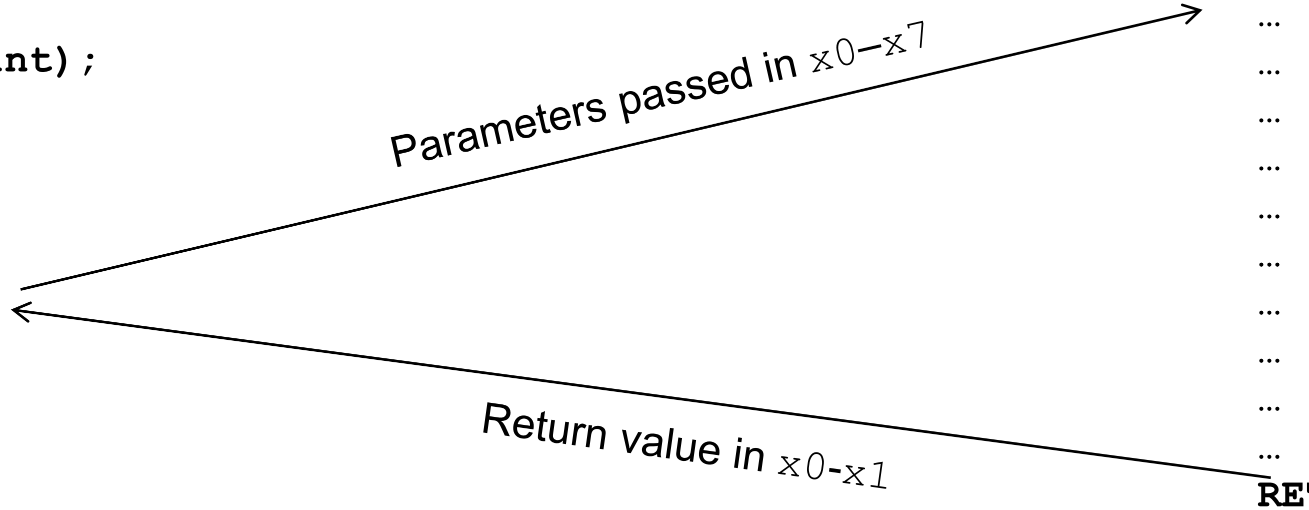
- **AArch64 does include SPSRs, covered later...**

Procedure call standard (1)

- **There is a set of rules known as a Procedure Call Standard (PCS) that specifies how registers should be used:**

```
extern int foo(int, int);

int main(void)
{
   ...
   a = foo(b, c);
   ...
}
```



```
 EXPORT foo
foo PROC
  …
  …
  …
  …
  …
  …
  …
  …
  …
  …
  RET
  ENDP
```

Must preserve
X19 - X29

Can corrupt:
X0 - X18

Return address: **X30 (LR)**

- **Some registers are reserved...**

Procedure call standard (2)

X0–X7	X8–X15	X16–X23	X24–X30
Parameter / result registers (X0–7)	XR (X8)	IP0 (X16)	Callee-saved (X24–28)
	Corruptible Registers (X9–15)	IP1 (X17)	
		PR (X18)	
		Callee-saved (X19–23)	FP (X29)
			LR (X30)

- **IP0 & IP1: Intra-procedure-call temporary registers (corruptible)**
- **XR: Indirect result location parameter (corruptible)**
- **PR: Platform registers. Reserved for the use of platform ABIs**
- **FP: Frame pointer**

Procedure call standard (3)

- **The PCS also covers the use of the floating point/SIMD registers:**

`D0-D7`	`D8-D15`	`D16-D23`	`D24-D31`
Parameter / result registers	Callee saved registers	Corruptible registers	Corruptible registers

AArch64 ↔ AArch32 register mappings

X0-X7	X8-X15	X16-X23	X24-X30
R0	R8_usr	R14_irq	R8_fiq
R1	R9_usr	R13_irq	R9_fiq
R2	R10_usr	R14_svc	R10_fiq
R3	R11_usr	R13_svc	R11_fiq
R4	R12_usr	R14_abt	R12_fiq
R5	R13_usr	R13_abt	R13_fiq
R6	R14_usr	R14_und	R14_fiq
R7	R13_hyp	R13_und	

- **When moving from AArch32 to AArch64**
  - Registers accessible in both registers widths
    - Top 32 bits: UNKNOWN
    - Bottom 32 bits: The value of the AArch32 register
  - Registers that are not accessible in AArch32 retain value from previous AArch64 execution

System control

- **In AArch64 system configuration is controlled through system registers**
  - Register names tell you the lowest exception levels they can be accessed from
  - For example:

`TTBR0_EL1`	– can be accessed from EL1, EL2 and EL3
`TTBR0_EL2`	– can be accessed from EL2 and EL3

- **Accessed using `MSR` and `MRS` instructions**

```
MRS   x0, TTBR0_EL1                  ; Move TTBR0_EL1 into x0
MSR   TTBR0_EL1, x0                  ; Move x0 into TTBR0_EL1
```

Some important System Registers

- **`SCTLR_ELn` (System Control Register)**
  - Controls architectural features, for example MMU, caches and alignment checking
- **`ACTLR_ELn` (Auxiliary Control Register)**
  - Controls processor specific features
- **`SCR_EL3` (Secure Configuration Register)**
  - Controls secure state and trapping of exceptions to EL3
- **`HCR_EL2` (Hypervisor Configuration Register)**
  - Controls virtualization settings, and trapping of exceptions to EL2
- **`MIDR_EL1` (Main ID Register)**
  - The type of processor the code is running on (e.g. part number and revision)
- **`MPIDR_EL1` (Multiprocessor Affinity Register)**
  - The core and cluster IDs, in multi-core/cluster systems
- **`CTR_EL0` (Cache Type register)**
  - Information about the integrated caches (e.g. the size)

Agenda

Architecture versions

Privilege levels

AArch64 Registers

- **A64 Instruction Set**

AArch64 Exception Model

AArch64 Memory Model

A64 overview

- **AArch64 introduces new A64 instruction set**
  - Similar set of functionality as traditional A32 (ARM) and T32 (Thumb) ISAs
- **Fixed length 32-bit instructions**
- **Syntax similar to A32 and T32**

```
ADD W0, W1, W2          ← w0 = w1 + w2   (32-bit addition)
ADD X0, X1, X2          ← x0 = x1 + x2   (64-bit addition)
```

- **Most instructions are not conditional**
- **Optional floating point and Advanced SIMD instructions**
- **Optional cryptographic extensions**

Agenda

Architecture versions

Privilege levels

AArch64 Registers

A64 Instruction Set

- **AArch64 Exception Model**

AArch64 Memory Model

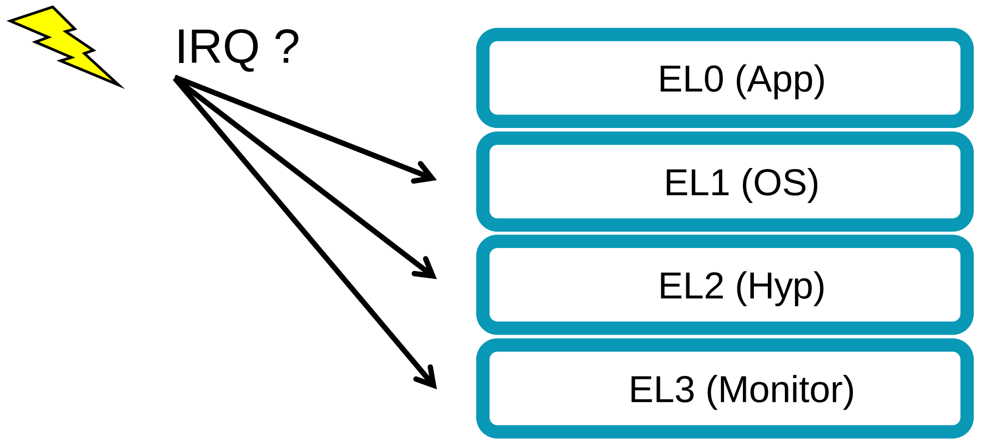
AArch64 exceptions

- **In AArch64 exceptions are split between:**
  - Synchronous
    - Data aborts from MMU, permission/alignment failures, service call instructions, etc.
  - Asynchronous
    - IRQ/FIQ
    - SError (System Error)

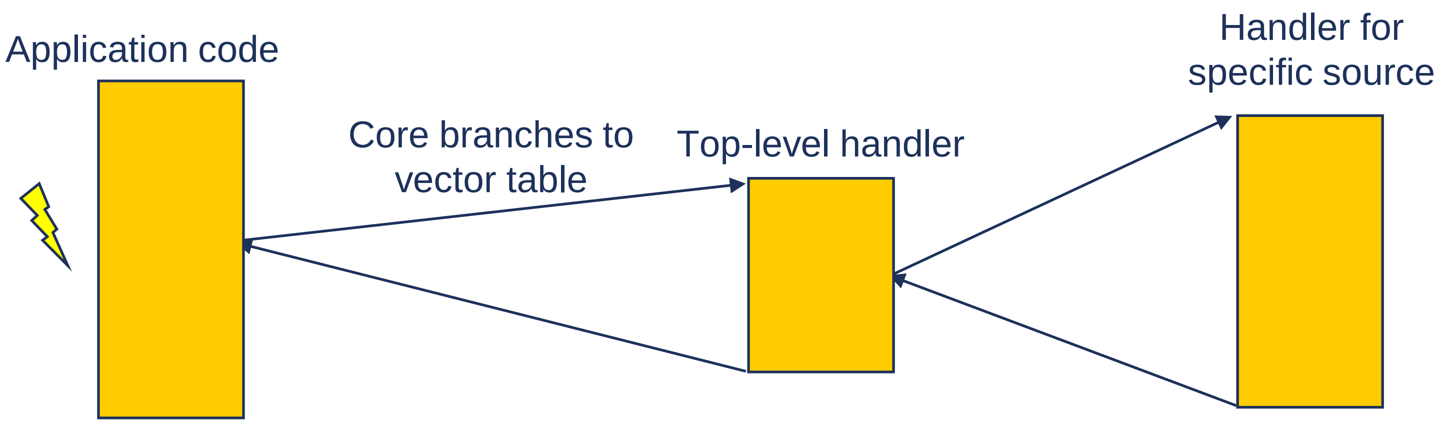
- **On taking an exception the EL can stay the same OR get higher**
  - Exceptions are never taken to EL0

- **Synchronous exceptions are normally taken in the current EL**

- **Asynchronous exceptions can be routed to a higher EL**
  - **`SCR_EL3`** specifies exceptions to be routed to EL3
  - **`HCR_EL2`** specifies exceptions to be routed to EL2
  - Separate bits to control routing of IRQs, FIQs and SErrors



Taking an exception



- **When an exception occurs:**
  - **`SPSR_ELn`** updated
  - **`PSTATE`** updated
    - EL stays the same OR gets higher
  - Return address stored to **`ELR_ELn`**
  - PC set to vector address
  - If synchronous or SError exception, **`ESR_ELn`** updated with cause of exception

- **To return from an exception execute ERET instruction, this:**
  - Restores **`PSTATE`** from **`SPSR_ELn`**
  - Restores PC from **`ELR_ELn`**

Agenda

- Architecture versions
- Privilege levels
- AArch64 Registers
- A64 Instruction Set
- AArch64 Exception Model
- **AArch64 Memory Model**

Memory types

- **Address locations must be described in terms of a *type***
  - The “type” tells the processor how it can access that location
    - Access ordering rules
    - Speculation

- **Normal**
  - Used for code and data
  - Processor allowed to re-order, re-size and repeat accesses
  - Speculative accesses allowed

- **Device**
  - Used for peripherals
  - Accesses could have side effects, so there are more restrictions on what optimizations a processor can perform
  - Speculative data accesses not allowed

- **Other attributes can also be specified**
  - For example whether a region is executable, shareable and cacheable

Alignment

- **Unaligned data accesses are allowed to address ranges marked as Normal**

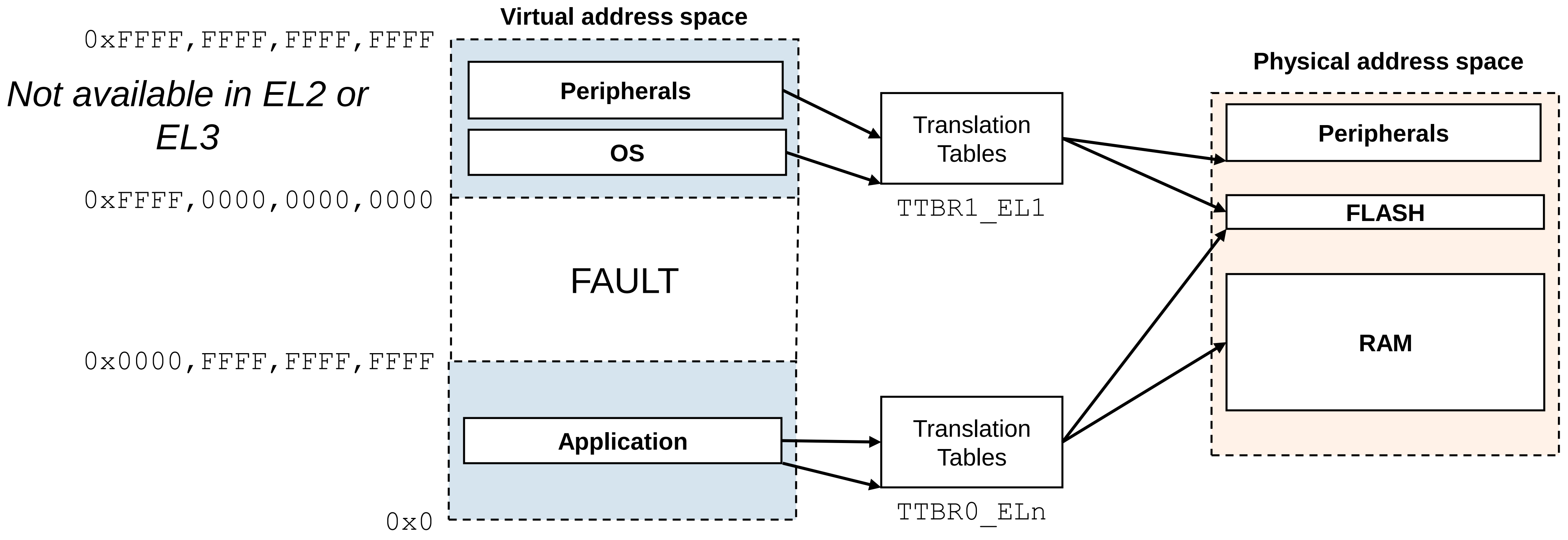
- **Optionally, all unaligned data accesses can trapped**
  - Trapped unaligned accesses cause a synchronous data abort
  - Trapping can be enabled independently separately for EL0/EL1, EL2 and EL3
    - Controlled by **`SCTLR_ELn.A`** bits

- **Unaligned data accesses to addresses marked as Device will always trigger an exception**
  - Synchronous data abort

- **Instruction fetches must always be aligned**
  - A64 instructions must be 4-byte aligned (bits 1:0 = b00)
  - Synchronous exception

Virtual address space

- **Virtual addresses are 64-bit wide, but not all addresses are accessible**
  - Virtual memory address space split between two translation tables
    - Each covering a configurable size, up to 48 bits of address space (**`TCR_ELn`**)
  - Addresses not covered by either translation table generate translation faults



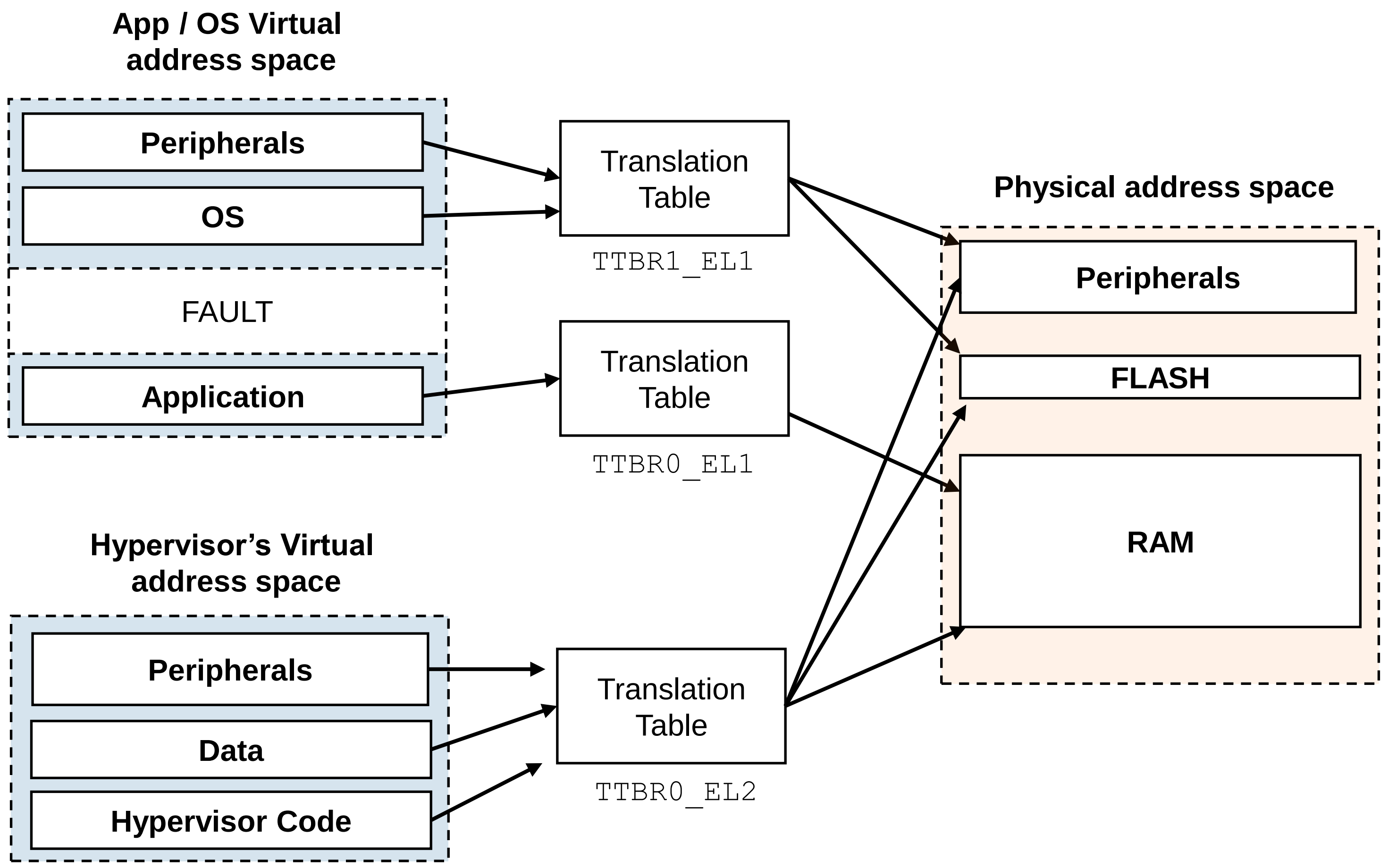
Multiple virtual address spaces

- **A system may define multiple virtual address spaces:**

- **OS and applications**
  - **`TTBR0_EL1`**
  - **`TTBR1_EL1`**
  - **`TCR_EL1`**

- **Hypervisor**
  - **`TTBR0_EL2`**
  - **`TCR_EL2`**

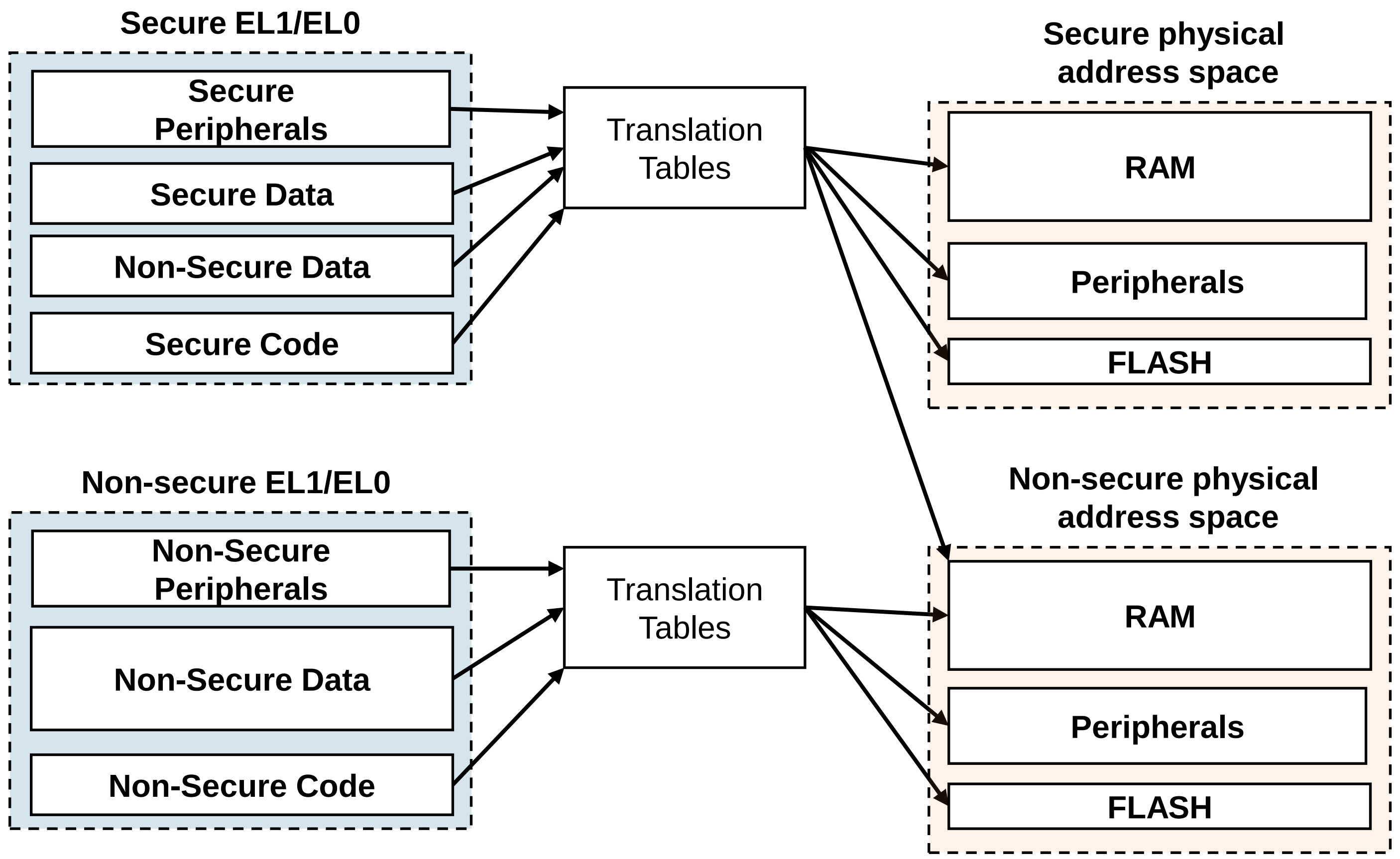
- **Secure Monitor**
  - **`TTBR0_EL3`**
  - **`TCR_EL3`**



Physical address spaces

- **ARMv8-A defines two security states: Secure and Non-secure (Normal)**
  - It also defines two physical address spaces: Secure and Non-secure

- **These are in theory completely separate:**
  - SP:0x8000 != NP:0x8000
  - But most systems treat Secure/Non-Secure as an attribute for access control
- **Normal world can only access the non-secure physical address space**
- **Secure world can access BOTH physical address spaces**
  - Controlled through translation tables



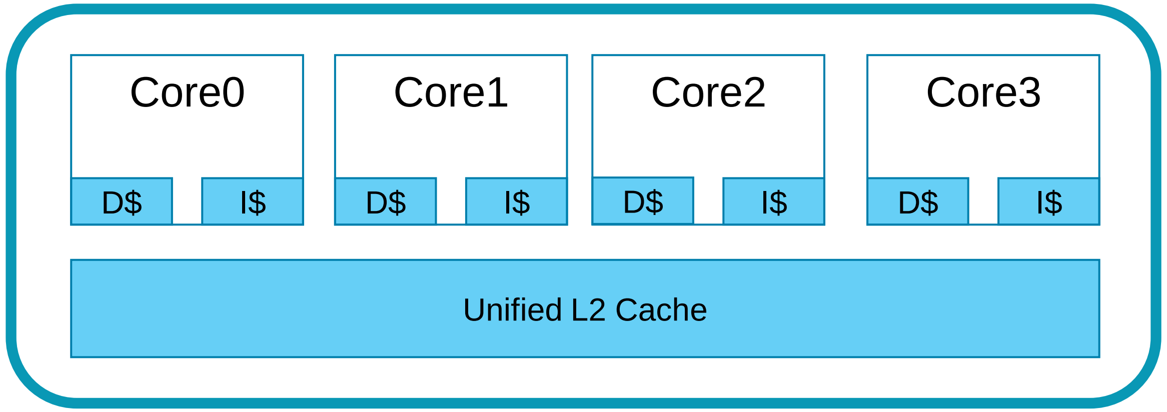
MPCore configurations

- **Many implementations of ARM processors have a multi-core configuration**
  - Multiple cores contained within the same block

- **Each core has its own MMU configuration , register bank, internal state and Program Counter**
  - Core0 might be executing in Non-secure, AArch32 EL0 while Core1 is executing in Secure, AArch64 EL1

- **Cores can be powered and brought in and out of reset independently**
  - ID registers allow discovery of core affinity

- **Each core has separate L1 data and instruction caches**
  - Hardware will maintain coherency between L1 data caches for certain memory types
  - Some cache and TLB instructions are broadcast to other cores
  - All cores share a common physical memory map



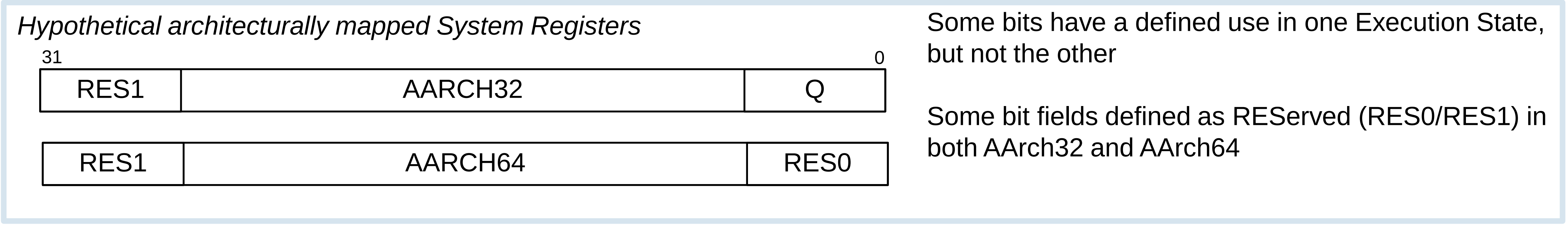
Appendix

ARMv8 terminology reference

- **EL3, EL2, EL1 and EL0 are Exception Levels**
  - The EL denotes the level of privilege
- **AArch32 and AArch64 are Execution States**
  - The programmer’s model being used
- **Secure and Non-Secure are Security States**
  - EL3 is always Secure, EL2 is always Non-Secure
  - EL0/1 can be Secure or Non-Secure (sometime S.ELn or NS.ELn are used as shorthand)
- **A64, A32 and T32 are Instruction Sets**
  - A64 used when in AArch64
  - A32 and T32 used when in AArch32
    - In previous architecture versions A32 was called ARM, and T32 was called Thumb
- **Examples:**
  - Processor currently executing in EL3 as AArch64, executing A64 instructions

REServed bits and the v8 Architecture

- **System registers often include REServed bit fields**
  - Indicating fields that are not used by hardware



Some bits have a defined use in one Execution State, but not the other

Some bit fields defined as REServed (RES0/RES1) in both AArch32 and AArch64

- **Where a bit can be RES at one Execution State and used in another**
  - The Architecture defines the bit field as writeable or “stateful”
    - Allows the correct value to be written for a context switch

- **Where bits are unused in both Execution States**
  - Typically an implementation would make these fields write-ignore
  - However the Architecture does permit writing to such fields
    - Changing the value will have no functional impact in current implementations
    - Future Architecture revisions may use these fields
      - RES0/RES1 indicate expected values SW should write to guarantee current behaviour

ARM®

System Register contents at reset

- **In AArch64 most System Register fields are defined as UNKNOWN at reset**
  - If an implementation defines unused RES bits as stateful
    - These fields are also defined as UNKNOWN at reset
  - Recommended that software always writes the full register field when initializing (rather than read/modify write)
    - Including any RES0/RES1 bits as b0/b1

- **The Execution State of the highest EL (entered on reset) defines the reset contents of System Registers**
  - If the highest EL uses AArch64, but lower ELs use AArch32
  - You may need to initialize ARMv7/AArch32 System Registers with expected ARMv7/AArch32 reset values in software before changing EL

ARMv8.1-A

- **The ARM architecture continues to evolve, with the announcement of ARMv8.1-A**

- **Instruction set enhancements**
  - Atomic read-write instructions added to A64
    - For example: Compare and swap
  - Additional SIMD instructions
    - Example use case is colour space conversion
  - Load and stores with ordering limited to a configurable region

- **Virtualization Host Extensions**
  - To improve performance of Type 2 Hypervisors

- **And other enhancements to the memory system architecture, such as Privileged Access Never (PAN) state bit**

ARMv8-A software support

- **ARMv8 software support now widely available in the open source community**

- **Linux Kernel**
  - AArch64 support has been available in mainline for several releases
  - Under arch/arm64/

- **Filesystems**
  - AArch64 kernel supports both legacy ARMv7-A and AAarch64 filesystem components
  - Some guidance on building file-systems for AArch64 is available here https://wiki.linaro.org/HowTo/ARMv8/OpenEmbedded
  - Both Fedora and openSUSE have AArch64 releases

- **ARM Foundation Model**
  - Linaro provide an example kernel and file-system, which can be executed on ARM's free virtual software platform (Foundation Model)
  - http://www.arm.com/products/tools/models/fast-models/foundation-model.php
  - http://www.linaro.org/engineering/engineering-projects/armv8

ARMv8-A software support continued

- **Open Source Tools Support**
  - Linaro provide prebuilt AArch64 GCC toolchain binaries (GCC, GDB, etc.) with Linux and bare-metal library options
  - These are available as cross or native toolchains
  - GCC 4.8 includes –mcpu tuning support for Cortex-A53
  - https://launchpad.net/linaro-toolchain-binaries/

- **ARM tools**
  - The ARM compiler supports AArch64 and is suitable for bare-metal/validation environments
  - DS-5 includes debug support for ARMv8 hardware and models http://www.arm.com/products/tools/software-tools/ds-5/index.php
  - Fast Models allows the creation of Cortex-A5x based ARM Virtual Platforms for software development

ARM Technical Training

- **Available**
  - Covering all ARM cores and GPUs
  - Hardware and Software options
  - Customizable agendas from 2 hours (Live Remote only) to 5 days
- **Accessible**
  - Private, onsite delivery for corporate customers
  - Public course schedule for open enrolment
  - Live Remote webex for flexible delivery
- **Affordable**
  - Standard Private course fees include all ARM’s related expenses
  - Public course pricing accessible to individuals
  - Live Remote is cost effective for small teams
- **Learn from the experts**

http://www.arm.com/training

ARMv8-A Architecture Overview