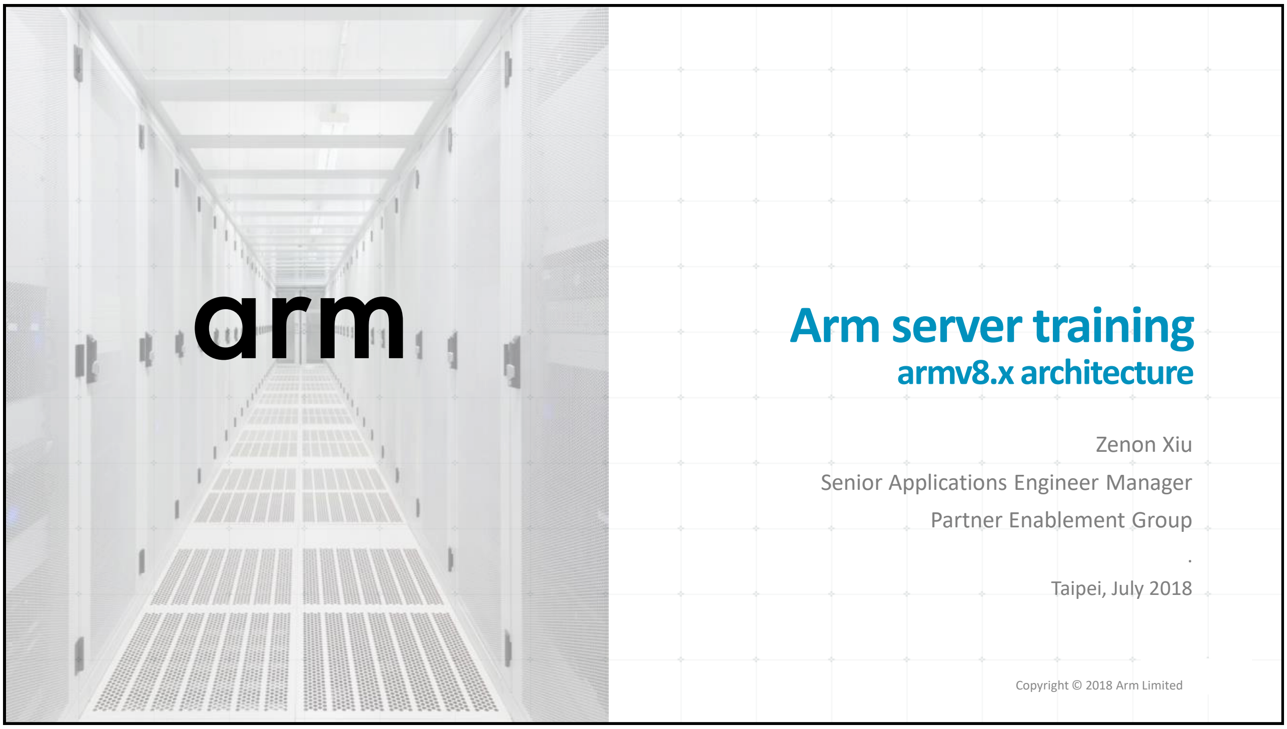


Arm server training

armv8.x architecture

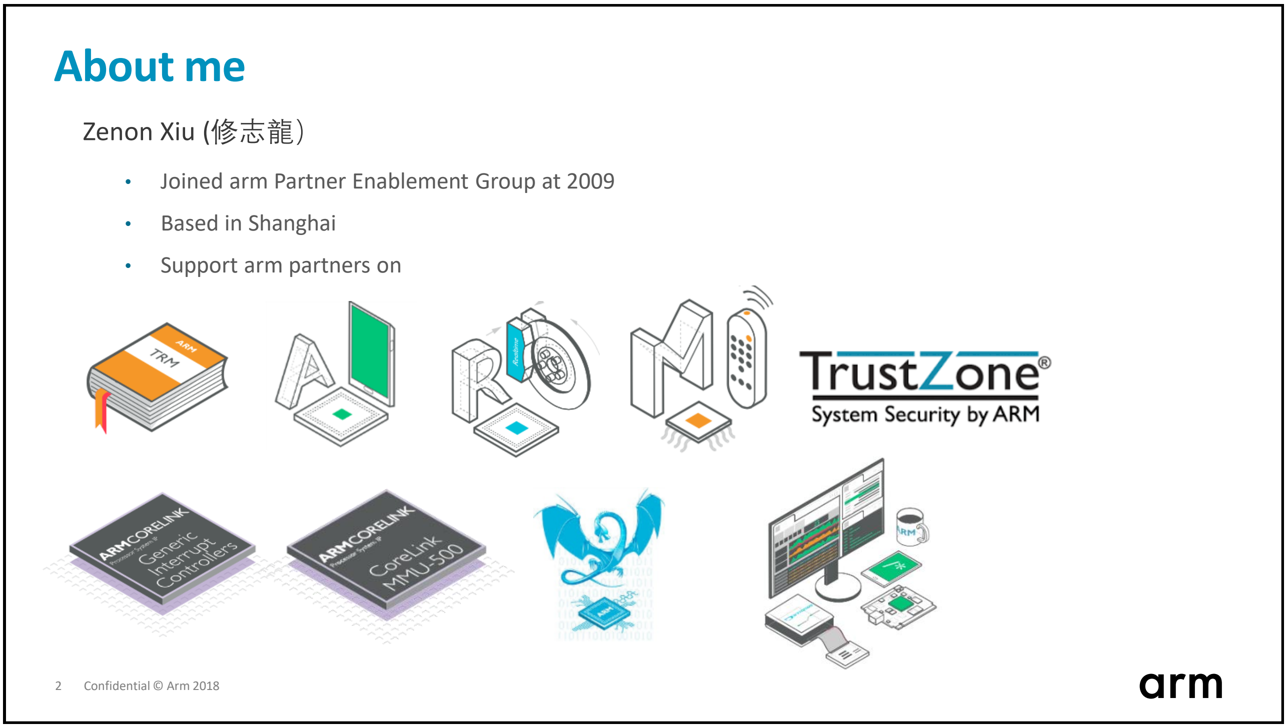
Zenon Xiu

Senior Applications Engineer Manager

Partner Enablement Group

.

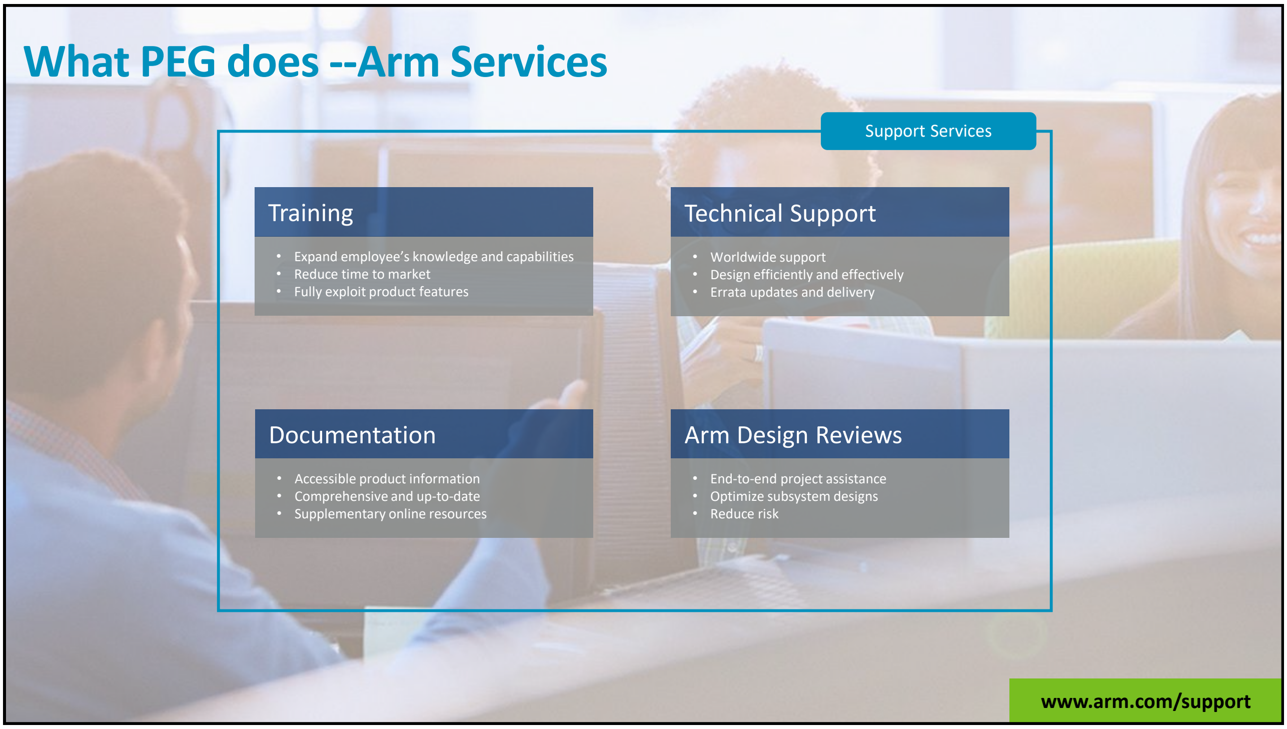
Taipei, July 2018



About me

Zenon Xiu (修志龍)

- Joined arm Partner Enablement Group at 2009
- Based in Shanghai
- Support arm partners on



What PEG does --Arm Services

Support Services

Training

- Expand employee's knowledge and capabilities
- Reduce time to market
- Fully exploit product features

Technical Support

- Worldwide support
- Design efficiently and effectively
- Errata updates and delivery

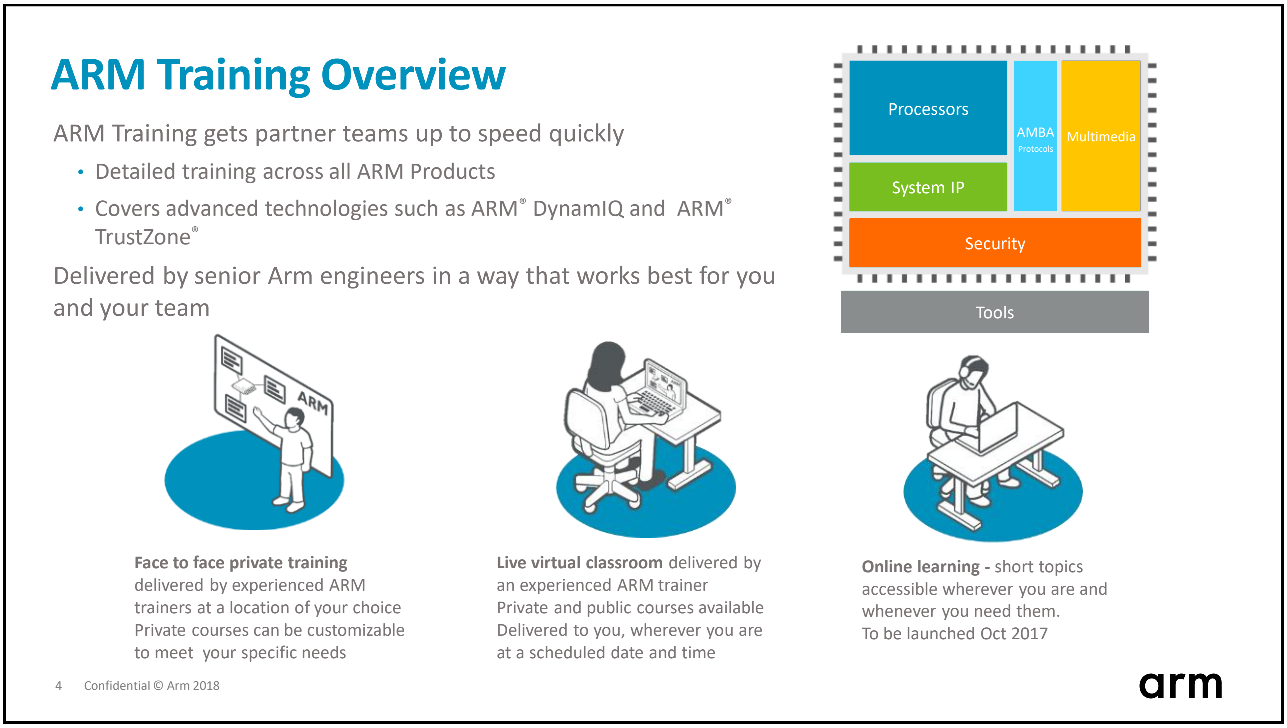
Documentation

- Accessible product information
- Comprehensive and up-to-date
- Supplementary online resources

Arm Design Reviews

- End-to-end project assistance
- Optimize subsystem designs
- Reduce risk

www.arm.com/support



ARM Training Overview

ARM Training gets partner teams up to speed quickly

- Detailed training across all ARM Products
- Covers advanced technologies such as ARM® DynamIQ and ARM® TrustZone®

Delivered by senior Arm engineers in a way that works best for you and your team

Processors

AMBA
Protocols

Multimedia

System IP

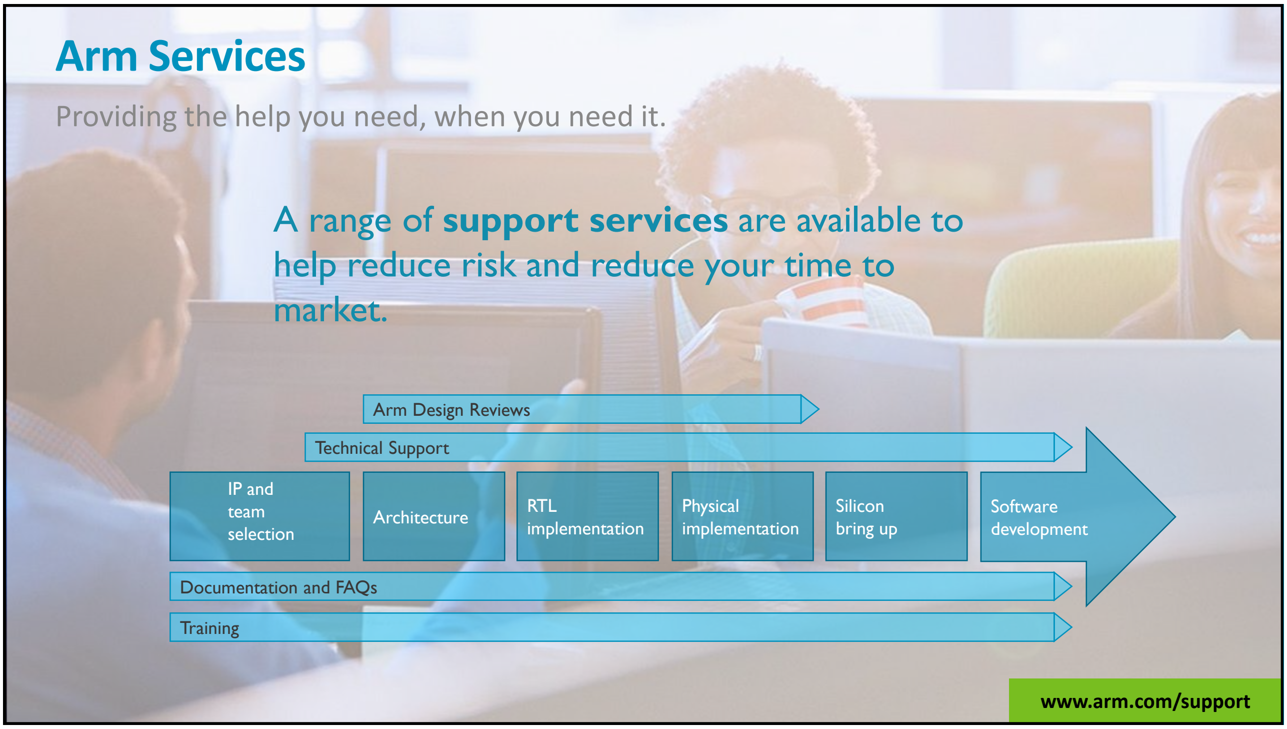
Security

Tools

Face to face private training delivered by experienced ARM trainers at a location of your choice
Private courses can be customizable to meet your specific needs

Live virtual classroom delivered by an experienced ARM trainer
Private and public courses available
Delivered to you, wherever you are at a scheduled date and time

Online learning - short topics accessible wherever you are and whenever you need them.
To be launched Oct 2017



Arm Services

Providing the help you need, when you need it.

A range of **support services** are available to help reduce risk and reduce your time to market.

Arm Design Reviews

Technical Support

IP and team selection

Architecture

RTL implementation

Physical implementation

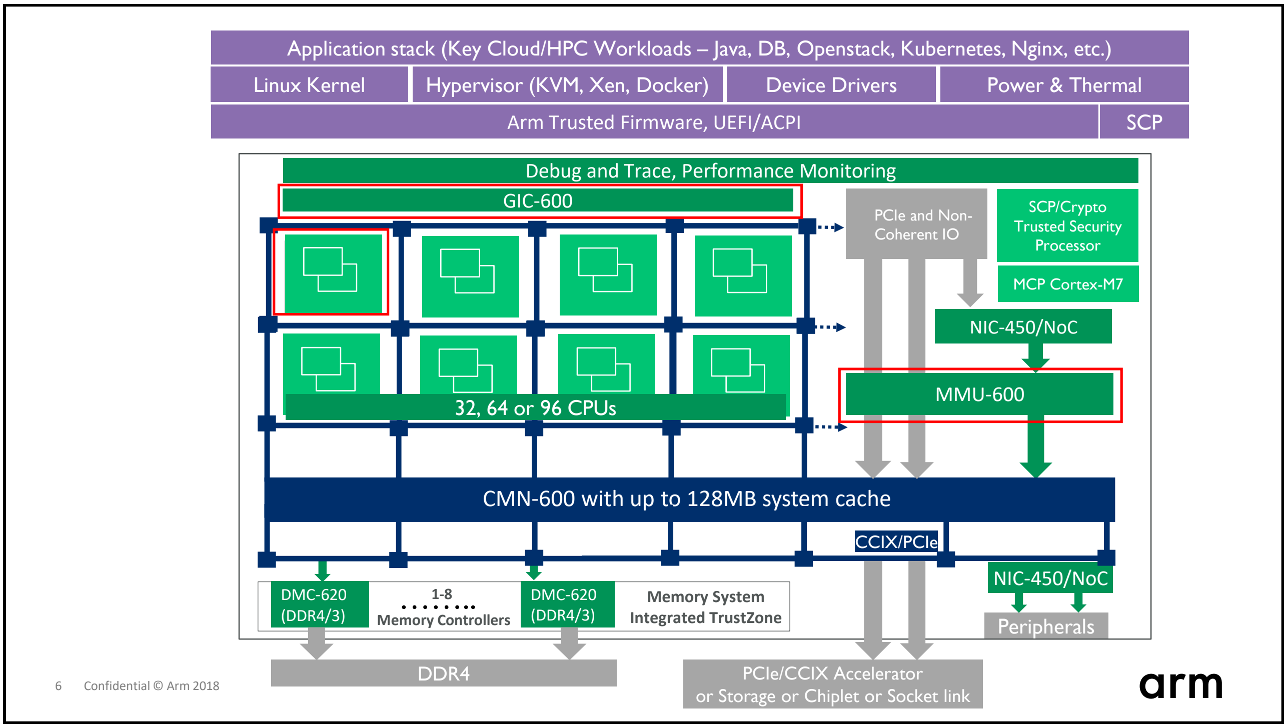
Silicon bring up

Software development

Documentation and FAQs

Training

www.arm.com/support



Application stack (Key Cloud/HPC Workloads – Java, DB, Openstack, Kubernetes, Nginx, etc.)

Linux Kernel

Hypervisor (KVM, Xen, Docker)

Device Drivers

Power & Thermal

Arm Trusted Firmware, UEFI/ACPI

SCP

Debug and Trace, Performance Monitoring

GIC-600

PCIe and Non-Coherent IO

SCP/Crypto Trusted Security Processor

MCP Cortex-M7

NIC-450/NoC

MMU-600

32, 64 or 96 CPUs

CMN-600 with up to 128MB system cache

CCIX/PCIe

NIC-450/NoC

DMC-620 (DDR4/3)

1-8 Memory Controllers

DMC-620 (DDR4/3)

Memory System Integrated TrustZone

Peripherals

DDR4

PCIe/CCIX Accelerator or Storage or Chiplet or Socket link

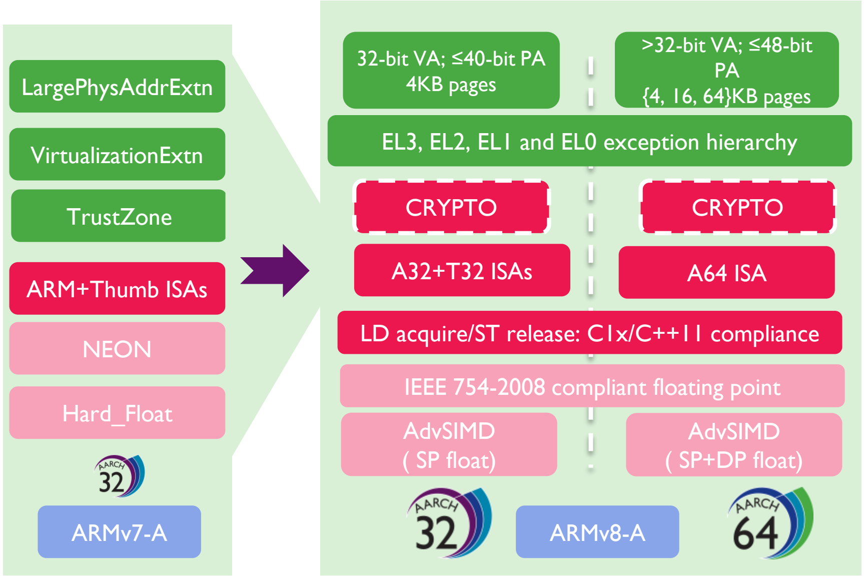
6 Confidential © Arm 2018

arm

Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

Arm Architecture Evolution: Armv8.0-A



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

- EL0-EL3

Virtual addresses now stored in 64-bit registers

- Extend virtual memory space

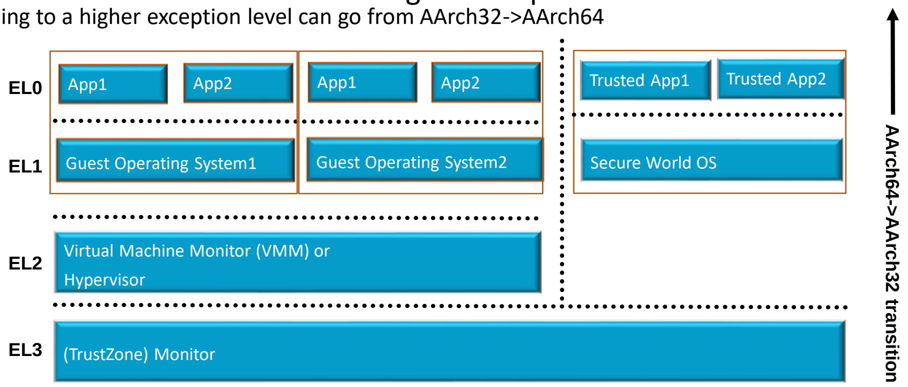
Armv8 Exception Model

4 exception levels: EL3-EL0

- Forms a privilege hierarchy, EL0 the least privileged

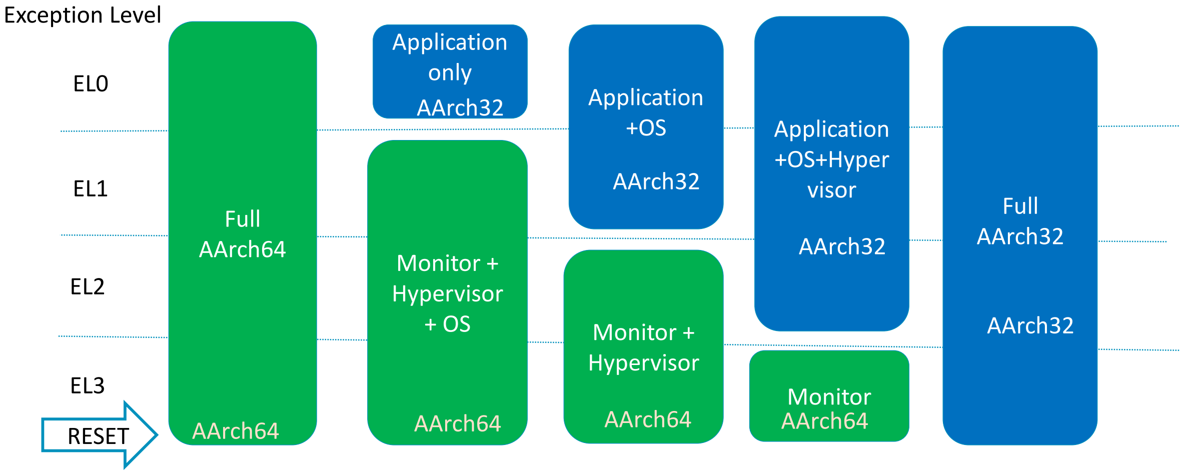
Exceptions can be taken to same or a higher exception level

- Going to a higher exception level can go from AArch32->AArch64

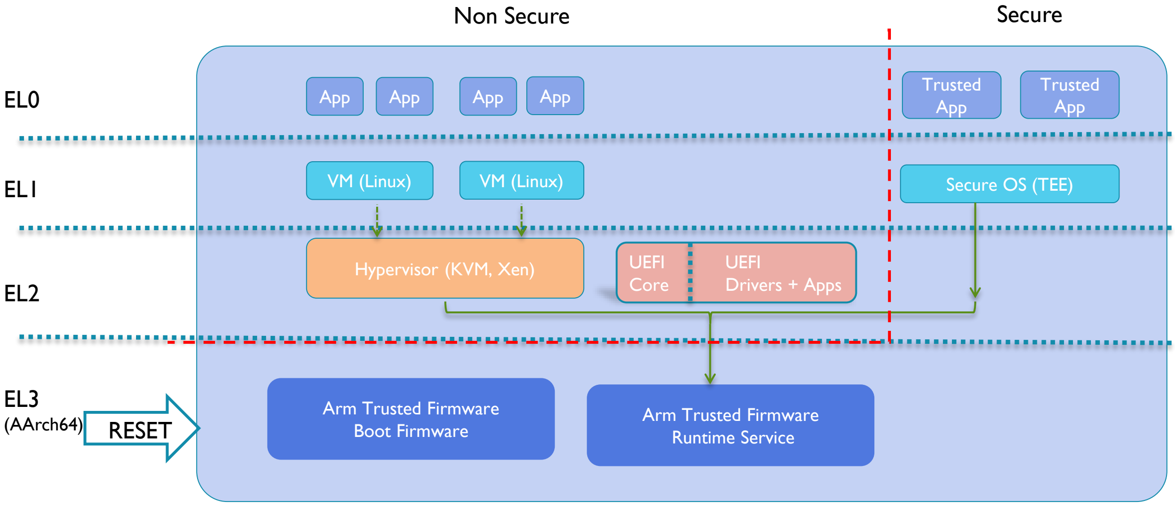


EL0
App1
App2
App1
App2
Trusted App1
Trusted App2
EL1
Guest Operating System1
Guest Operating System2
Secure World OS
EL2
Virtual Machine Monitor (VMM) or Hypervisor
EL3
(TrustZone) Monitor
AArch64->AArch32 transition

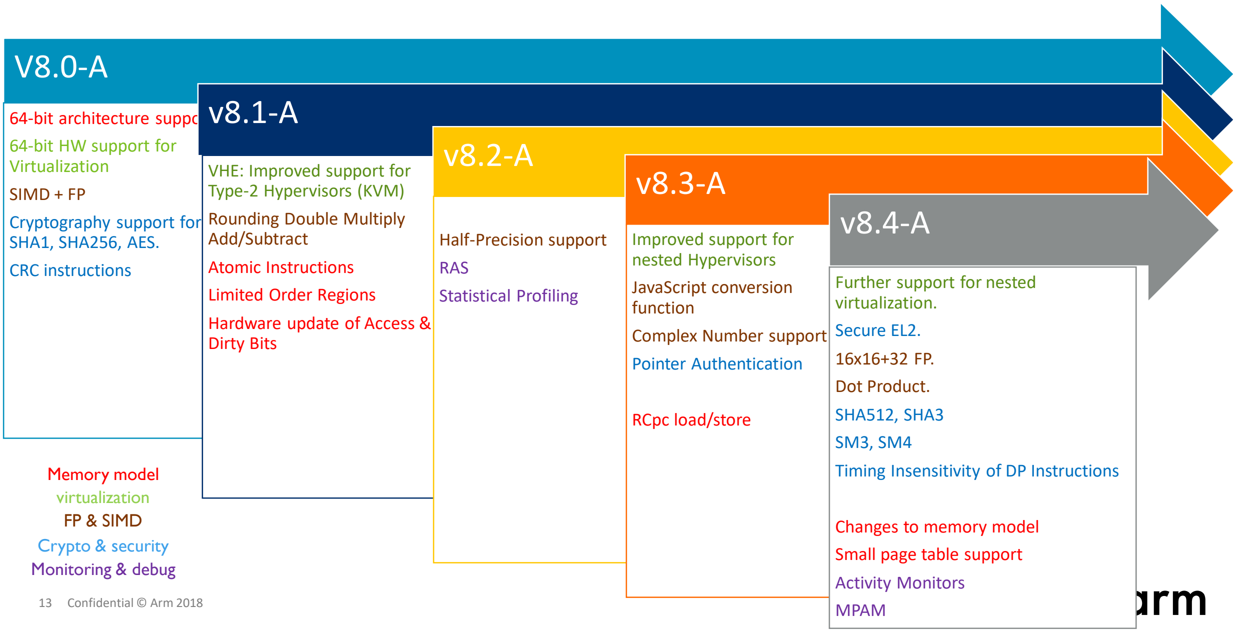
ARMv8 AArch64/AArch32 transaction



Armv8 software stack example



Armv8.x Architecture Features



V8.0-A

64-bit architecture suppo

64-bit HW support for Virtualization

SIMD + FP

Cryptography support for SHA1, SHA256, AES.

CRC instructions

v8.1-A

VHE: Improved support for Type-2 Hypervisors (KVM)

Rounding Double Multiply Add/Subtract

Atomic Instructions

Limited Order Regions

Hardware update of Access & Dirty Bits

v8.2-A

Half-Precision support

RAS

Statistical Profiling

v8.3-A

Improved support for nested Hypervisors

JavaScript conversion function

Complex Number support

Pointer Authentication

RCpc load/store

v8.4-A

Further support for nested virtualization.

Secure EL2.

16x16+32 FP.

Dot Product.

SHA512, SHA3

SM3, SM4

Timing Insensitivity of DP Instructions

Changes to memory model

Small page table support

Activity Monitors

MPAM

Memory model
virtualization
FP & SIMD
Crypto & security
Monitoring & debug

Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

Architecture timer

Virtualization

MPAM

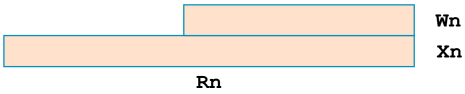
Security

RAS

AArch64 Registers

AArch64 provides 31 general purpose registers: `R0-R30`

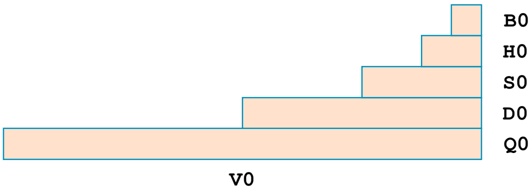
- Each register has a 64-bit (**`Xn`**) and 32-bit (**`Wn`**) form, common in all ELs



AArch64 introduces the “zero” register: XZR and WZR

Separate register file for floating point, SIMD, and crypto operations: `V0-V31`

- Each register has a 128-bit (**`Qn`**), 64-bit (**`Dn`**), 32-bit (**`Sn`**), 16-bit (**`Hn`**), and 8-bit (**`Bn`**) form



AArch64 Registers

AArch64 provides banked SP registers for each EL

AArch64 has PSTATE bit fields (not a register) to indicate or control program status

- Bit field could be changed individually, e.g.

NZCV	DFIA	SPSel	CurrentEL
ALU Flags	Exception Mask	SP select	Current EL indicator

- Access PSTATE
  - MSR

There are SPSR registers

- hold the saved process state when an exception

31 28	27 22	21	20	19 10	9 6	5	4 0
NZCV	RES0	SS	IL	RES0	DAIF	RES0	M

AArch64 Registers

In AArch64, system configuration is controlled through system registers

System registers are suffixed with "_ELx", for example SCTLR_EL1

- Suffix defines the lowest exception level that can access that system register
- For example:
  - TTBR0_EL1 Can be accessed from EL1, EL2, and EL3
  - TTBR0_EL3 Can be accessed from EL3

Use the MRS instruction to read a system register, and MSR instruction to write to a system register

```
– MRS    X0, SCTLR_EL1        ; X0 = SCTLR_EL1
– MSR    SCTLR_EL1, X0        ; SCTLR_EL1 = X0
```

Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

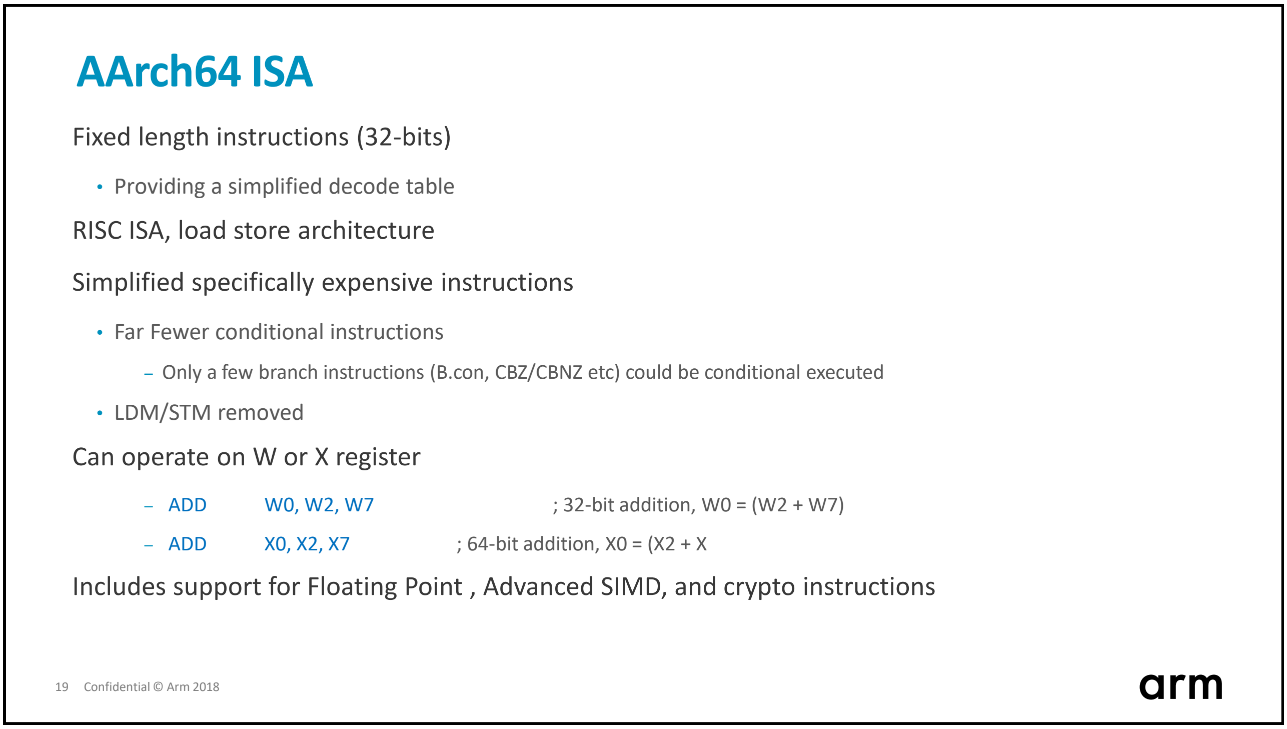
Architecture timer

Virtualization

MPAM

Security

RAS



AArch64 ISA

Fixed length instructions (32-bits)

- Providing a simplified decode table

RISC ISA, load store architecture

Simplified specifically expensive instructions

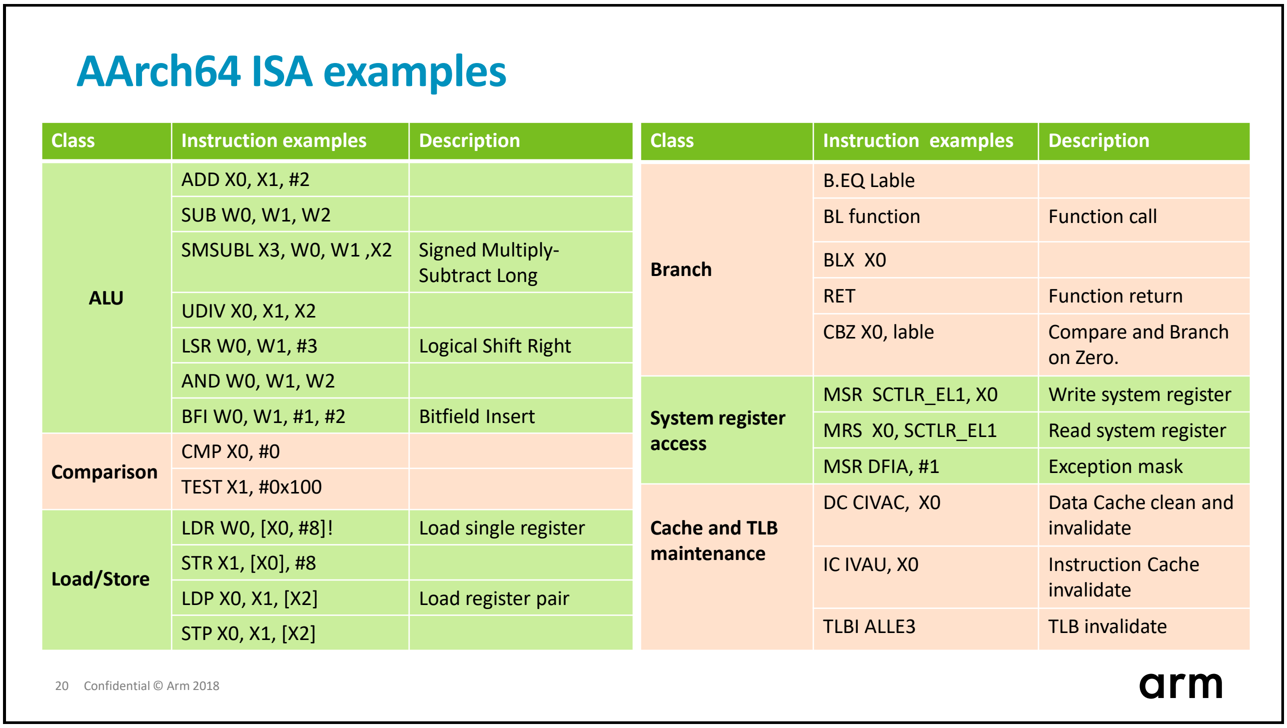
- Far Fewer conditional instructions
  - Only a few branch instructions (B.con, CBZ/CBNZ etc) could be conditional executed
- LDM/STM removed

Can operate on W or X register

```
– ADD    W0, W2, W7              ; 32-bit addition, W0 = (W2 + W7)
– ADD    X0, X2, X7         ; 64-bit addition, X0 = (X2 + X
```

Includes support for Floating Point , Advanced SIMD, and crypto instructions

arm



AArch64 ISA examples

Class	Instruction examples	Description
ALU	ADD X0, X1, #2	
	SUB W0, W1, W2	
	SMSUBL X3, W0, W1 ,X2	Signed Multiply-Subtract Long
	UDIV X0, X1, X2	
	LSR W0, W1, #3	Logical Shift Right
	AND W0, W1, W2	
	BFI W0, W1, #1, #2	Bitfield Insert
Comparison	CMP X0, #0	
	TEST X1, #0x100	
Load/Store	LDR W0, [X0, #8]!	Load single register
	STR X1, [X0], #8	
	LDP X0, X1, [X2]	Load register pair
	STP X0, X1, [X2]	

Class	Instruction examples	Description
Branch	B.EQ Lable	
	BL function	Function call
	BLX X0	
	RET	Function return
	CBZ X0, lable	Compare and Branch on Zero.
System register access	MSR SCTLR_EL1, X0	Write system register
	MRS X0, SCTLR_EL1	Read system register
	MSR DFIA, #1	Exception mask
Cache and TLB maintenance	DC CIVAC, X0	Data Cache clean and invalidate
	IC IVAU, X0	Instruction Cache invalidate
	TLBI ALLE3	TLB invalidate

arm

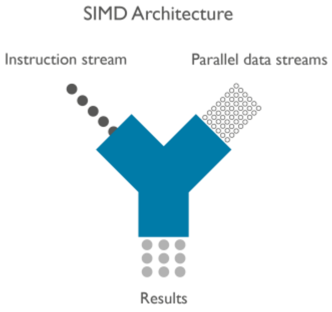
AArch64 Advance SIMD ISA

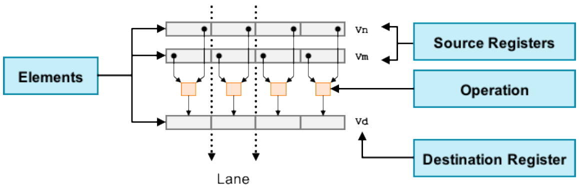
NEON is a wide SIMD data processing architecture

- Extension of the ARM® instruction set
- Armv8 NEON has 32 registers, 128-bits wide.

NEON Instructions perform “Packed SIMD” processing

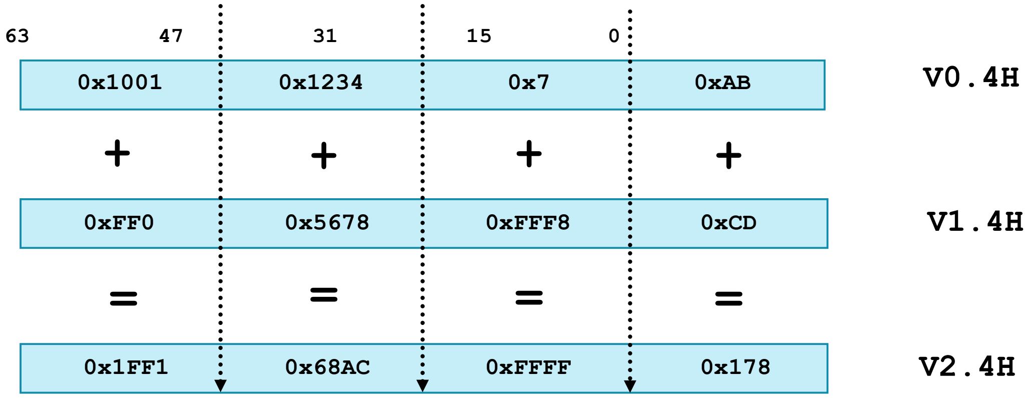
- Registers are considered as vectors of elements of the same data type
- Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single/double prec. Float (Armv8.2-A also support half prec Float)
- Instructions perform the same operation in all lanes





Sample SIMD Instructions - ADD

- `ADD V2.4H, V0.4H, V1.4H`
  - Register split into equal size and type elements (four 16-bit elements)
  - Operation performed on same element of each register



63 47 31 15 0

0x1001 0x1234 0x7 0xAB V0.4H

+ + + +

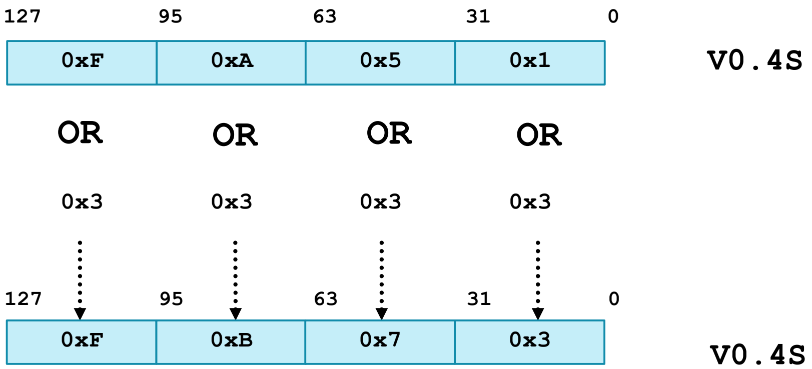
0xFF0 0x5678 0xFFF8 0xCD V1.4H

= = = =

0x1FF1 0x68AC 0xFFFF 0x178 V2.4H

Sample Instructions - ORR

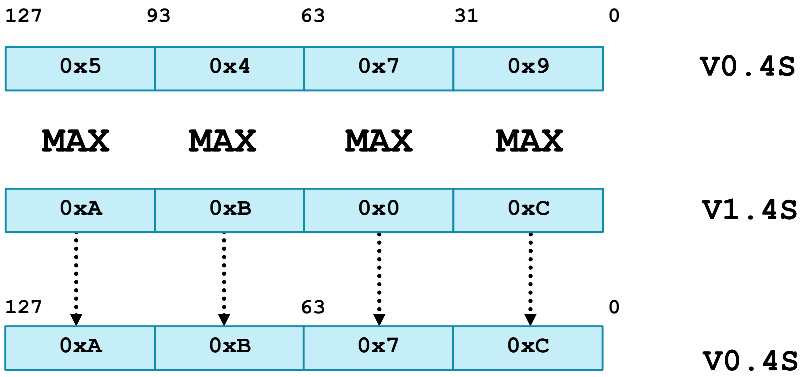
- `ORR V0.4S, #0x3`
  - Bitwise inclusive OR each element of `V0.4S` with constant `0x3`



Sample Instructions - SMAX

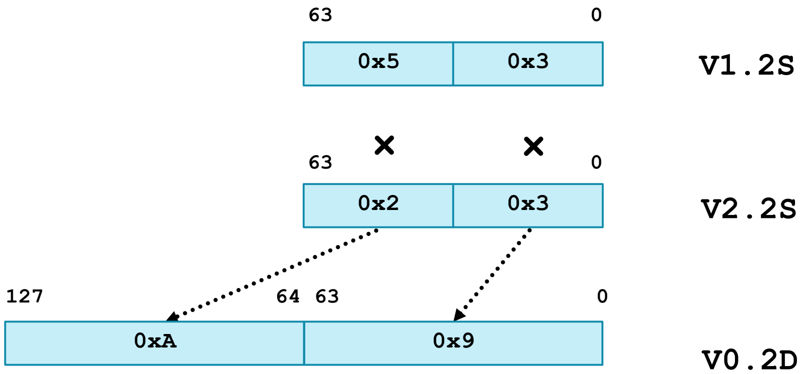
`SMAX V0.4S,V0.4S,V1.4S`

- Get the signed maximum of the same elements in `V0` and `V1`



Sample Instructions - `SMULL`

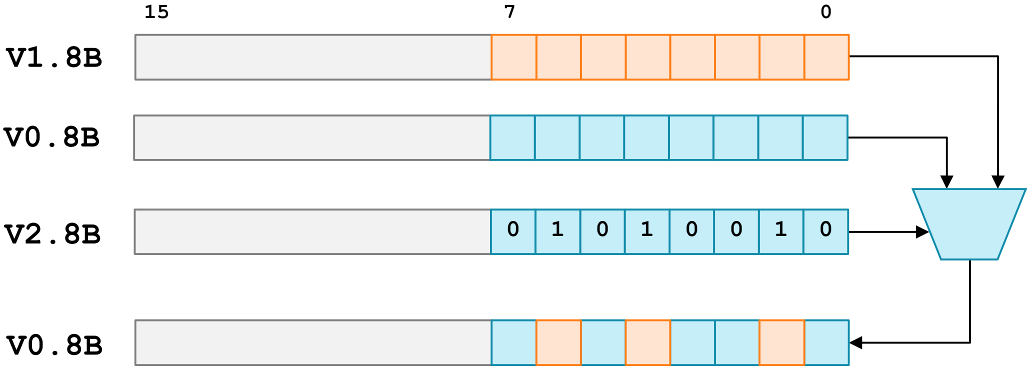
- `SMULL  V0.2D,V1.2S,V2.2S`
  - Signed multiply long - gets 64-bit integer elements from multiplication of 32-bit integer elements



arm

Sample Instructions - `BIT`

- `BIT V0.8B,V1.8B,V2.8B`
  - Insert each bit from `V1` into `V0` if the corresponding bit of `V2` is `1`, otherwise leave `V0` bit unchanged

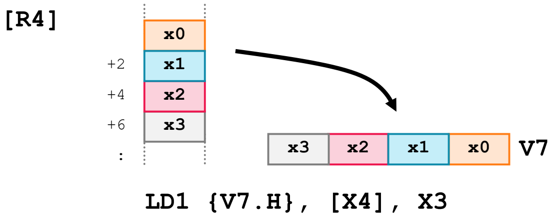


arm

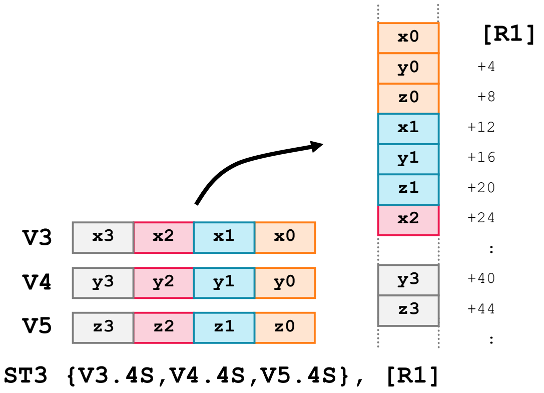
Multiple <n> - Element Structure Access

LD<n> and ST<n> can transfer multiple structures between memory and registers

- **LD1/ST1** can list 1 – 4 **V** registers
- **LD2/ST2** can list 2 V registers
- **LD3/ST3** must list 3 **V** registers
- **LD4/ST4** must list 4 **V** registers



`LD1 {V7.H}, [X4], X3`



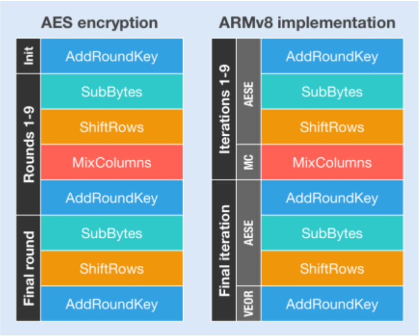
`ST3 {V3.4S,V4.4S,V5.4S}, [R1]`

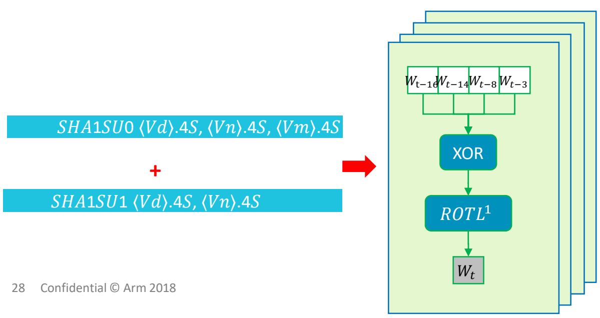
AArch64 Crypto ISA

Crypto extension AES

- **AESE** (AES single round encryption)
- **AESD** (AES single round decryption)
- **AESMC** (AES mix columns)
- **AESIMC** (AES inverse mix columns)

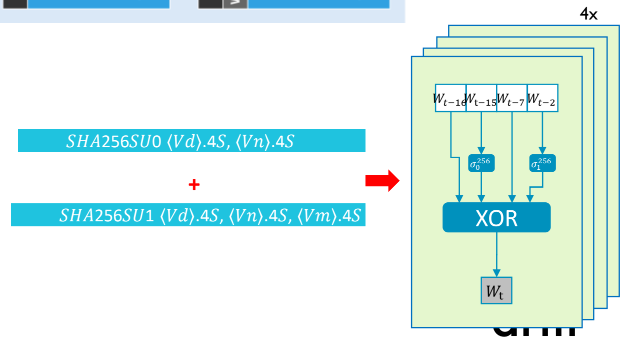
Crypto extension for SHA1 and SHA256




$SHA1SU0\ \langle Vd\rangle.4S,\ \langle Vn\rangle.4S,\ \langle Vm\rangle.4S$

+

$SHA1SU1\ \langle Vd\rangle.4S,\ \langle Vn\rangle.4S$


$SHA256SU0\ \langle Vd\rangle.4S,\ \langle Vn\rangle.4S$

+

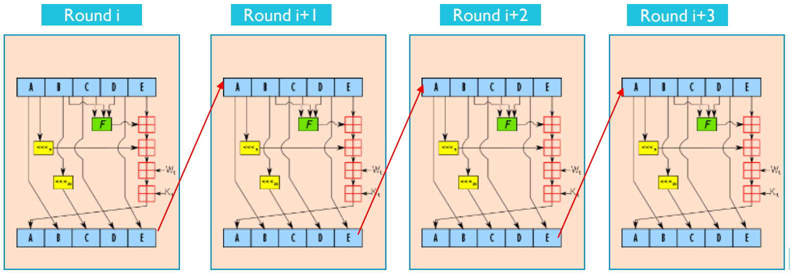
$SHA256SU1\ \langle Vd\rangle.4S,\ \langle Vn\rangle.4S,\ \langle Vm\rangle.4S$

AArch64 Crypto ISA

SHA1C/P/J ⟨Qd⟩, ⟨Sn⟩.4S,⟨Vm⟩.4S

+

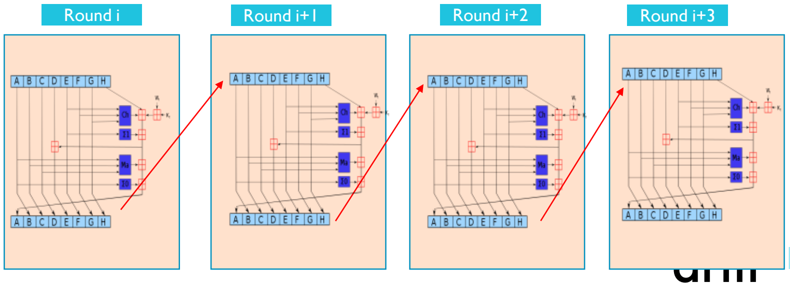
SHA1H ⟨Sd⟩, ⟨Sn⟩



SHA256H ⟨Qd⟩, ⟨Qn⟩, ⟨Vm⟩.4S

+

SHA256H2 ⟨Qd⟩, ⟨Qn⟩, ⟨Vm⟩.4S

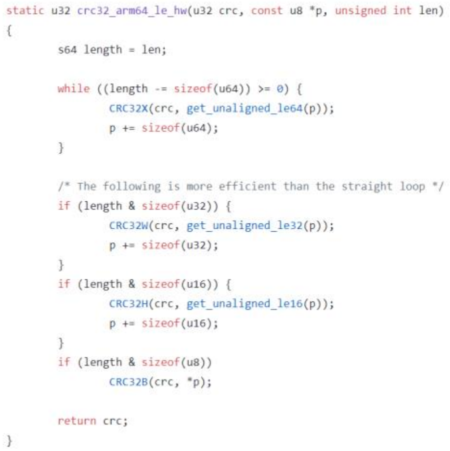


ARMv8 CRC32 instructions

CRC instructions operate on the general-purpose register file to update a 32-bit CRC sum from an input value of 8, 16, 32 or 64 bits.

Two different families of CRC instruction are provided to support two commonly used polynomials.

Example instruction	Description
CRC32B Wd, Wn, Wm	Accumulate one byte of input data from Wm<7:0> into the 32-bit CRC sum from Wn, and write the updated sum to Wd. Uses a polynomial of 0x04C11DB7.
CRC32X Wd, Wn, Xm	Accumulate one doubleword (eight bytes) of input data from Xm into the 32-bit CRC sum from Wn, and write the updated sum to Wd. Uses a polynomial of 0x04C11DB7.
CRC32CB Wd, Wn, Wm	Accumulate one byte of input data from Wm<7:0> into the 32-bit CRC sum from Wn, and write the updated sum to Wd. Uses a polynomial of 0x1EDC6F41.
CRC32CX Wd, Wn, Xm	Accumulate one doubleword (eight bytes) of input data from Xm into the 32-bit CRC sum from Wn, and write the updated sum to Wd. Uses a polynomial of 0x1EDC6F41.



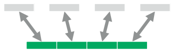
```
static u32 crc32_arm64_le_hw(u32 crc, const u8 *p, unsigned int len)
{
        s64 length = len;

        while ((length -= sizeof(u64)) >= 0) {
                CRC32X(crc, get_unaligned_le64(p));
                p += sizeof(u64);
        }

        /* The following is more efficient than the straight loop */
        if (length & sizeof(u32)) {
                CRC32W(crc, get_unaligned_le32(p));
                p += sizeof(u32);
        }
        if (length & sizeof(u16)) {
                CRC32H(crc, get_unaligned_le16(p));
                p += sizeof(u16);
        }
        if (length & sizeof(u8))
                CRC32B(crc, *p);

        return crc;
}
```

Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

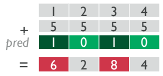
SVE

A vector extension to the Armv8-A architecture with some major new features:



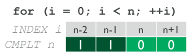
Gather-load and scatter-store

Loads a single register from several non-contiguous memory locations.



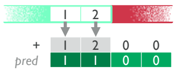
Per-lane predication

Operations work on individual lanes under control of a predicate register.



Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.



Vector partitioning and software-managed speculation

First Faulting Load instructions allow memory accesses to cross into invalid pages.

No preferred vector width

The above features allow the production of compiled binaries that are agnostic to hardware vector width (which can be between 128-2048 bit at 128 bit increments).

New SVE architectural state

Scalable vector registers

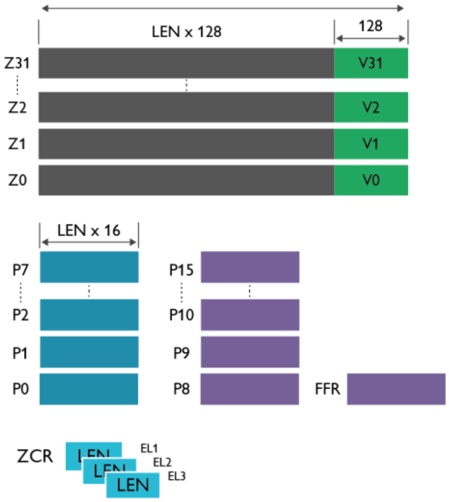
- Z0-Z31 extending NEON's V0-V31.
- Packed DP, SP & HP floating-point elements.
- Packed 64, 32, 16 & 8-bit integer elements.

Scalable predicate registers

- P0-P7 governing predicates for load/store/arithmetic.
- P8-P15 additional predicates for loop management.
- FFR first fault register for speculation.

Scalable vector control registers

- ZCR_ELx vector length (LEN=1..16).
- For exception/privilege levels EL1 to EL3.



Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

AArch64 exceptions

In AArch64, exceptions are split between:

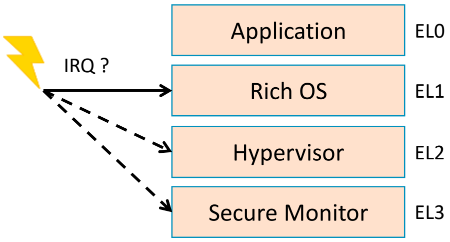
- Synchronous: Data Aborts from the MMU, Permission Faults, Alignment Faults, service call instructions (e.g. `SVC`), etc
- Asynchronous: IRQs, FIQs, SErrors (System Errors)

On taking an exception, the EL can either stay the same or get higher

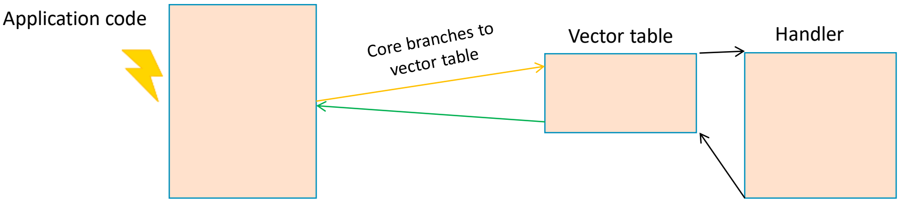
- Exceptions are never taken to EL0

Asynchronous exceptions can be routed to a higher EL

- `HCR_EL2` controls routing to EL2
- `SCR_EL3` controls routing to EL3
- Separate bits to control routing of IRQs, FIQs, and SErrors



Taking an exception



When an exception occurs:

- `SPSR_ELx` updated
- `PSTATE` updated (EL stays the same or gets higher)
- Return address stored to `ELR_ELx`
- PC set to vector address
- `ESR_ELx` updated with cause of exception
  - Only if synchronous or SError exception

Execute an `ERET` instruction to return from an exception:

- Restores `PSTATE` from `SPSR_ELx`
- Restores PC from `ELR_ELx`

Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

Architecture timer

Virtualization

MPAM

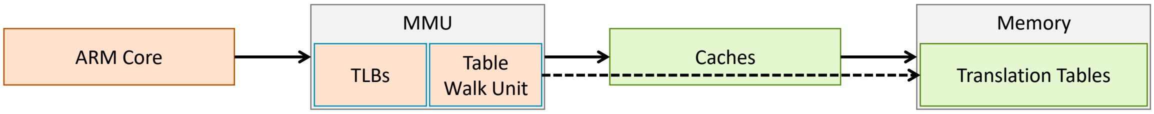
Security

RAS

What is a Memory Management Unit?

- **The MMU handles the translation of virtual address to physical addresses**
  - Provides hardware to read translation tables in memory ("table walking")
    - Translation Table Base Registers (`TTBR`) hold the physical base address of the tables
  - Translation Lookaside Buffers (TLBs) cache recent translations

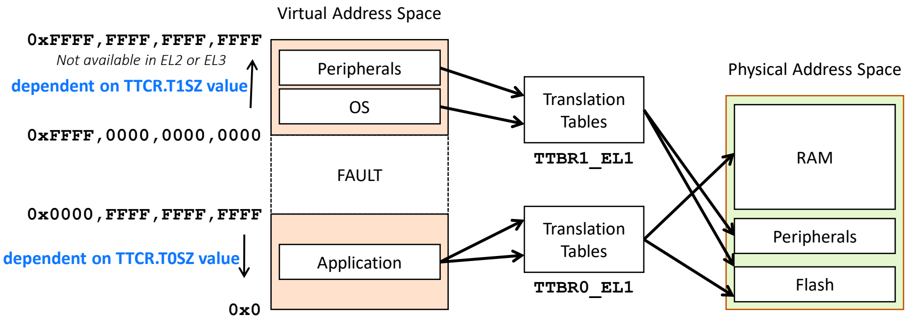
- **When the MMU is enabled, all accesses made by the core are passed through it**
  - MMU will use cached translations from the TLB(s), or perform a table walk
  - Translation must occur before a cache lookup can complete



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_ELx`)
- Addresses not covered by either translation table automatically generate translation faults



Virtual Address Space

0xFFFF,FFFF,FFFF,FFFF

Not available in EL2 or EL3

dependent on TTCR.T1SZ value

0xFFFF,0000,0000,0000

0x0000,FFFF,FFFF,FFFF

dependent on TTCR.T0SZ value

0x0

Peripherals

OS

FAULT

Application

Translation Tables

TTBR1_EL1

Translation Tables

TTBR0_EL1

Physical Address Space

RAM

Peripherals

Flash

Multiple virtual address spaces

A system may define multiple virtual address spaces

OS / Applications

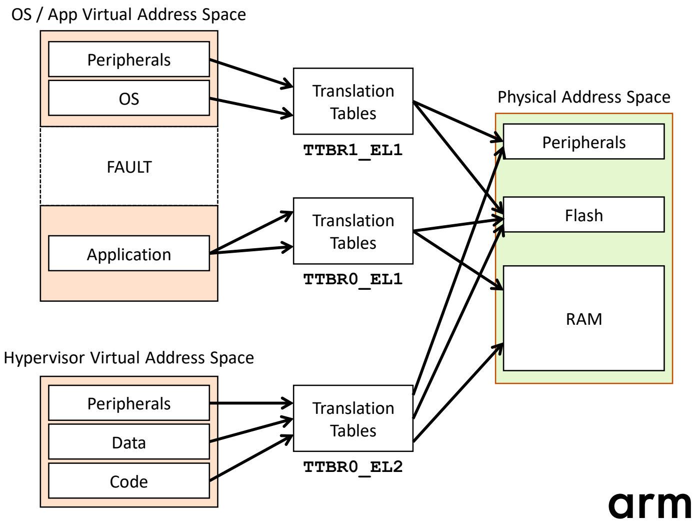
- `TTBR0_EL1`
- `TTBR1_EL1`
- `TCR_EL1`

Hypervisor

- `TTBR0_EL2`
- `TCR_EL2`

Secure Monitor

- `TTBR0_EL3`
- `TCR_EL3`



OS / App Virtual Address Space

Peripherals

OS

FAULT

Application

Translation Tables

TTBR1_EL1

Translation Tables

TTBR0_EL1

Physical Address Space

Peripherals

Flash

RAM

Hypervisor Virtual Address Space

Peripherals

Data

Code

Translation Tables

TTBR0_EL2

AArch64 Translation Tables

AArch64 supports 3 different translation granules

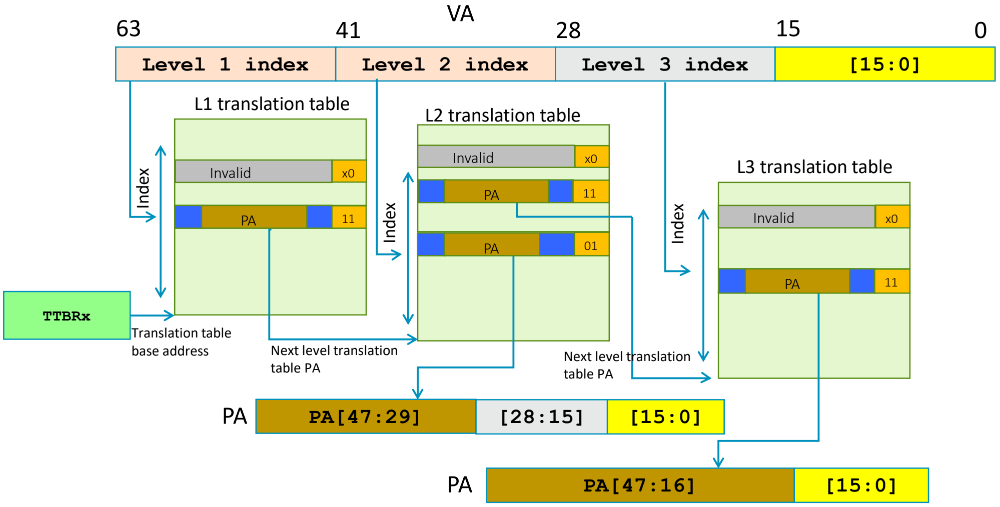
- 4KB, 16KB, or 64KB
  - Defines block size at lowest level of translation table and size of tables
- Configurable for each TTBR
- It is IMPLEMENTATION DEFINED which of the three are supported
  - `ID_AA64MMFR0_EL1` reports supported sizes

Larger granules reduce the number of levels of table required

- Particularly important in Virtualized systems

Translation table and address translation

64KB Granule translation example



Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

Weak memory mode and memory types

The ARM architecture defines a weak ordering model

- This means that accesses *might not* occur in program order
- ARM memory consistency model allows:

Memory access to different locations	Reordered?
Loads reorders after loads	Yes
Loads reorders after stores	Yes
Stores reorders after stores	Yes
Loads reorders after stores	Yes
Atomic reordered with loads	Yes
Atomic reordered with stores	Yes
Dependent loads reordered	No

ARM architecture defines memory types to tells the processor how it can access that location

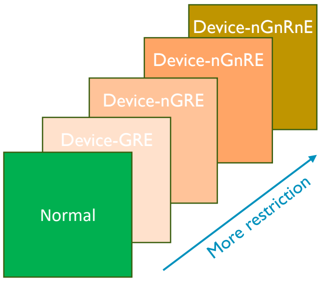
- Normal memory
- Device memory

arm

Memory types

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

Speculative instruction fetches are allowed to any region that's executable at *some* exception level

Memory Types: Normal

The Normal type is used for code and most data regions

Normal memory gives the best performance because it imposes the fewest restrictions

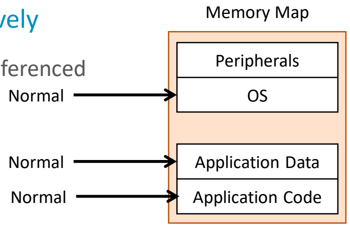
- Allows the processor to re-order, repeat, and merge accesses

For optimal performance, application code and data should be marked as Normal

- Ordering can still be enforced when required using explicit barrier operations

Address regions marked as Normal can be accessed speculatively

- Data or instructions fetched from memory before being explicitly referenced



Memory Types: Device

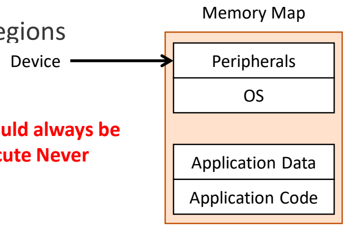
The Device type is used for regions where accesses can have side-effects

- Example: Writing to a peripheral’s control register may trigger an interrupt as a side-effect
- Typically only used for peripherals

Device type imposes more restrictions on the core

Attempting to execute from a region marked as Device is UNPREDICTABLE

Speculative data accesses cannot be performed to Device regions

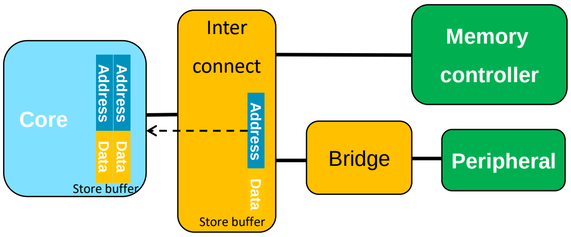


Device regions should always be marked as Execute Never

Memory Types: Device (2)

Four variants of Device are available:

- Device-nGnRnE - most restrictive
- Device-nGnRE
- Device-nGRE
- Device-GRE - least restrictive



Gathering (G, nG)

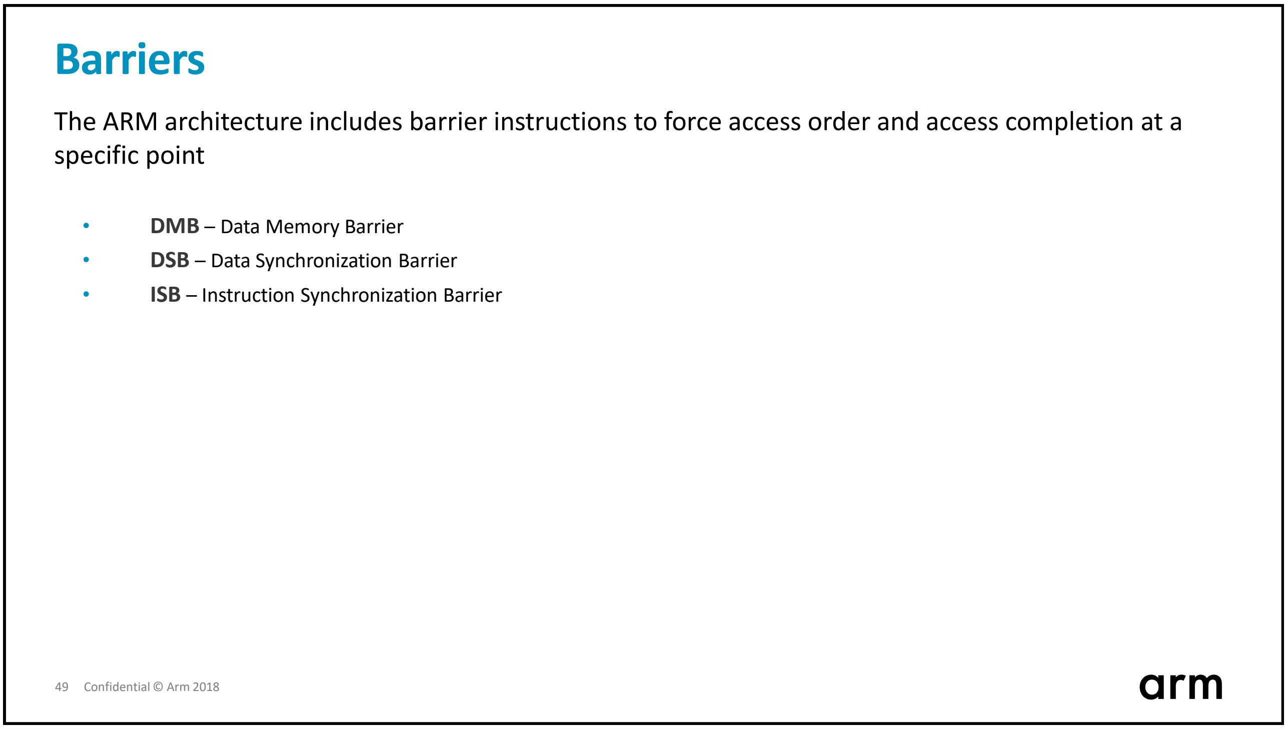
- Determines whether multiple accesses can be merged into a single bus transaction
- nG: number/size of accesses on the bus = number/size of accesses in code

Re-ordering (R, nR)

- Determines whether accesses to same device can be re-ordered
- nR: accesses to the same IMPLEMENTATION DEFINED block size will appear on the bus in program order

Early Write Acknowledgement (E, nE)

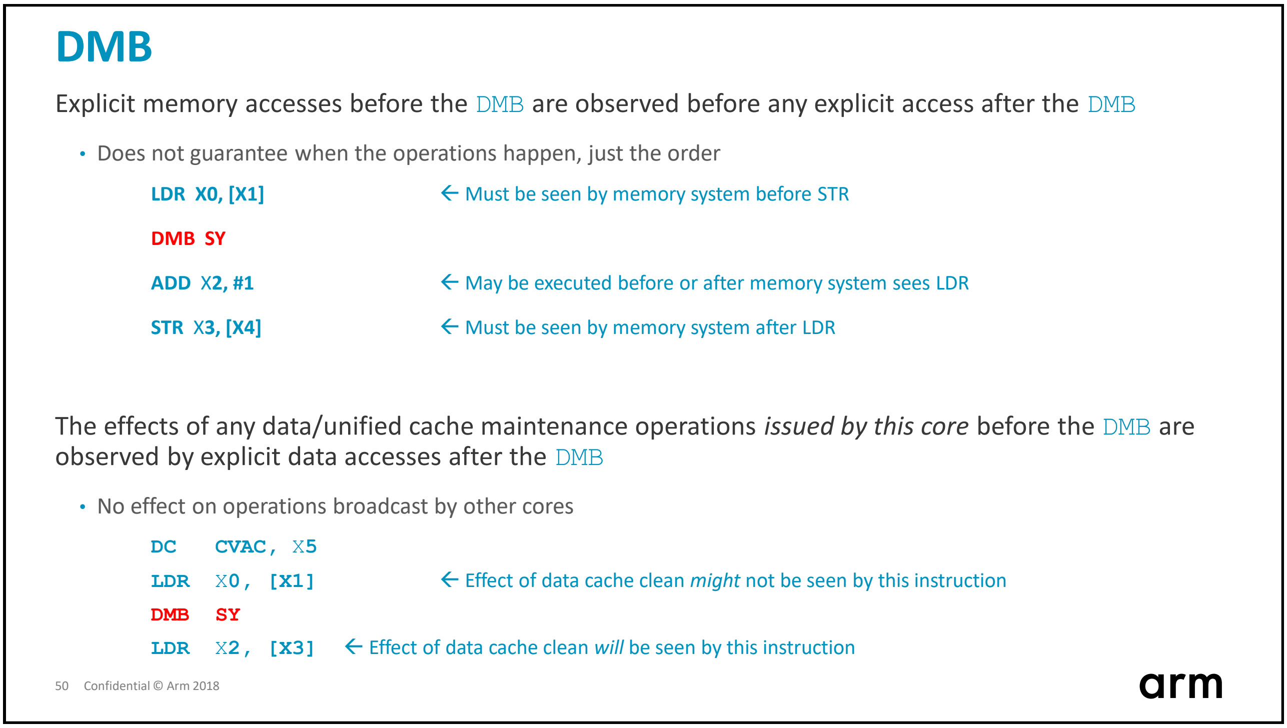
- Indicates to the memory system whether a buffer can send acknowledgements
- nE: The response should come from the end slave, not buffering in interconnect



Barriers

The ARM architecture includes barrier instructions to force access order and access completion at a specific point

- **DMB** – Data Memory Barrier
- **DSB** – Data Synchronization Barrier
- **ISB** – Instruction Synchronization Barrier



DMB

Explicit memory accesses before the DMB are observed before any explicit access after the DMB

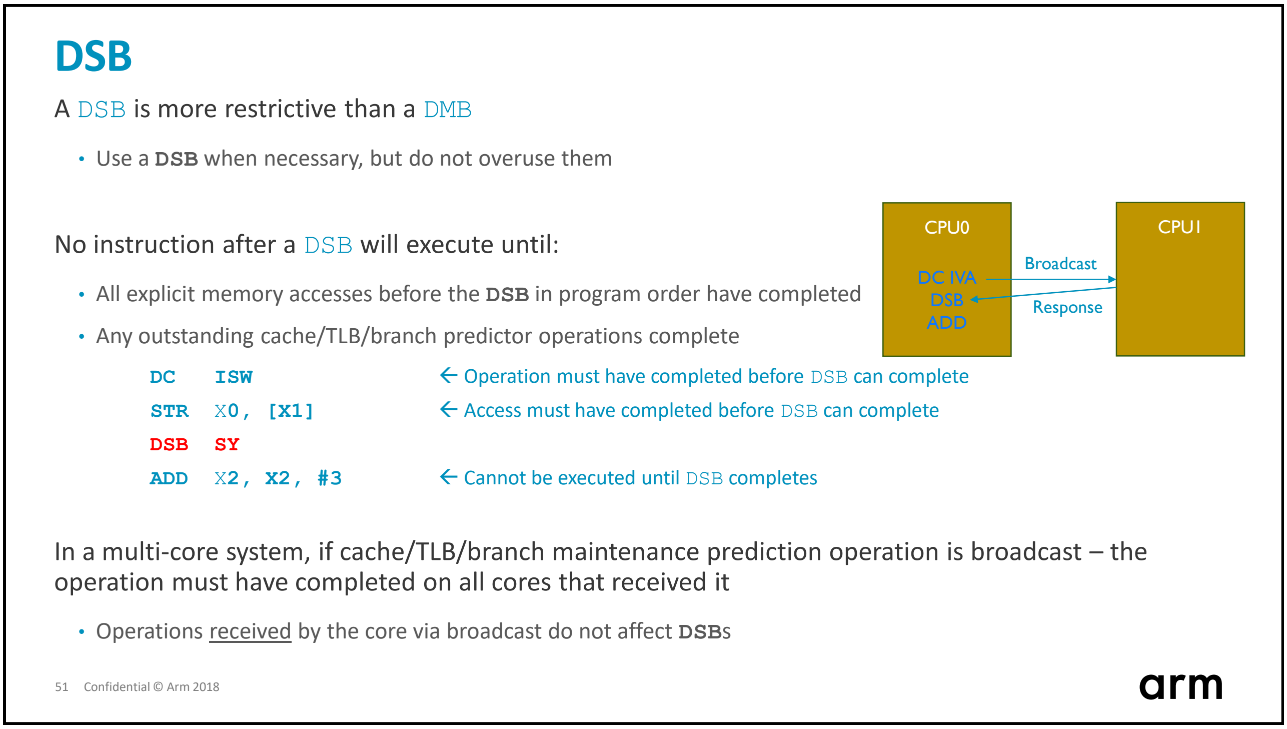
- Does not guarantee when the operations happen, just the order

```
LDR  X0, [X1]            ← Must be seen by memory system before STR

DMB  SY

ADD  X2, #1              ← May be executed before or after memory system sees LDR

STR  X3, [X4]            ← Must be seen by memory system after LDR
```

The effects of any data/unified cache maintenance operations *issued by this core* before the DMB are observed by explicit data accesses after the DMB

- No effect on operations broadcast by other cores

```
DC    CVAC, X5
LDR   X0, [X1]           ← Effect of data cache clean might not be seen by this instruction
DMB   SY
LDR   X2, [X3]   ← Effect of data cache clean will be seen by this instruction
```



DSB

A DSB is more restrictive than a DMB

- Use a **DSB** when necessary, but do not overuse them

No instruction after a DSB will execute until:

- All explicit memory accesses before the **DSB** in program order have completed
- Any outstanding cache/TLB/branch predictor operations complete

CPU0

DC IVA
DSB
ADD

Broadcast

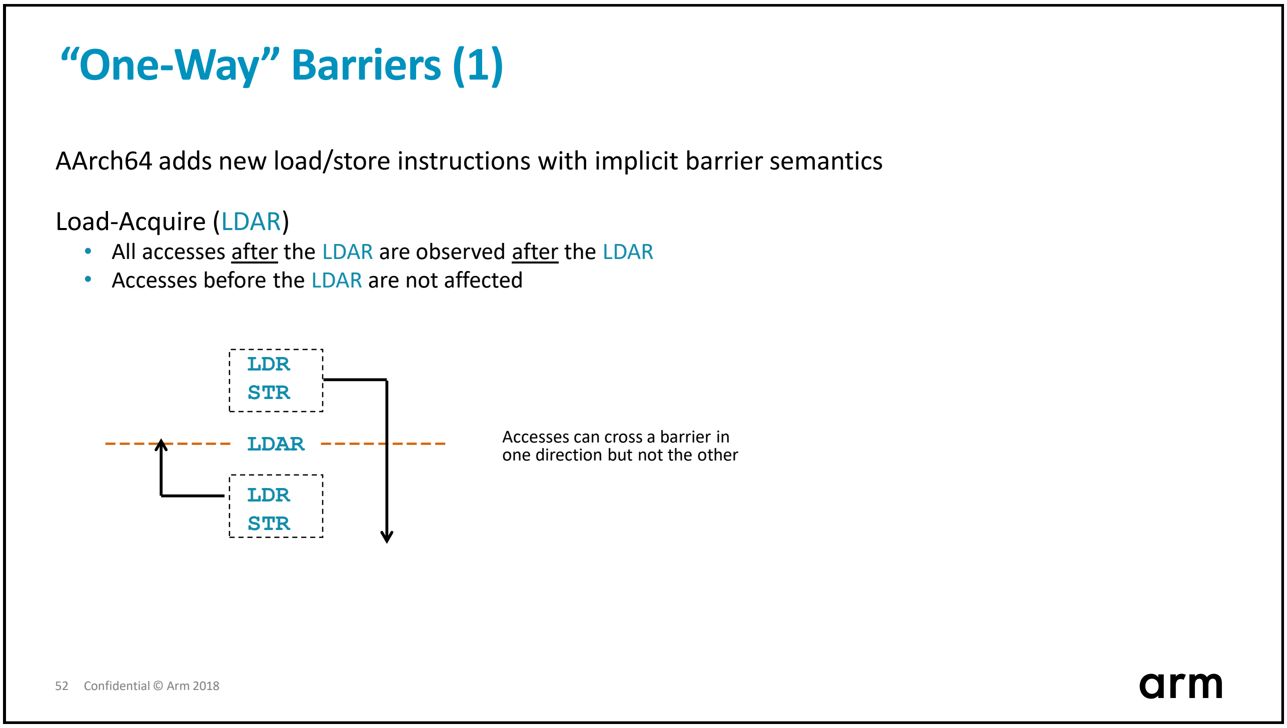
Response

CPU1

```
DC    ISW                 ← Operation must have completed before DSB can complete
STR   X0, [X1]            ← Access must have completed before DSB can complete
DSB   SY
ADD   X2, X2, #3          ← Cannot be executed until DSB completes
```

In a multi-core system, if cache/TLB/branch maintenance prediction operation is broadcast – the operation must have completed on all cores that received it

- Operations received by the core via broadcast do not affect **DSB**s



“One-Way” Barriers (1)

AArch64 adds new load/store instructions with implicit barrier semantics

Load-Acquire (LDAR)

- All accesses after the LDAR are observed after the LDAR
- Accesses before the LDAR are not affected

```
LDR
STR
LDAR
LDR
STR
```

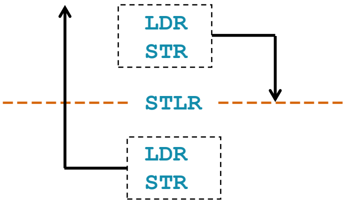
Accesses can cross a barrier in
one direction but not the other

“One-Way” Barriers (2)

AArch64 adds new load/store instructions with implicit barrier semantics

Store-Release (STLR)

- All accesses before the **STLR** are observed before the **STLR**
- Accesses after the **STLR** are not affected



Accesses can cross a barrier in
one direction but not the other

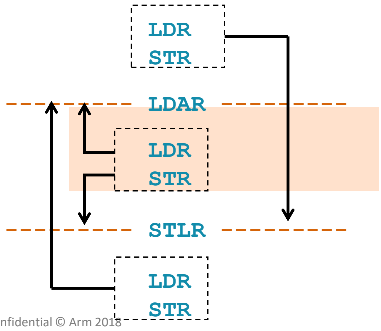
“One-Way” Barriers (3)

LDAR and STLR may be used as a pair

- To protect a critical section of code
- May have lower performance impact than a full **DMB**
- No ordering is enforced *within* the critical section

Exclusive versions also available

- **LDAXR**, **STLXR**
- Remove the need for explicit barrier instructions in synchronization code



Critical code section

Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

Architecture timer

Virtualization

MPAM

Security

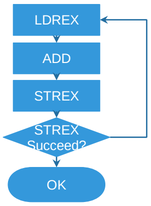
RAS

Atomic

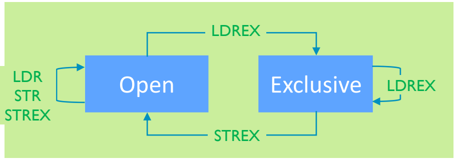
load/store exclusive

- Split the operation of atomically updating memory into two separate steps, Load exclusive and store exclusive.
- Together, they provide atomic updates in conjunction with exclusive monitors that track exclusive memory accesses

```
1:  ldrex  w0, [var]
    add    w0, w0, #num
    strex  w1, w0, [var]
    teq    w1, #0
    bne    1b
```



Exclusive Monitor Logic

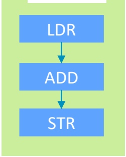


Atomic operations in Armv8.1-a onwards with Large System Extension

- add, bitwise ops, max/min in variants with and without returning a result
- swap, compare and swap

```
LDADD  w1, w2, [x0]
```

Atomic



Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

Architecture timer

Virtualization

MPAM

Security

RAS

Architecture timer

System counter generates a fixed frequency incrementing count, which is distributed to all cores

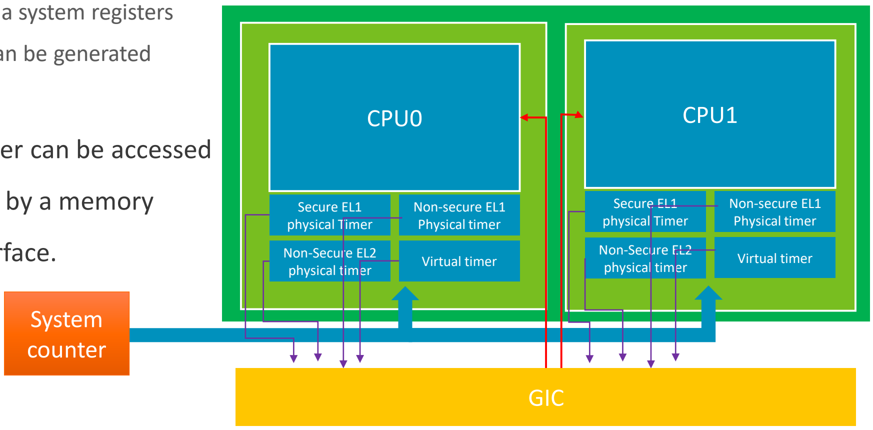
Each core implements a number of comparators

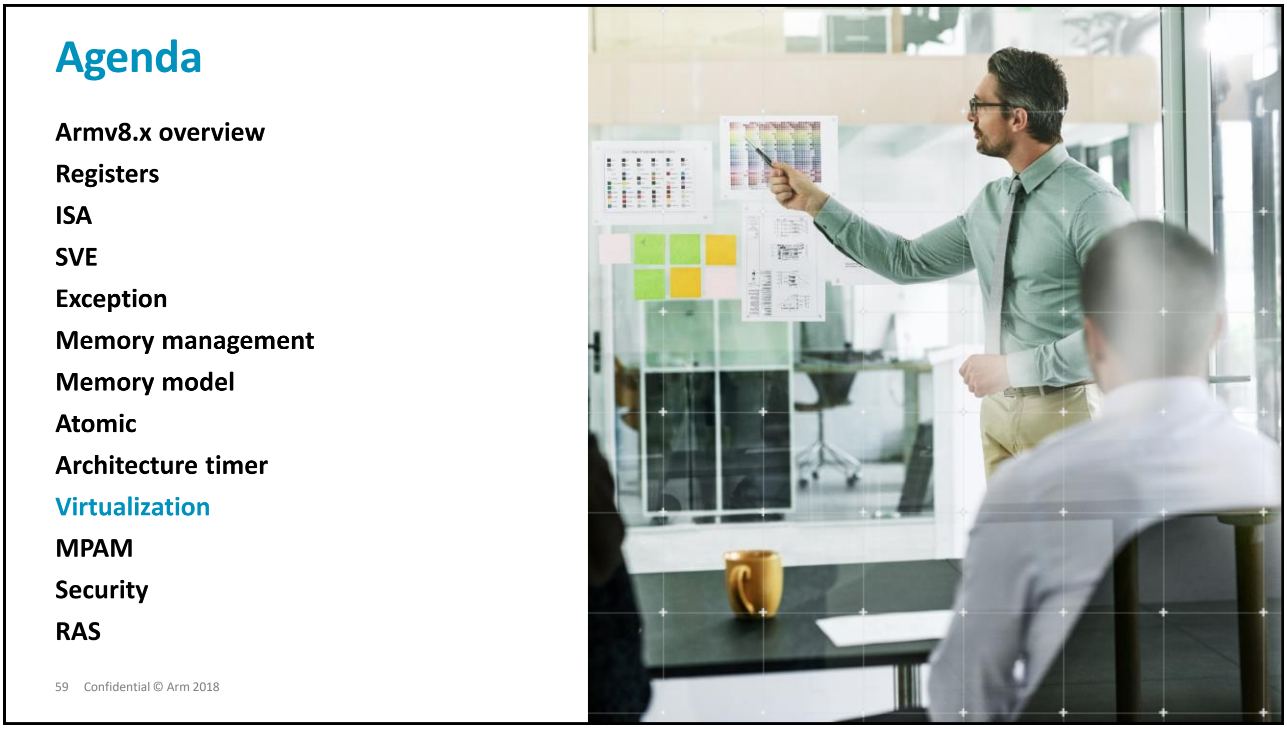
- Secure physical EL1, Non-secure physical EL2, Non-secure EL1 physical and Virtual comparators
- Accessed via system registers
- Interrupt can be generated

System counter can be accessed

Or controlled by a memory

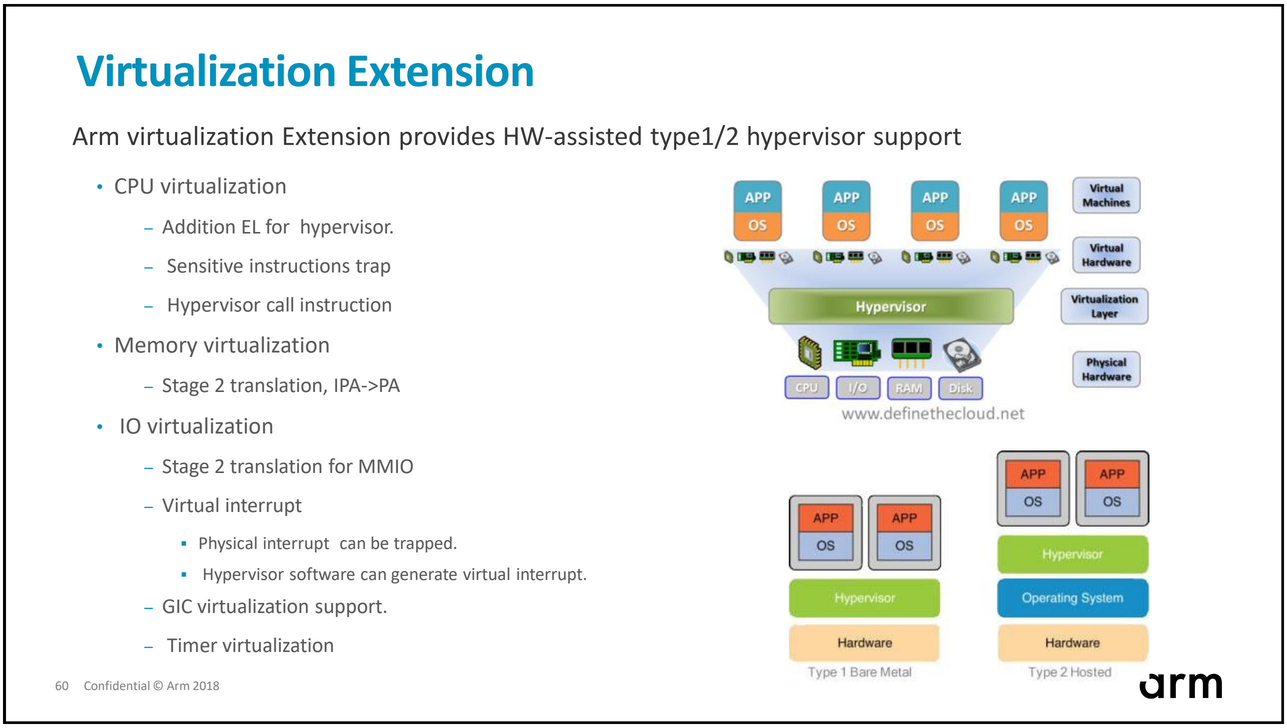
Mapped interface.


CPU0
Secure EL1 physical Timer
Non-secure EL1 Physical timer
Non-Secure EL2 physical timer
Virtual timer
CPU1
Secure EL1 physical Timer
Non-secure EL1 Physical timer
Non-Secure EL2 physical timer
Virtual timer
System counter
GIC



Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS



Virtualization Extension

Arm virtualization Extension provides HW-assisted type1/2 hypervisor support

- CPU virtualization
  - Addition EL for hypervisor.
  - Sensitive instructions trap
  - Hypervisor call instruction
- Memory virtualization
  - Stage 2 translation, IPA->PA
- IO virtualization
  - Stage 2 translation for MMIO
  - Virtual interrupt
    - Physical interrupt can be trapped.
    - Hypervisor software can generate virtual interrupt.
  - GIC virtualization support.
  - Timer virtualization

www.definethecloud.net

Type 1 Bare Metal

Type 2 Hosted

Virtualization Extension- CPU Virtualization

The Hypervisor can be configured to trap certain instructions

- Configured through Hypervisor Control Register (HCR_EL2)
- When trapped, Exception Syndrome Register (ESR_EL2) provides information about trapped instruction

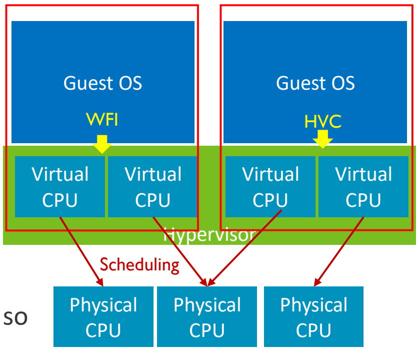
Instructions could be trapped

- System instructions, e.g. cache and TLB maintenance instructions
- Accesses to Auxiliary Control Register (ACTLR_EL1) Reads to ID registers
- WFE and WFI instructions

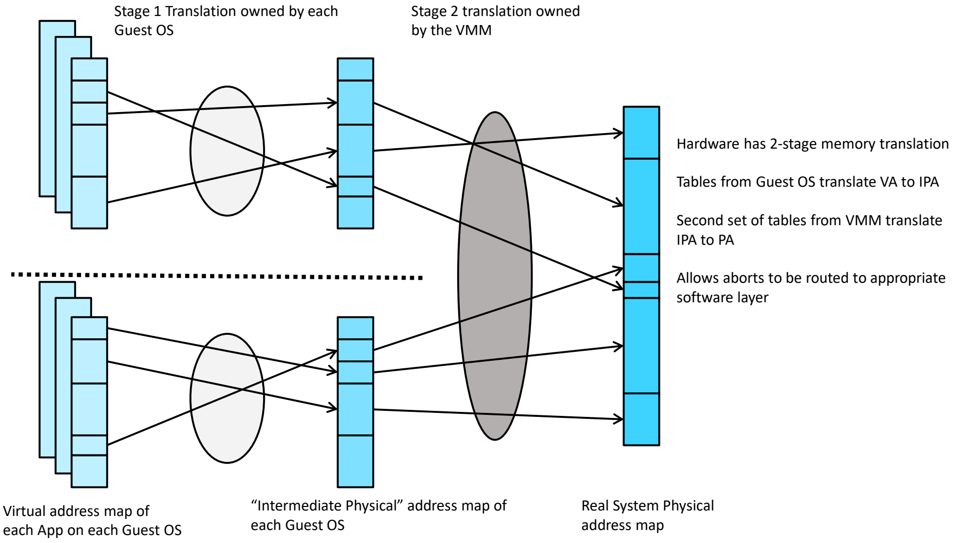
Some registers have dedicated control for virtualization purpose, so no trapping is required

- E.g., MIDR_EL1, MPIDR_EL1

Hypervisor calls -HVC, guest OS could call services provided by hypervisor



Virtual Memory in two Stages


Stage 1 Translation owned by each Guest OS
Stage 2 translation owned by the VMM
Virtual address map of each App on each Guest OS
"Intermediate Physical" address map of each Guest OS
Real System Physical address map

Hardware has 2-stage memory translation

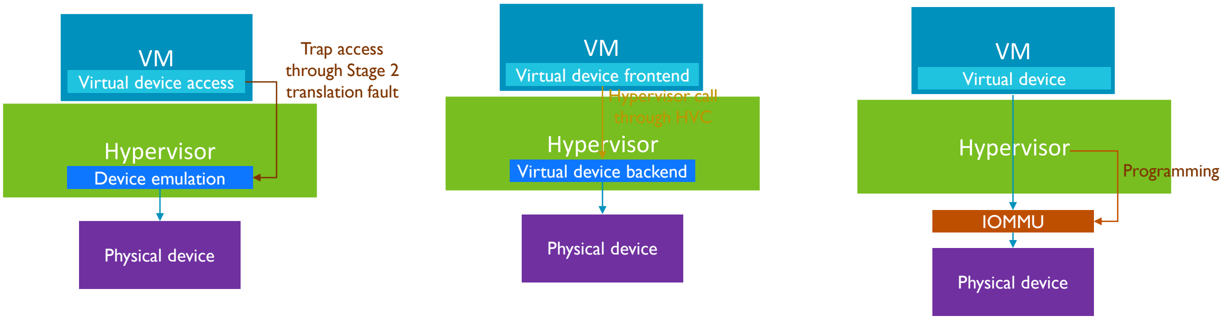
Tables from Guest OS translate VA to IPA

Second set of tables from VMM translate IPA to PA

Allows aborts to be routed to appropriate software layer

Virtualization Extension- Device virtualization

3 methods for device access virtualization

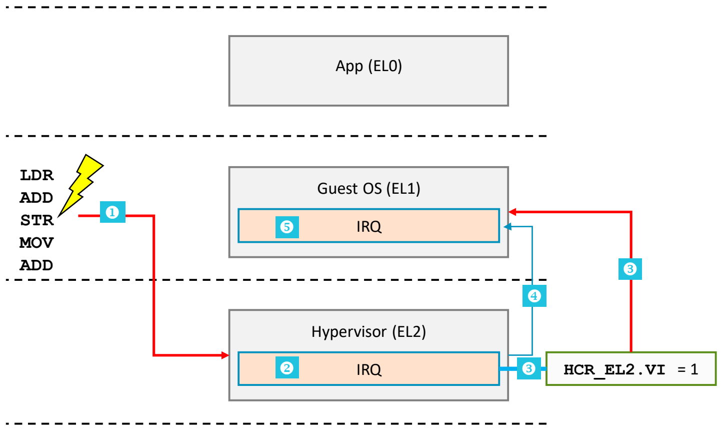


Emulated device

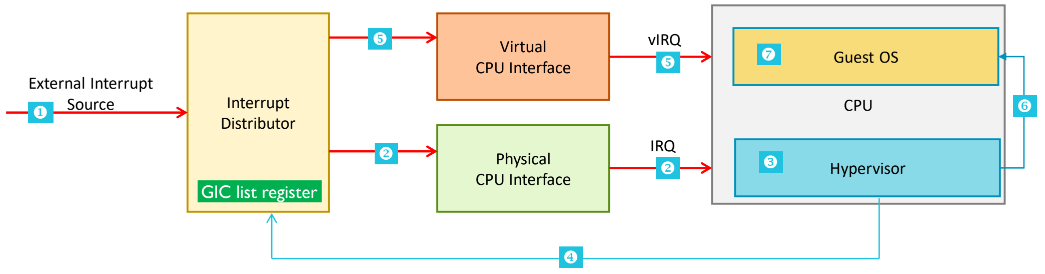
Paravirtualization

Direct access

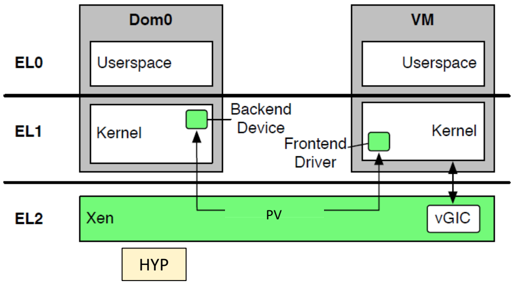
Virtual interrupt signaling (Internal)



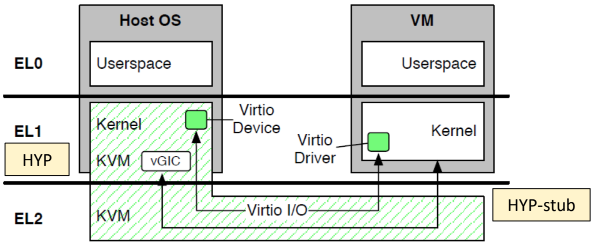
Virtual interrupt signaling (GIC)



Xen and KVM implementation



Xen – Type1 Hypervisor



KVM – Type2 Hypervisor

Overhead without Virtualization Host Extension(VHE)

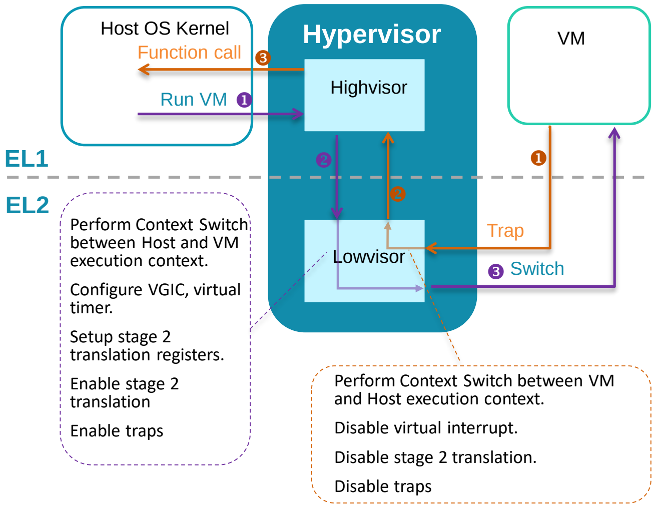
Host OS must run in both EL1 and EL2

- Requires kernel code to be aware where it is running

Calls to EL2 functions must be via HVC trap

- Required for any access to EL2 registers
- Configuring Stage 2 translations

Moving from Guest to Host OS requires a full context switch of EL1 system registers



Perform Context Switch between Host and VM execution context.

Configure VGIC, virtual timer.

Setup stage 2 translation registers.

Enable stage 2 translation

Enable traps

Perform Context Switch between VM and Host execution context.

Disable virtual interrupt.

Disable stage 2 translation.

Disable traps

Pre-VHE KVM Virtualization

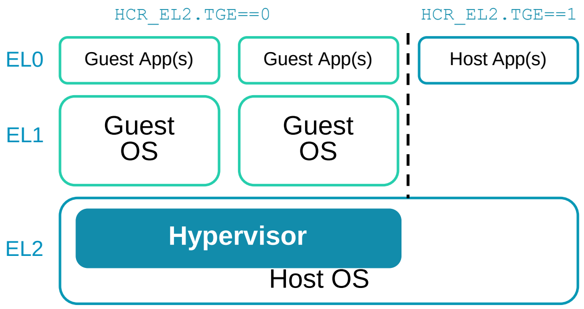
Armv8.2 Virtualization Host Extension(VHE)

VHE reduces the overhead of transaction between VM and hypervisor

- Allows host kernel and hypervisor totally run in the same EL
- Minimum the need of host kernel modification to support VHE

When software enables VHE (HCR_EL2.E2H=1)

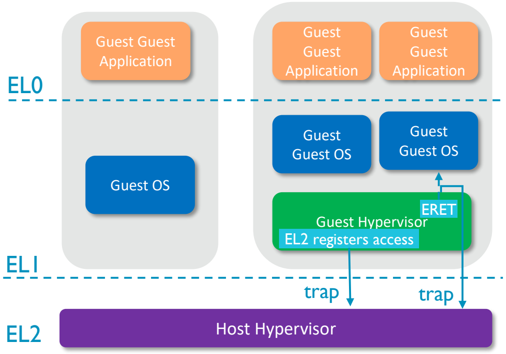
- EL2 translation regime supports two translation table TTBR0/1
- When HCR_EL2.TGE==1, translation for EL0 (running host application) uses EL0/EL2 translation regime, stage 2 translation is disabled.
  When HCR_EL2.TGE==0, translation for EL0 (running guest application) uses EL0/EL1 translation regime.
- Add ASID and VMID support in EL2.
- Add virtual timer support in EL2.
- Redirect TTBR0/1_EL1, SCTRL_EL1 etc registers access to EL2 physical registers.
- Introduce new XXX_EL12 (e.g TTBR0_EL12) registers in EL2.

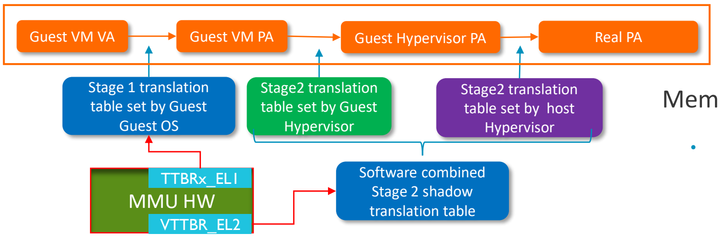


Armv8.3 Nested virtualization support

Supports running the guest hypervisor in EL1, controlled by HCR_EL2.NV

- Traps EL2 operations executed in EL1 to EL2
  - Emulating EL2 in EL1
  - shadow EL1 register state that reflects the EL2 register state
- Traps ERET to EL2




Guest VM VA
Guest VM PA
Guest Hypervisor PA
Real PA
Stage 1 translation table set by Guest Guest OS
Stage2 translation table set by Guest Hypervisor
Stage2 translation table set by host Hypervisor
TTBRx_EL1
MMU HW
VTTBR_EL2
Software combined Stage 2 shadow translation table

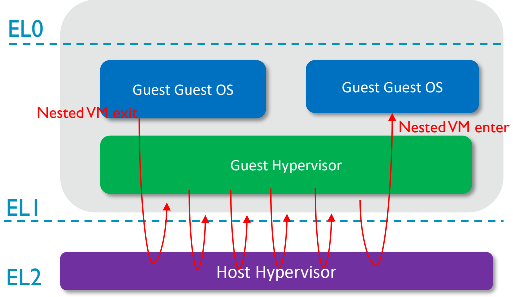
Memory Virtualization

- Software combines stage 2 translation of guest hypervisor and stage 2 translation of host hypervisor to a shadow translation table

Armv8.4 Nested virtualization support

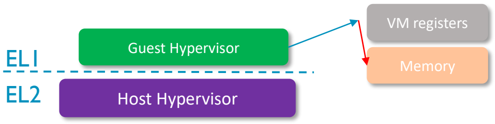
Nested Virtualization Extensions(NEVE)

- Provides two techniques to avoid traps based on register classification



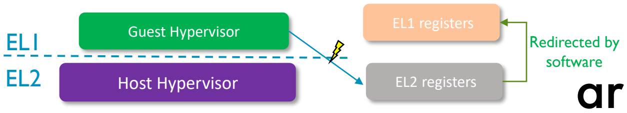
VM Registers: Redirection to Memory

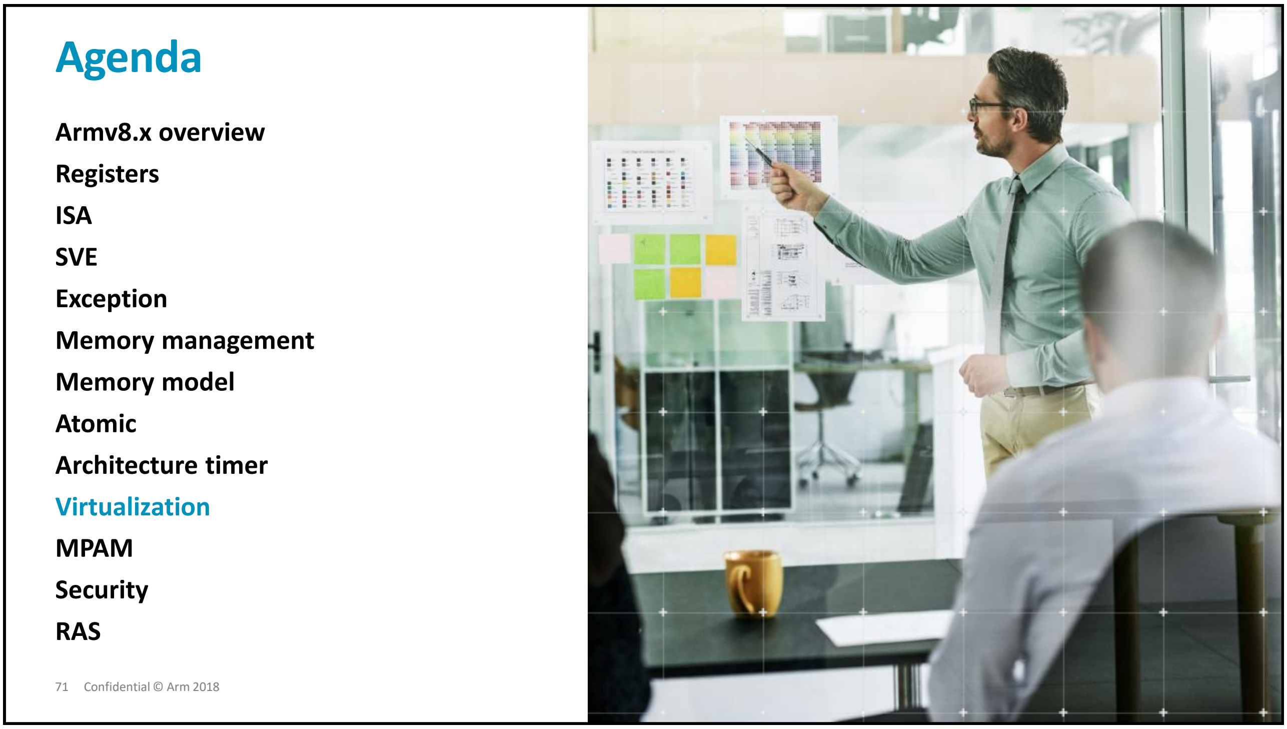
- NEVE redirects VM register access instructions to memory
- On nested VM entry, the host hypervisor can get VM register states from memory



Hypervisor Control Registers

- Traps are handled by redirecting to EL1 registers in software





Agenda

Armv8.x overview

Registers

ISA

SVE

Exception

Memory management

Memory model

Atomic

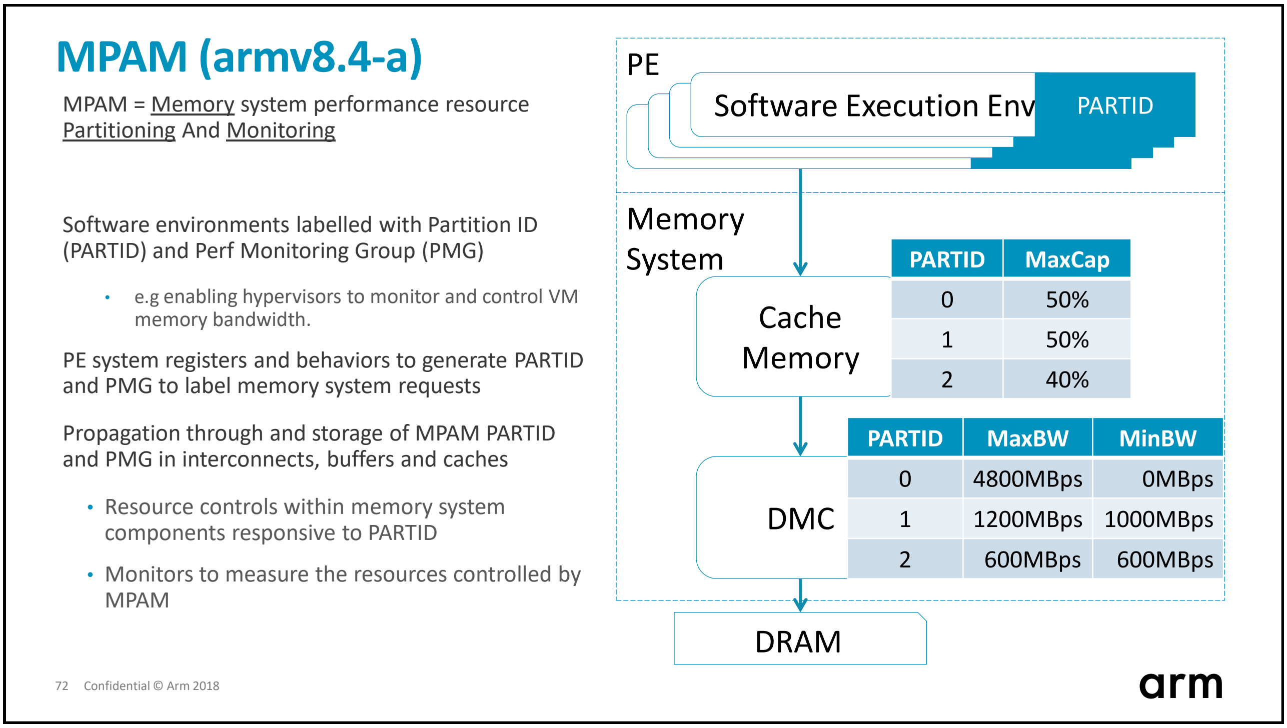
Architecture timer

Virtualization

MPAM

Security

RAS



MPAM (armv8.4-a)

MPAM = Memory system performance resource Partitioning And Monitoring

Software environments labelled with Partition ID (PARTID) and Perf Monitoring Group (PMG)

- e.g enabling hypervisors to monitor and control VM memory bandwidth.

PE system registers and behaviors to generate PARTID and PMG to label memory system requests

Propagation through and storage of MPAM PARTID and PMG in interconnects, buffers and caches

- Resource controls within memory system components responsive to PARTID
- Monitors to measure the resources controlled by MPAM

PARTID	MaxCap
0	50%
1	50%
2	40%

PARTID	MaxBW	MinBW
0	4800MBps	0MBps
1	1200MBps	1000MBps
2	600MBps	600MBps

Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

TrustZone solution

TrustZone processors have two security states

- Separate MMU configurations
- Registers protection

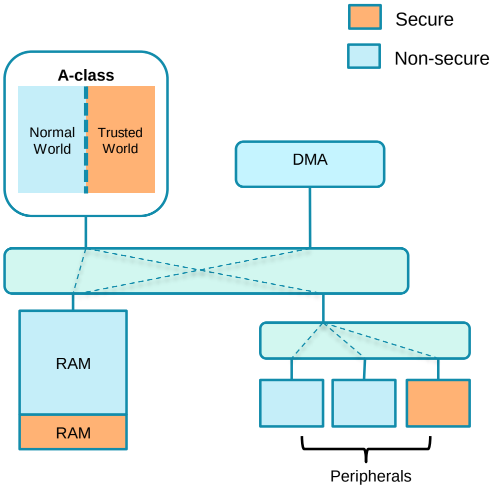
Memory system is security aware

- Addresses have a security state on the bus
- Enforced by hardware

Peripheral access can be arbitrated by security state

- Trusted code can have access to IO peripherals

Memory system can be implemented to allow dynamic partitioning

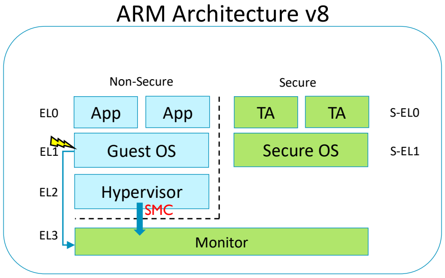


🟧 Secure
🟦 Non-secure

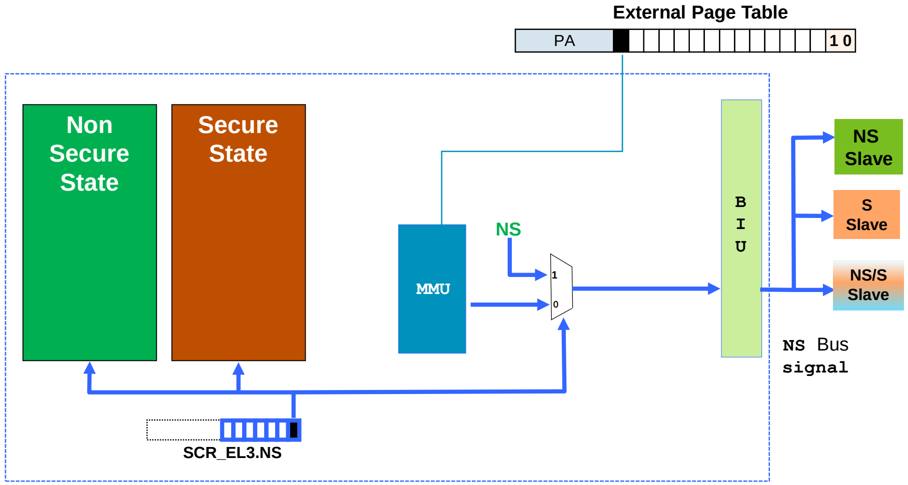
Arm Architecture Security Extensions

Arm security extension introduces,

- Two secure state: Secure and non secure state.
- EL3 is always secure, and EL3 is the gateway between secure and non secure state.
  - EL3 has its own MMU address translation.
  - SMC instruction for secure monitor call
- Memory security
  - CPU in non secure state never can issue secure access to memory subsystem
  - CPU in secure state can issue secure or non secure access to memory subsystem, depending on the memory attribute set in MMU translation table.
  - Cache and TLB entries are flagged with secure state
- Exceptions could be routed to EL3

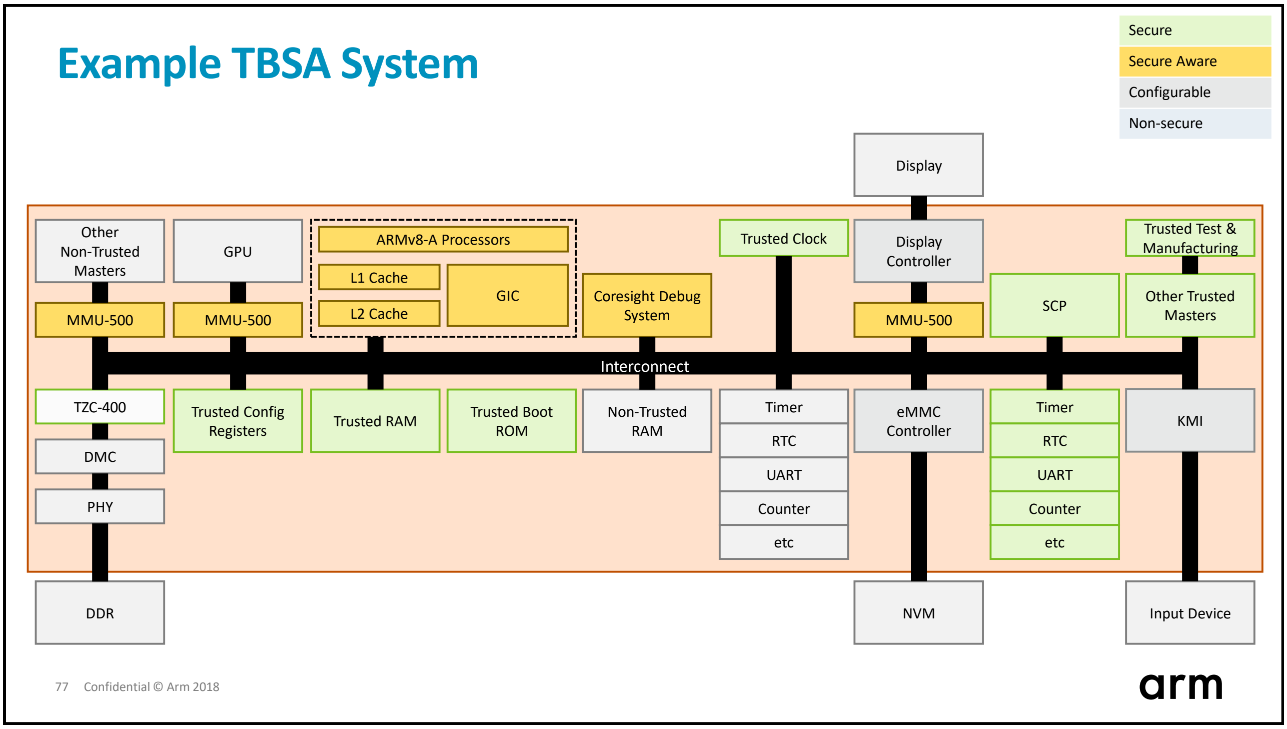


Memory access secure state indication



Secure state can access both secure and non-secure physical memory

- Controlled by the secure translation tables

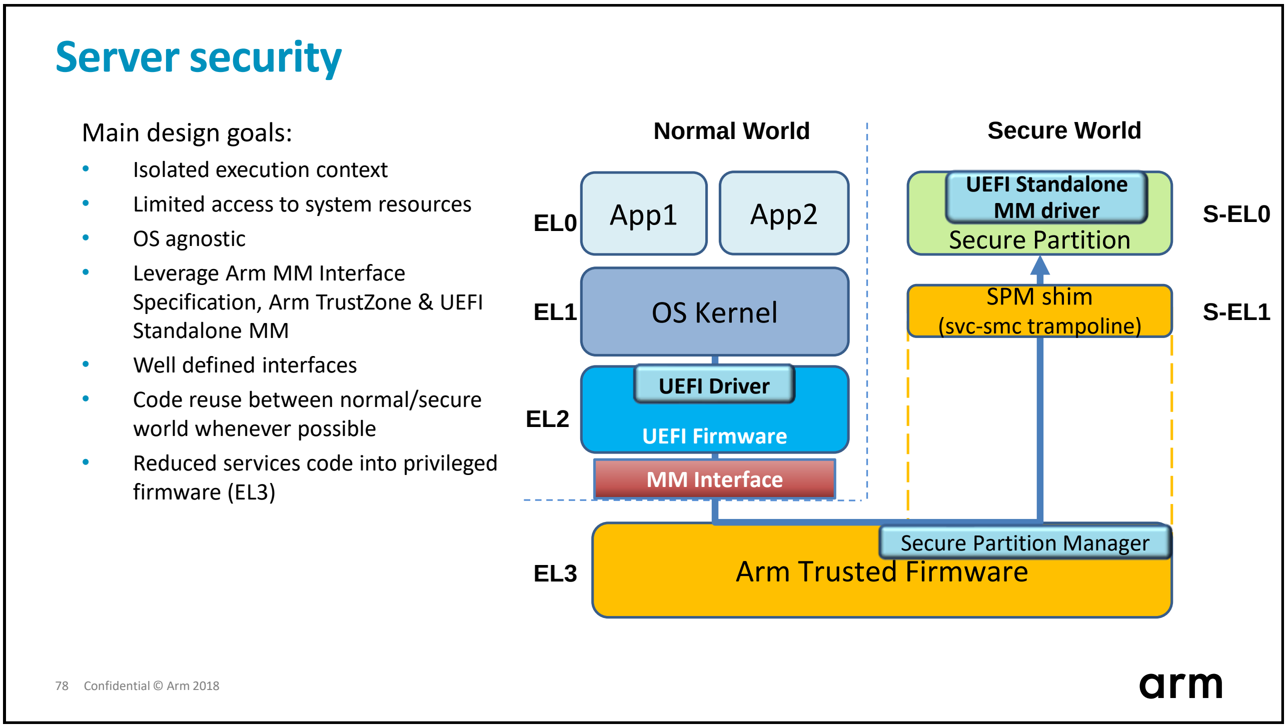


Example TBSA System

Secure
Secure Aware
Configurable
Non-secure

Display
Other Non-Trusted Masters
GPU
ARMv8-A Processors
L1 Cache
L2 Cache
GIC
Coresight Debug System
Trusted Clock
Display Controller
Trusted Test & Manufacturing
SCP
Other Trusted Masters
MMU-500
MMU-500
MMU-500
Interconnect
TZC-400
DMC
PHY
Trusted Config Registers
Trusted RAM
Trusted Boot ROM
Non-Trusted RAM
Timer
RTC
UART
Counter
etc
eMMC Controller
Timer
RTC
UART
Counter
etc
KMI
DDR
NVM
Input Device

arm



Server security

Main design goals:

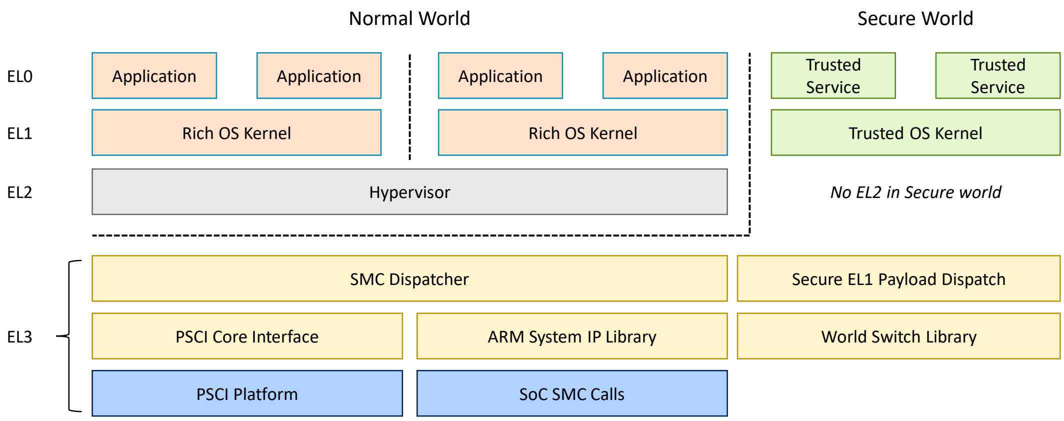
- Isolated execution context
- Limited access to system resources
- OS agnostic
- Leverage Arm MM Interface Specification, Arm TrustZone & UEFI Standalone MM
- Well defined interfaces
- Code reuse between normal/secure world whenever possible
- Reduced services code into privileged firmware (EL3)

Normal World
Secure World
EL0
App1
App2
UEFI Standalone MM driver
Secure Partition
S-EL0
EL1
OS Kernel
SPM shim (svc-smc trampoline)
S-EL1
EL2
UEFI Driver
UEFI Firmware
MM Interface
EL3
Secure Partition Manager
Arm Trusted Firmware

arm

ARM Trusted Firmware architecture

🟨 ARM Trusted Firmware
🟦 SoC Specific Firmware



79 Confidential © Arm 2018

arm

Agenda

Armv8.x overview
Registers
ISA
SVE
Exception
Memory management
Memory model
Atomic
Architecture timer
Virtualization
MPAM
Security
RAS

80 Confidential © Arm 2018

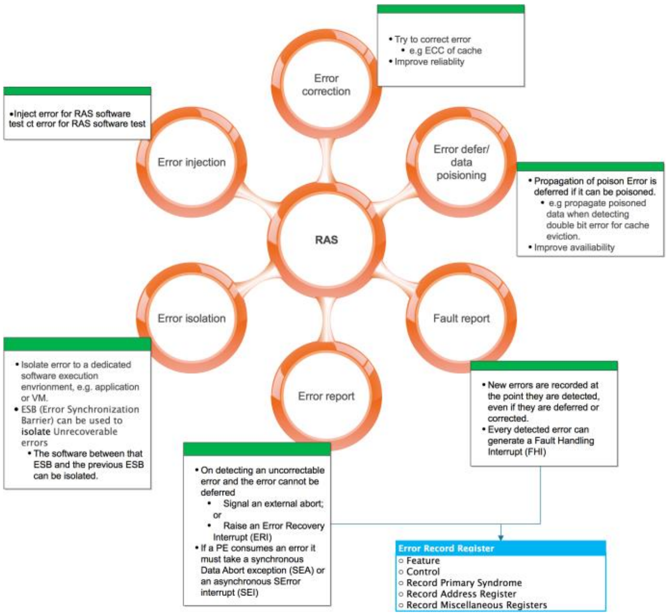
Armv8.2 RAS Extensions

RAS is required for ARMv8.2, it stands for:

- Reliability - continuity of correct service
- Availability - readiness for correct service
- Serviceability - ability to undergo modifications and repairs

RAS extensions provides framework for building RAS features around

- It describes error types, fault/error reporting and handling, and unrecoverable error isolation etc.
- Does not specify RAS features in implementations



- Try to correct error
  - e.g ECC of cache
- Improve reliablity

- Propagation of poison Error is deferred if it can be poisoned.
  - e.g propagate poisoned data when detecting double bit error for cache eviction.
- Improve availiability

- New errors are recorded at the point they are detected, even if they are deferred or corrected.
- Every detected error can generate a Fault Handling Interrupt (FHI)

- On detecting an uncorrectable error and the error cannot be deferred
  - Signal an external abort; or
  - Raise an Error Recovery Interrupt (ERI)
- If a PE consumes an error it must take a synchronous Data Abort exception (SEA) or an asynchronous SError interrupt (SEI)

- Isolate error to a dedicated software execution envrionment, e.g. application or VM.
- ESB (Error Synchronization Barrier) can be used to isolate Unrecoverable errors
  - The software between that ESB and the previous ESB can be isolated.

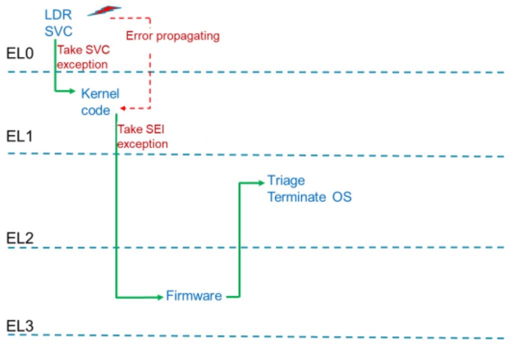
- Inject error for RAS software test ct error for RAS software test

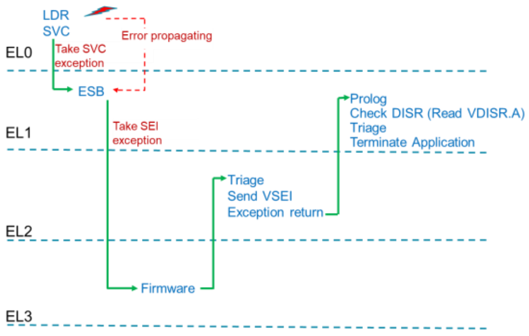
Error Record Register
- ○ Feature
- ○ Control
- ○ Record Primary Syndrome
- ○ Record Address Register
- ○ Record Miscellaneous Registers

Armv8.2 RAS Extensions

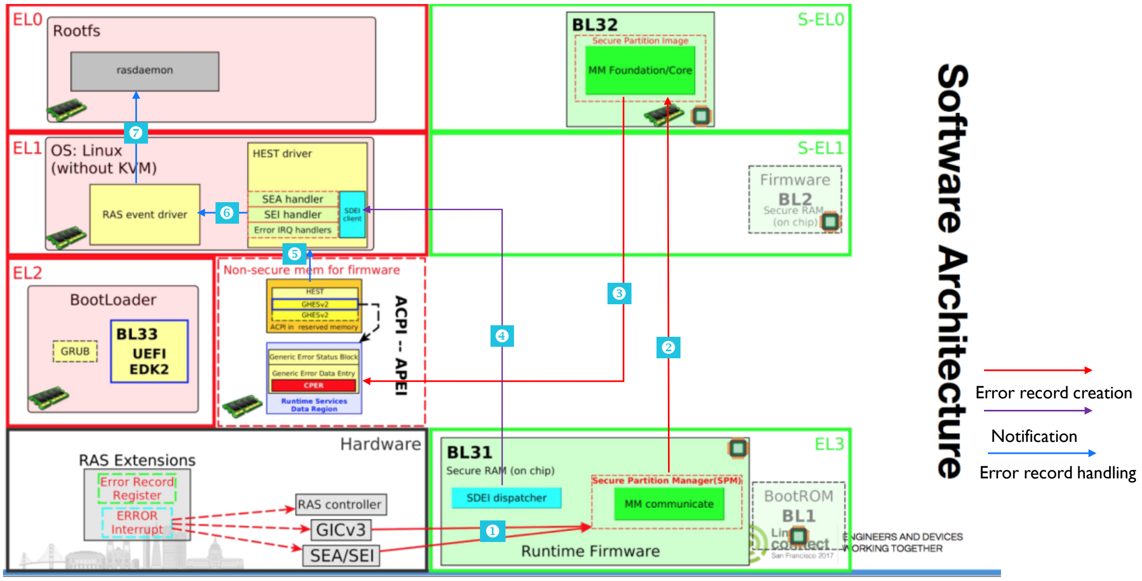
ESB is for error isolation between ELs

- It guarantees that all Unrecoverable errors generated before the ESB have pended an SEI exception.

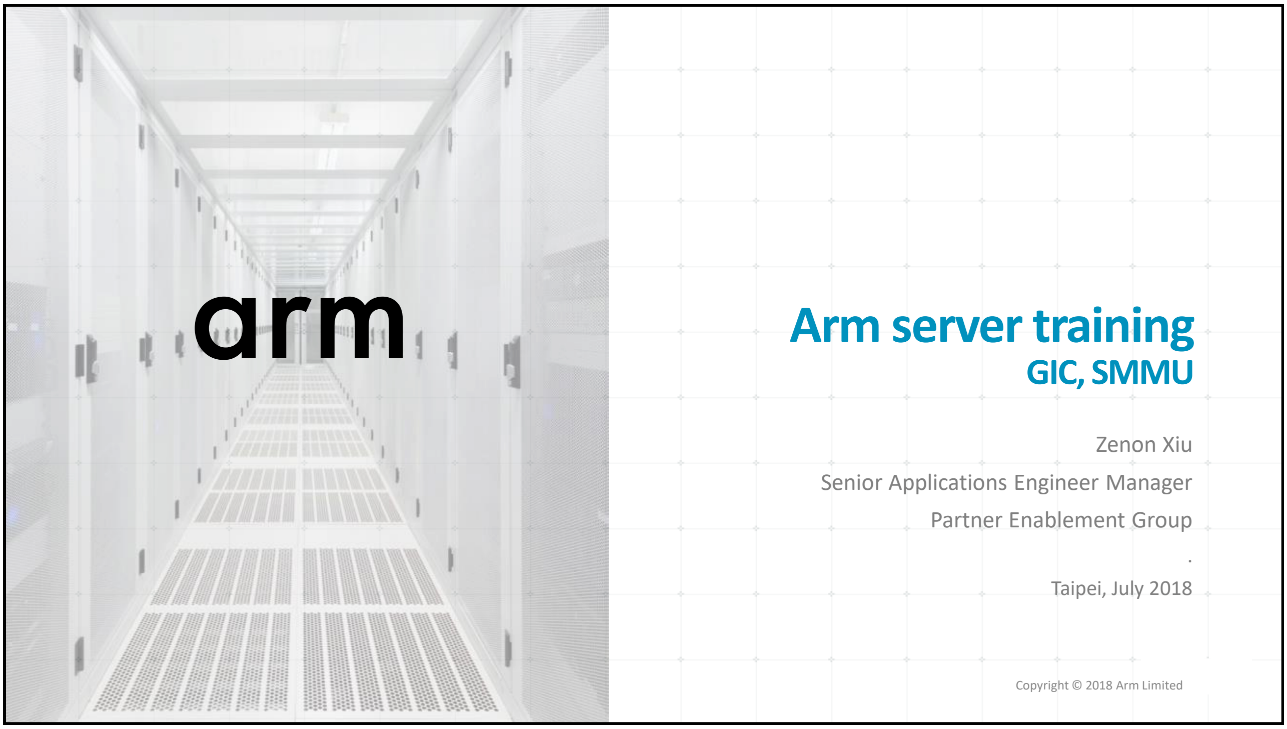




RAS software



Source: Linaro Connect SFO17



Arm server training

GIC, SMMU

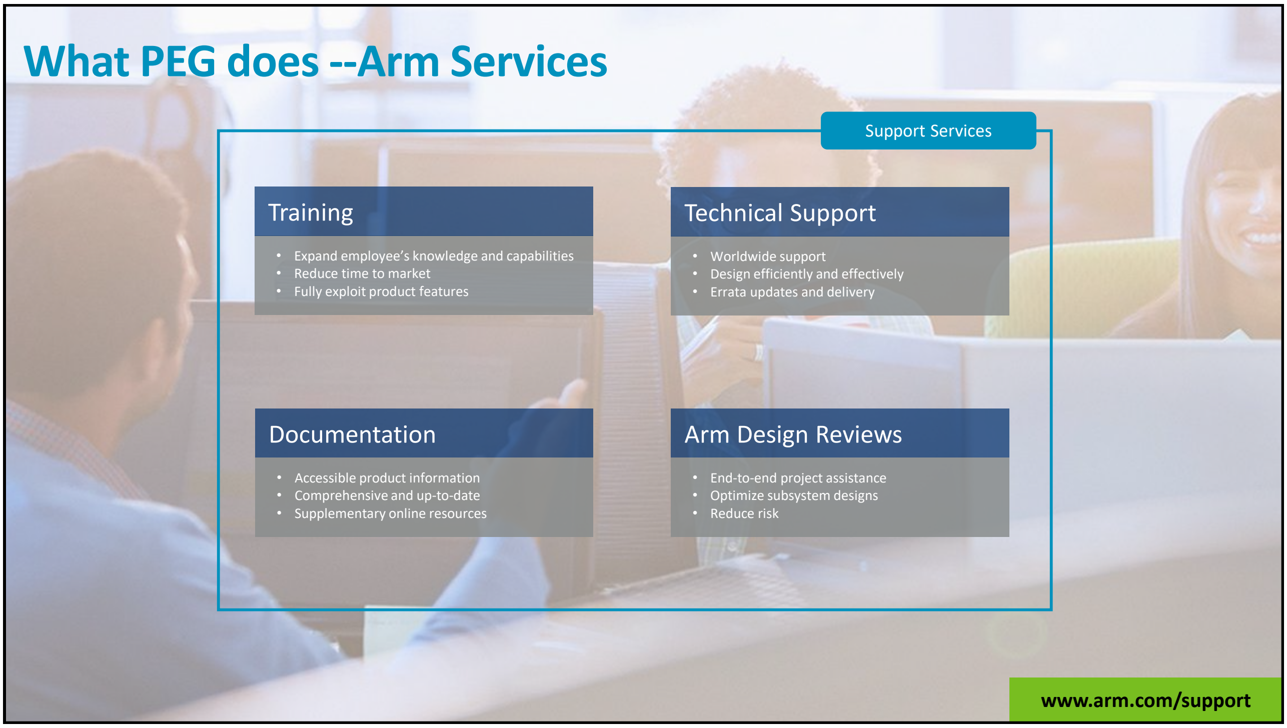
Zenon Xiu

Senior Applications Engineer Manager

Partner Enablement Group

.

Taipei, July 2018



What PEG does --Arm Services

Support Services

Training

- Expand employee's knowledge and capabilities
- Reduce time to market
- Fully exploit product features

Technical Support

- Worldwide support
- Design efficiently and effectively
- Errata updates and delivery

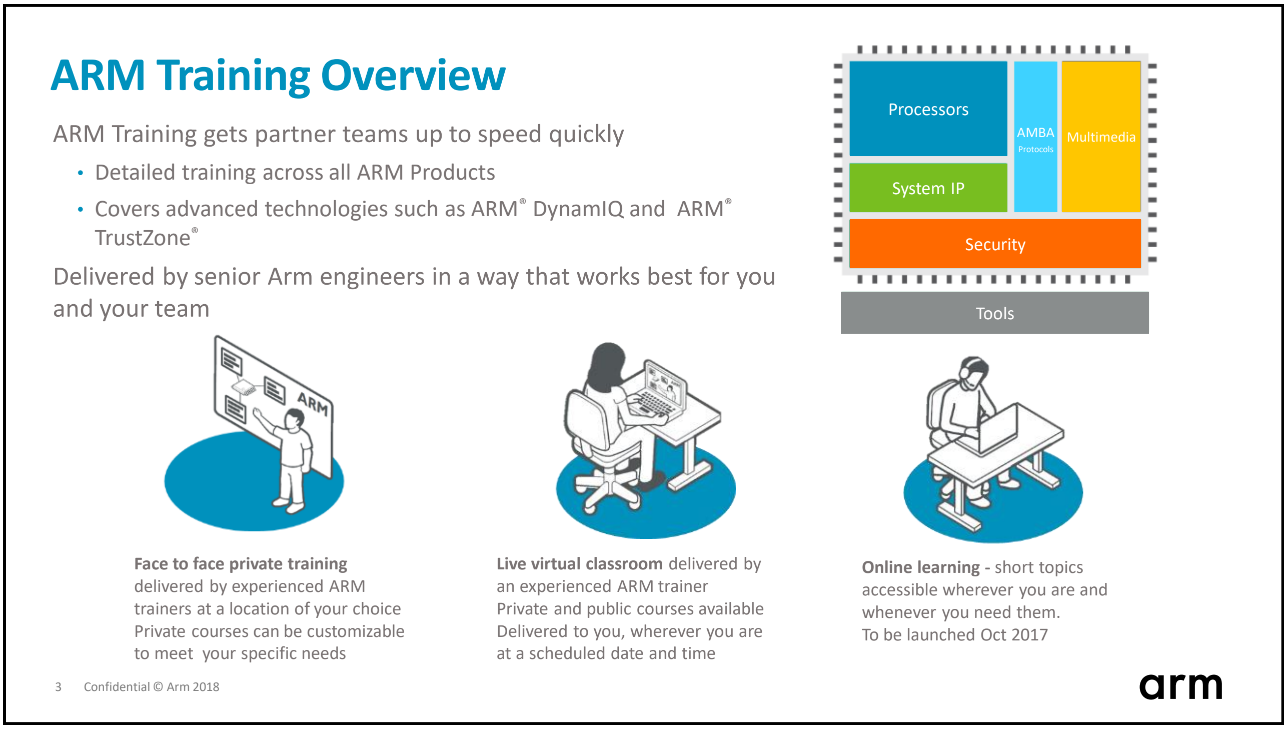
Documentation

- Accessible product information
- Comprehensive and up-to-date
- Supplementary online resources

Arm Design Reviews

- End-to-end project assistance
- Optimize subsystem designs
- Reduce risk

www.arm.com/support



ARM Training Overview

ARM Training gets partner teams up to speed quickly

- Detailed training across all ARM Products
- Covers advanced technologies such as ARM® DynamIQ and ARM® TrustZone®

Delivered by senior Arm engineers in a way that works best for you and your team

Processors
AMBA
Protocols
Multimedia
System IP
Security
Tools
ARM

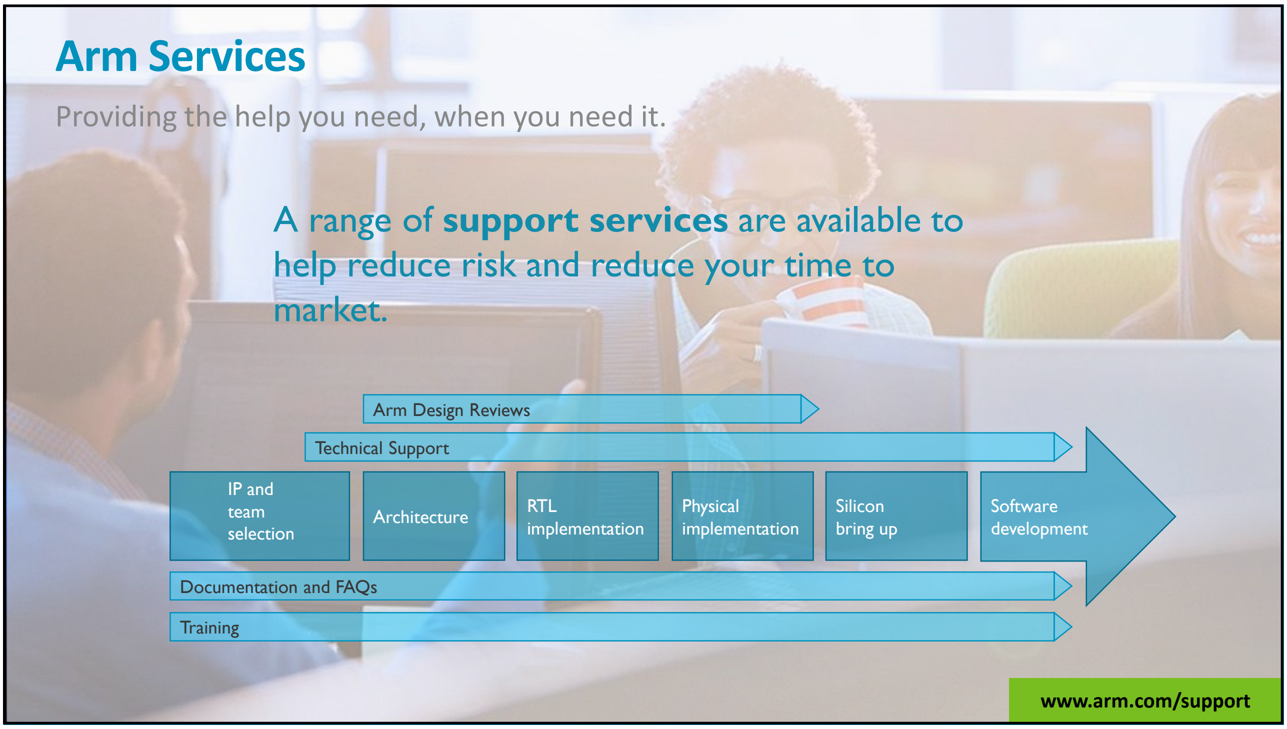
Face to face private training delivered by experienced ARM trainers at a location of your choice
Private courses can be customizable to meet your specific needs

Live virtual classroom delivered by an experienced ARM trainer
Private and public courses available
Delivered to you, wherever you are at a scheduled date and time

Online learning - short topics accessible wherever you are and whenever you need them.
To be launched Oct 2017

3 Confidential © Arm 2018

arm



Arm Services

Providing the help you need, when you need it.

A range of **support services** are available to help reduce risk and reduce your time to market.

Arm Design Reviews

Technical Support

IP and team selection

Architecture

RTL implementation

Physical implementation

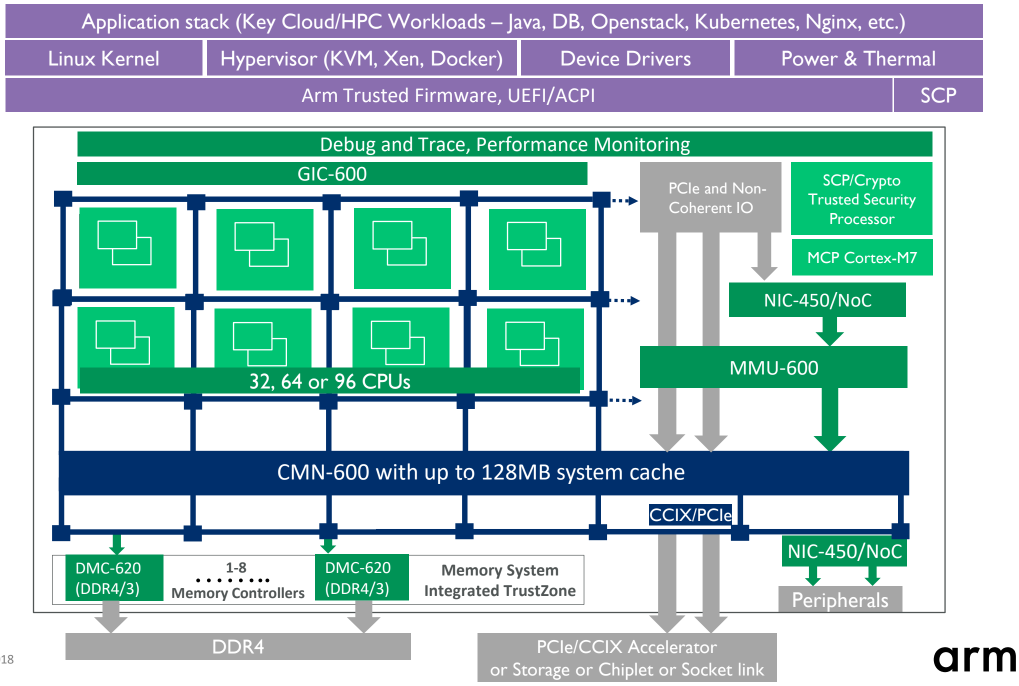
Silicon bring up

Software development

Documentation and FAQs

Training

www.arm.com/support



Application stack (Key Cloud/HPC Workloads – Java, DB, Openstack, Kubernetes, Nginx, etc.)

Linux Kernel | Hypervisor (KVM, Xen, Docker) | Device Drivers | Power & Thermal

Arm Trusted Firmware, UEFI/ACPI | SCP

Debug and Trace, Performance Monitoring

GIC-600

PCIe and Non-Coherent IO

SCP/Crypto Trusted Security Processor

MCP Cortex-M7

NIC-450/NoC

MMU-600

32, 64 or 96 CPUs

CMN-600 with up to 128MB system cache

CCIX/PCIe

NIC-450/NoC

DMC-620 (DDR4/3)

1-8 Memory Controllers

DMC-620 (DDR4/3)

Memory System Integrated TrustZone

Peripherals

DDR4

PCIe/CCIX Accelerator or Storage or Chiplet or Socket link

arm

Agenda

GICv3/v4

SMMUv3

GIC versions

GICv2

Adds:

- Support for virtualization
- Improved handling of Group 1 interrupts by secure software

Implemented by:

- Cortex-A15 MPCore*
- Cortex-A7 MPCore*
- CoreLink GIC-400

* Inclusion of GIC is optional

GICv3

Adds:

- Support for more than 8 cores
- Support for message signalled interrupts
- System Register access to some registers
- Vastly expanded the Interrupt ID space

Implemented by:

- CoreLink GIC-500
- CoreLink GIC-600

GICv4

Adds:

- Direct injection of virtual interrupts

GICv3/v4 overview

0 .. 15: Software Generated Interrupts (SGIs)

- Used for software inter-processor interrupts
- Generated by software to a set of target processors

16 .. 31: Peripheral Interrupts (PPIs)

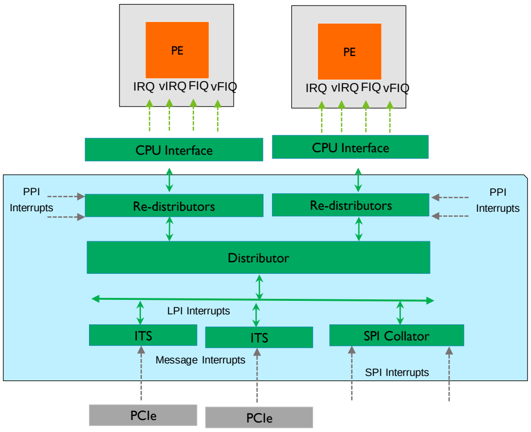
- Banked per processor
- Used for interrupts from peripheral private to a processor

32 .. 991(configurable): Shared Peripheral Interrupts (SPIs)

- Global interrupts that that can be sent to any one processor
  - Either “1 of all” processors or a targeted processor

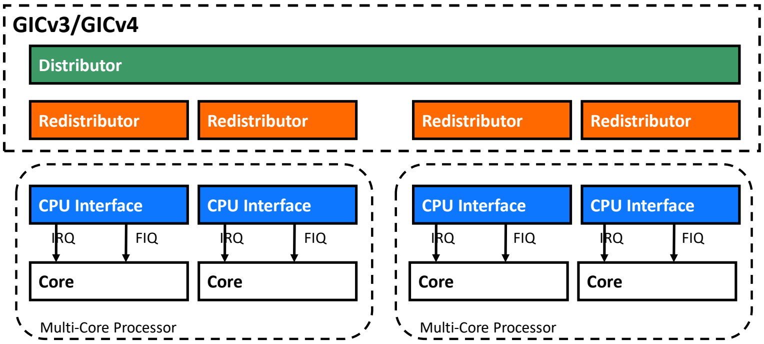
8192+: Locality-Specific Interrupts (LPIs) (*New in GICv3)*

- Interrupts that can be sent to any one targeted processor



Redistributors

GICv3 introduces Redistributors



There is a Redistributor per-connected core, holding settings for private interrupts (PPIs and SGIs)

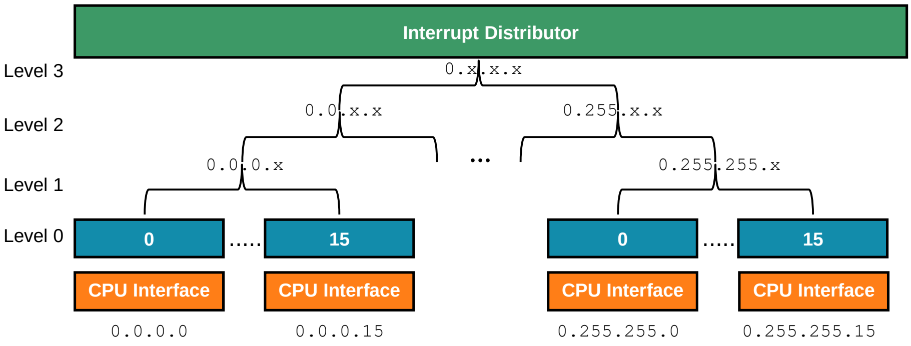
- Allows for distributed designs with Redistributors kept close to target core

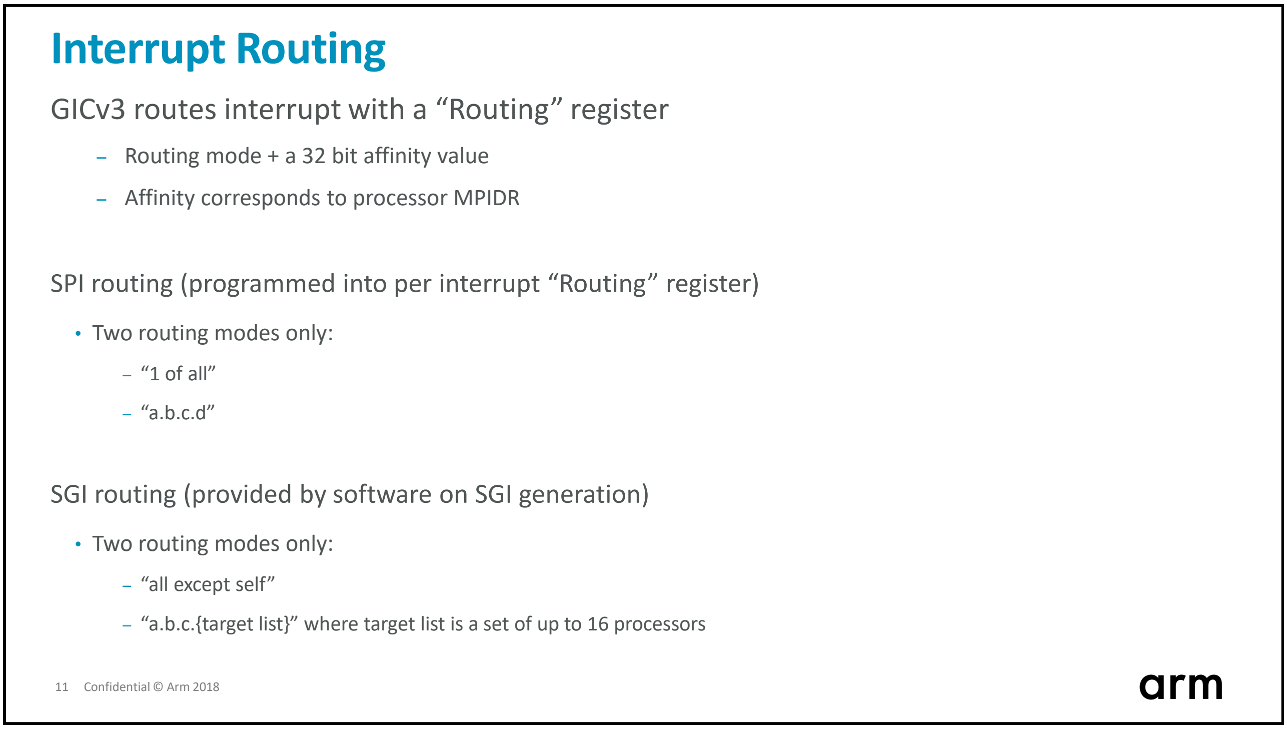
Affinity levels and routing

There is increasingly demand for systems with higher core counts

GICv3 increases the number of cores that can be connected

- Connected cores identified by an affinity value, matching system used in ARMv8-A





Interrupt Routing

GICv3 routes interrupt with a “Routing” register

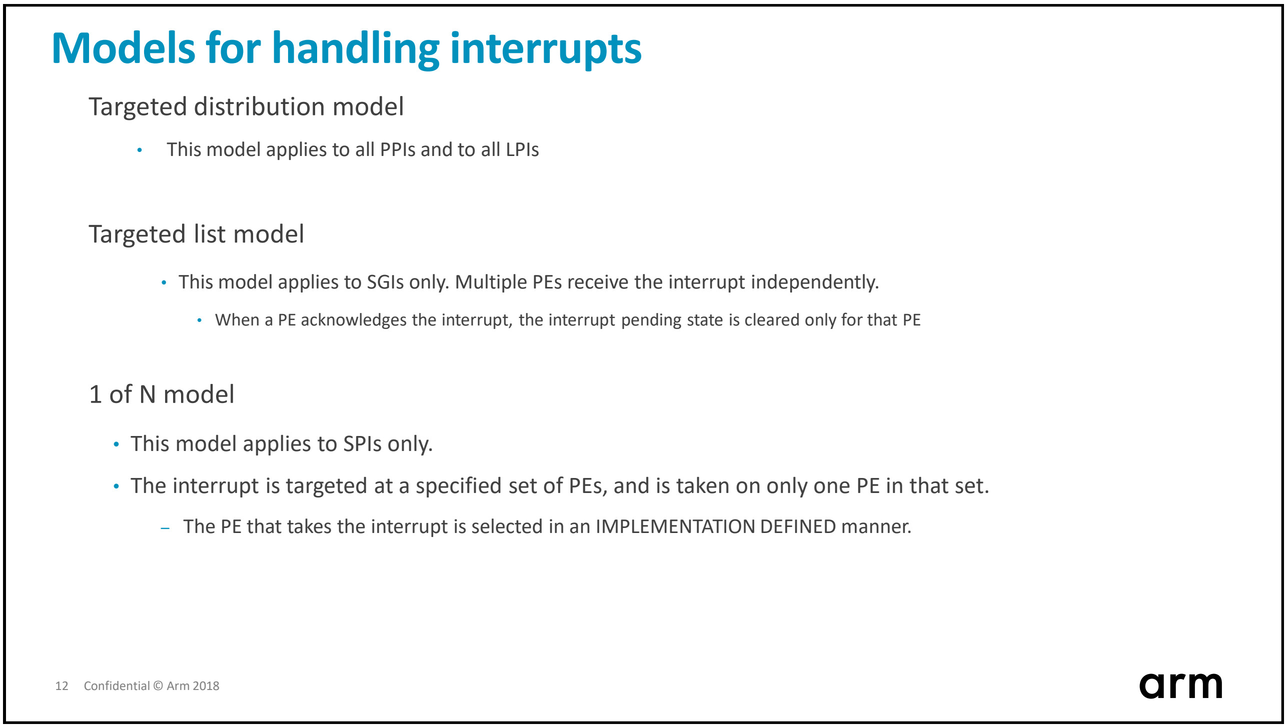
- Routing mode + a 32 bit affinity value
- Affinity corresponds to processor MPIDR

SPI routing (programmed into per interrupt “Routing” register)

- Two routing modes only:
  - “1 of all”
  - “a.b.c.d”

SGI routing (provided by software on SGI generation)

- Two routing modes only:
  - “all except self”
  - “a.b.c.{target list}” where target list is a set of up to 16 processors



Models for handling interrupts

Targeted distribution model

- This model applies to all PPIs and to all LPIs

Targeted list model

- This model applies to SGIs only. Multiple PEs receive the interrupt independently.
  - When a PE acknowledges the interrupt, the interrupt pending state is cleared only for that PE

1 of N model

- This model applies to SPIs only.
- The interrupt is targeted at a specified set of PEs, and is taken on only one PE in that set.
  - The PE that takes the interrupt is selected in an IMPLEMENTATION DEFINED manner.

SPI, SGI and PPI configuration

Enable

- Only enabled interrupts can be forwarded to a core
- Disabled interrupts can still become pending

Priority

- Each interrupt has an 8-bit priority associated with it

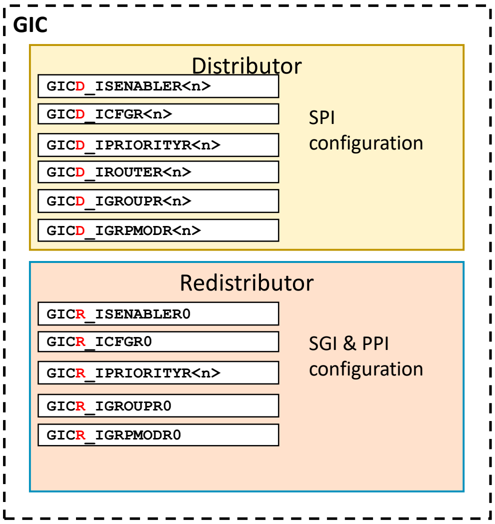
Configuration

- Whether the interrupt is level-sensitive or edge-triggered
- SGIs are always edge-triggered

Routing (SPIs only)

- Interrupt can target one named core, or any core

Security/Group



GIC

Distributor

- GICD_ISENABLER<n>
- GICD_ICFGR<n>
- GICD_IPRIORITYR<n>
- GICD_IROUTER<n>
- GICD_IGROUPR<n>
- GICD_IGRPMODR<n>

SPI configuration

Redistributor

- GICR_ISENABLER0
- GICR_ICFGR0
- GICR_IPRIORITYR<n>
- GICR_IGROUPR0
- GICR_IGRPMODR0

SGI & PPI configuration

Introduction to LPIs

LPIs use very large global ID spaces

- State for these ID spaces held in external memory

Physical LPIs

- Each interrupt routed to a single processor at any time
- Configuration table in memory (priorities and enables)
- Per-processor pending table in memory

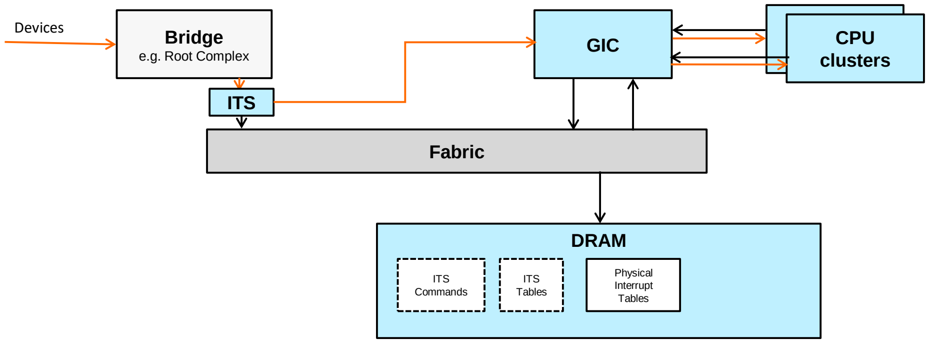
On GIC-500 & GIC-600, only generated via write to the Interrupt Translation Service

Typically an MSI(-X)

ITS

Message-based interrupts from devices are sent to Interrupt Translation Service in GIC

- Supports standard MSI(-X) interrupts
- Memory-backed to support large number of interrupts
- Cached within GIC to reduce latency and power for common interrupts
- Provides ID translation, allowing devices to be programmed by a VM

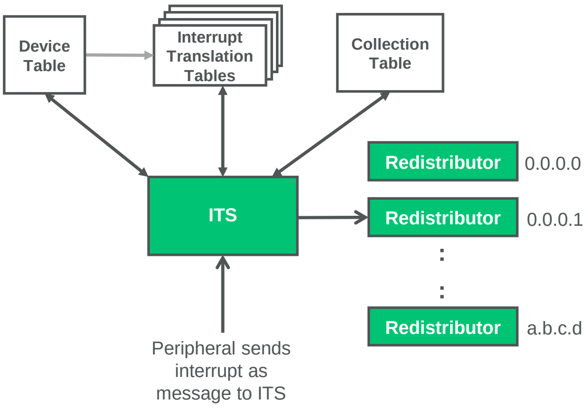


How ITS works

An Interrupt Translation Service (or ITS) maps interrupts to INTIDs and Redistributors

How is an interrupt translated?

- Peripheral sends interrupt as a message to the ITS
  - The message specifies the DeviceID (which peripheral) and an EventID (which interrupt from that peripheral)
- ITS uses the DeviceID to index into the Device Table
  - Returns pointer to a peripheral specific Interrupt Translation Table
- ITS uses the EventID to index into the Interrupt Translation Table
  - Returns the INTID and Collection ID
- ITS uses the Collection ID to index into the Collection Table
  - Returns the target Redistributor
- ITS forwards interrupt to Redistributor



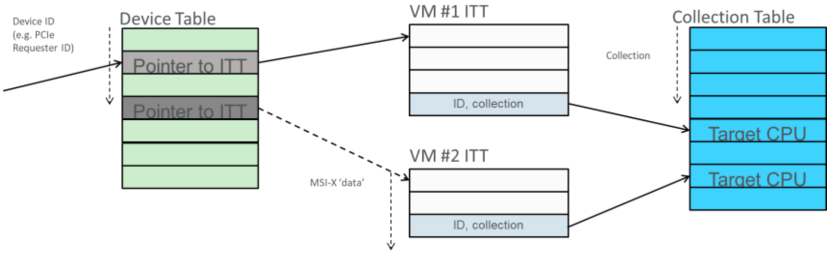
ITS - Interrupt Isolation and Translation

Each “device” has a single owner

- Hypervisor and Guest operating system

Each “device” indexes a “Device table” using “Device ID” to select an Interrupt Translation Table (ITT)

- PCIe root complex must drive Requester ID on AW channel e.g. from SMMU



Protection

- Each “device” only be able to generate own interrupts

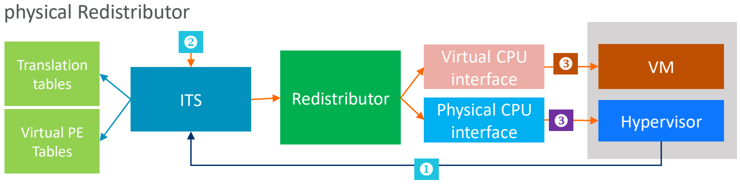
Translation

- Each “device” can have different mapping: MSI-X ‘data’ => (target CPU, interrupt ID)

GICv4

GIC V4 supports the direct injection of virtual interrupts,

- Reduces the hypervisor mediation overhead.
- Map an EventID-DeviceID combination to a vINTID and pINITD(door-bell interrupt) for a specific vPE
  - Optionally a door-bell interrupt can be specified. This is a pINTID that is generated if the vPE is not scheduled when the interrupt is generated.
- Map a vPE to a physical Redistributor

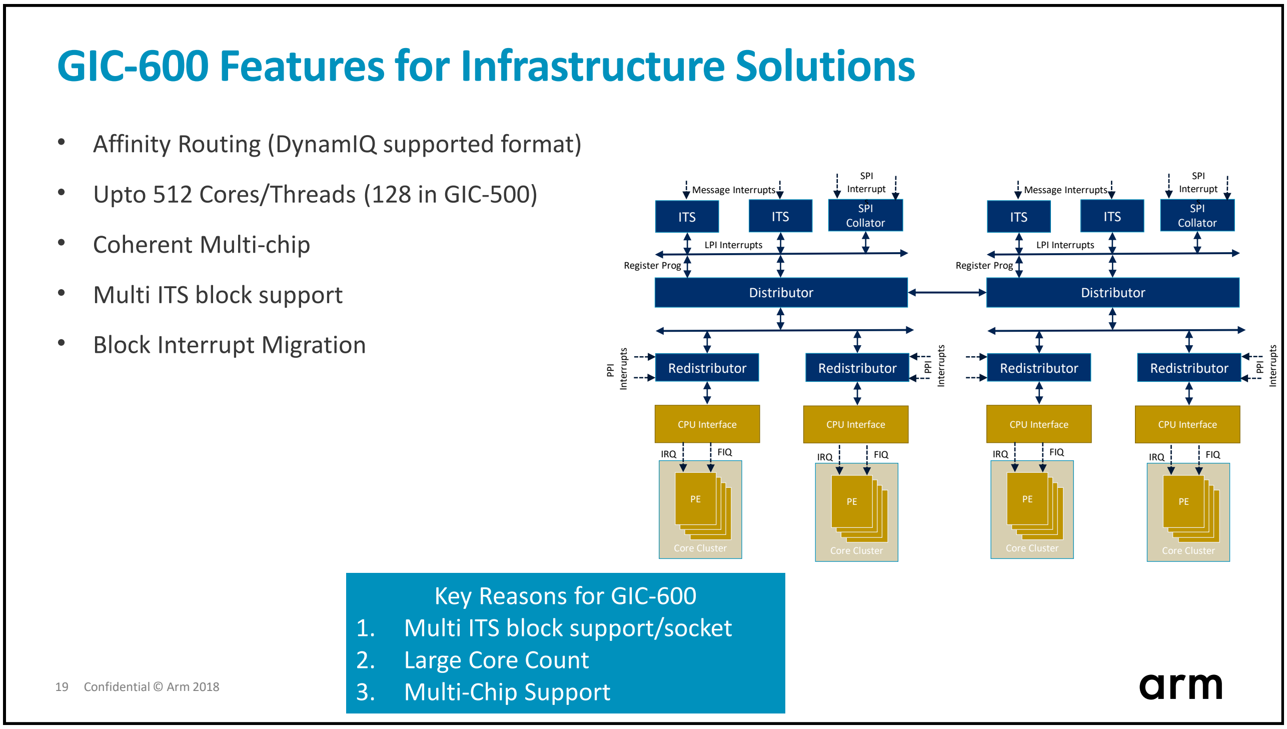


❶ Hypervisor programs ITS to map EventID/DeviceID to virtual INTID and virtual CPU

❷ ITS uses Eventide-DeviceID to translation the MSI, and route the interrupt

❸ If VM is scheduled: Redistributor forwards virtual interrupt vINTID to virtual CPU interface

❸ If VM is not scheduled: Redistributor forwards door-bell interrupt pINTID to physical CPU interface



GIC-600 Features for Infrastructure Solutions

- Affinity Routing (DynamIQ supported format)
- Upto 512 Cores/Threads (128 in GIC-500)
- Coherent Multi-chip
- Multi ITS block support
- Block Interrupt Migration

Key Reasons for GIC-600

1. Multi ITS block support/socket
2. Large Core Count
3. Multi-Chip Support

Agenda

GICv3/v4

SMMUv3

SMMU

System MMU (or SMMU) provides memory management for DMA masters

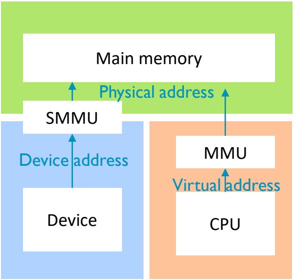
- CPU MMU only used for the processor cores
- Sometimes referred to as IOMMU

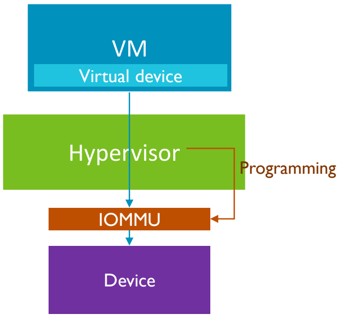
Provides memory translation

- Enables large regions of memory to be allocated without being physically contiguous
- Masters with small addresses (e.g 32-bit) can address all physical memory (e.g 40-bit)

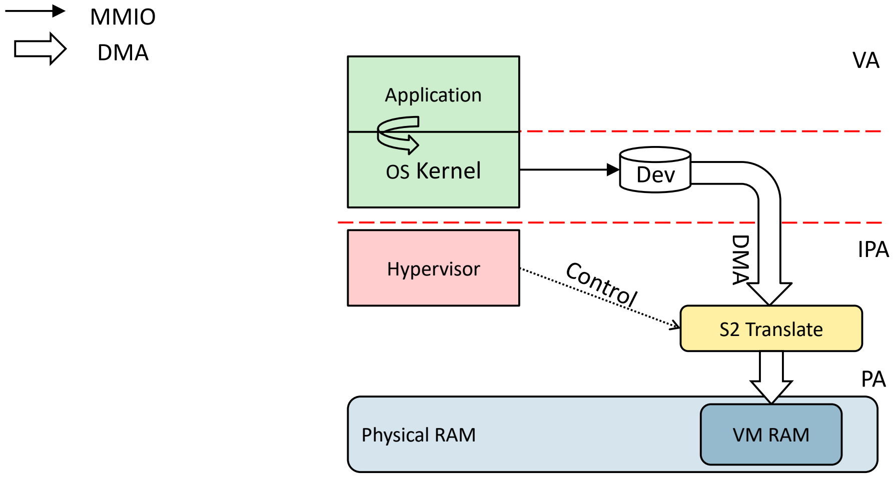
Provides isolation and protection from accessing illegal memory locations

- From process to process or from VM guest to guest
- When using virtualization masters & software do not need to be aware of Hypervisor

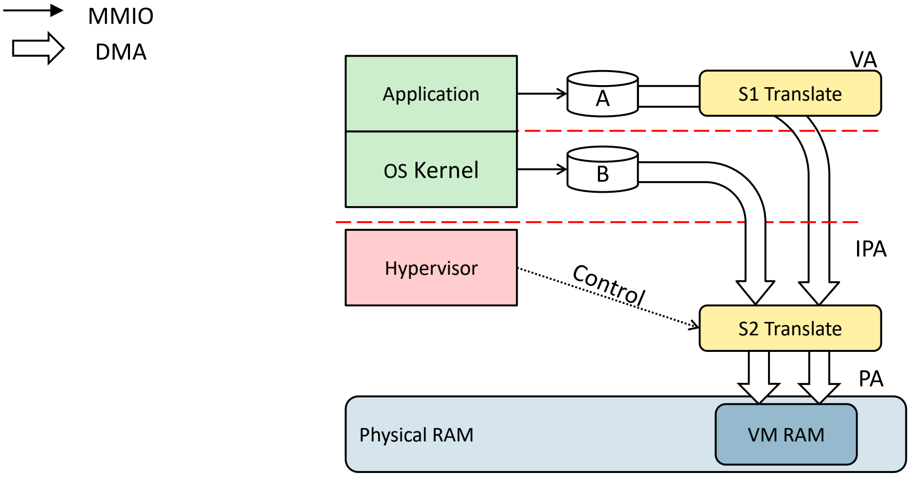




SMMU example usage 1



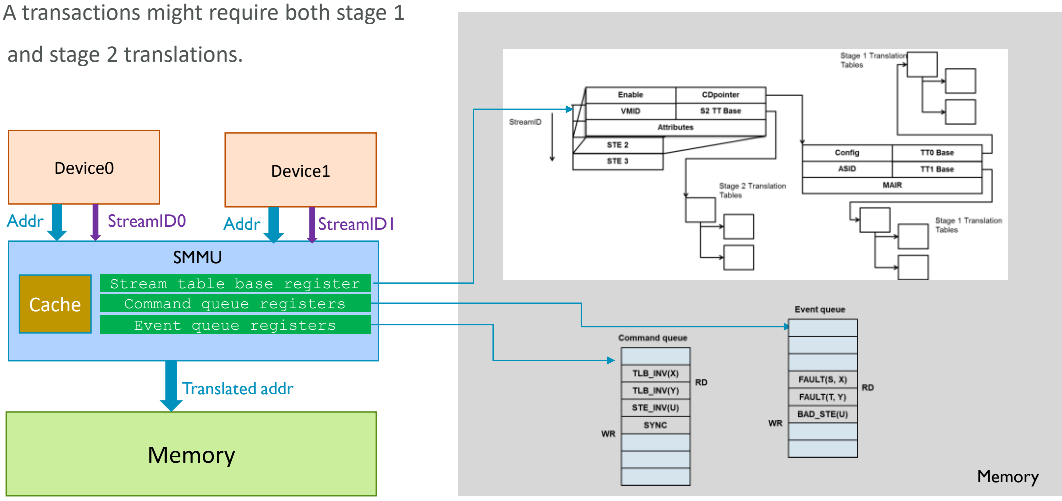
SMMU example usage 2



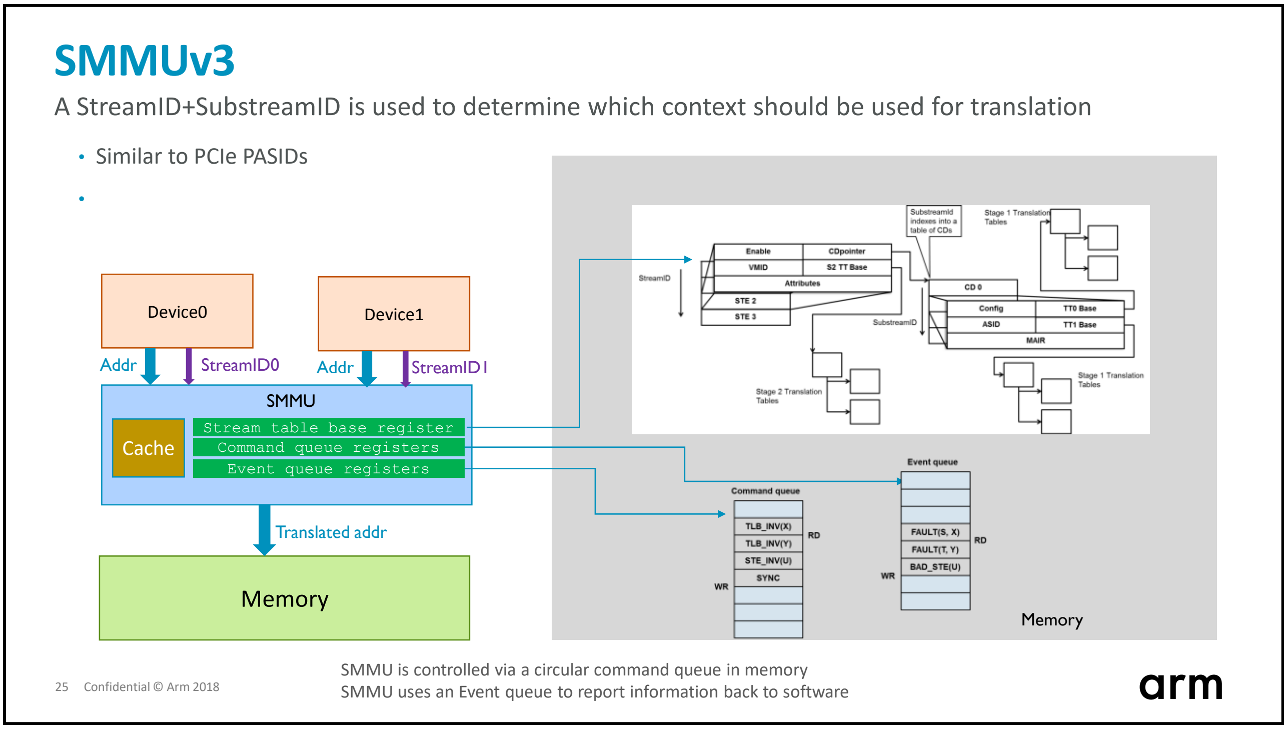
SMMUv3

A StreamID is used to determine which context should be used for translation

- A transactions might require both stage 1 and stage 2 translations.



SMMU is controlled via a circular command queue in memory
SMMU uses an Event queue to report information back to software

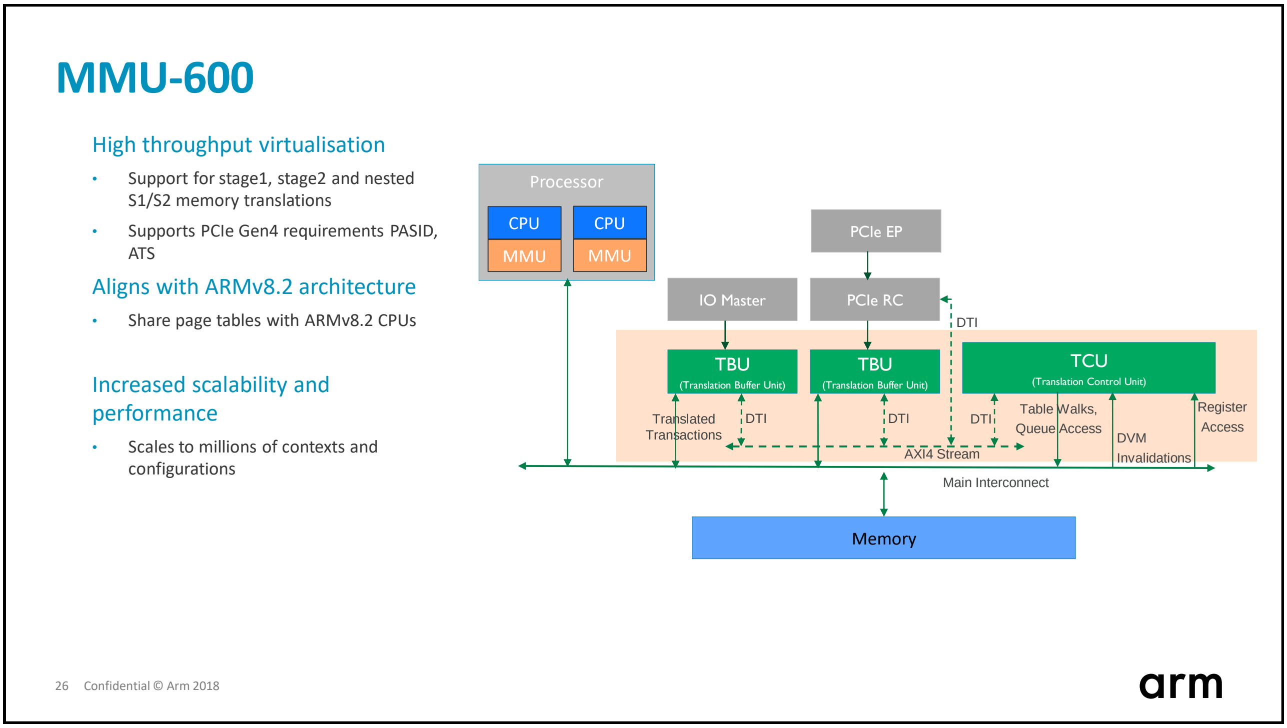


SMMUv3

A StreamID+SubstreamID is used to determine which context should be used for translation

- Similar to PCIe PASIDs
-

SMMU is controlled via a circular command queue in memory
SMMU uses an Event queue to report information back to software



MMU-600

High throughput virtualisation

- Support for stage1, stage2 and nested S1/S2 memory translations
- Supports PCIe Gen4 requirements PASID, ATS

Aligns with ARMv8.2 architecture

- Share page tables with ARMv8.2 CPUs

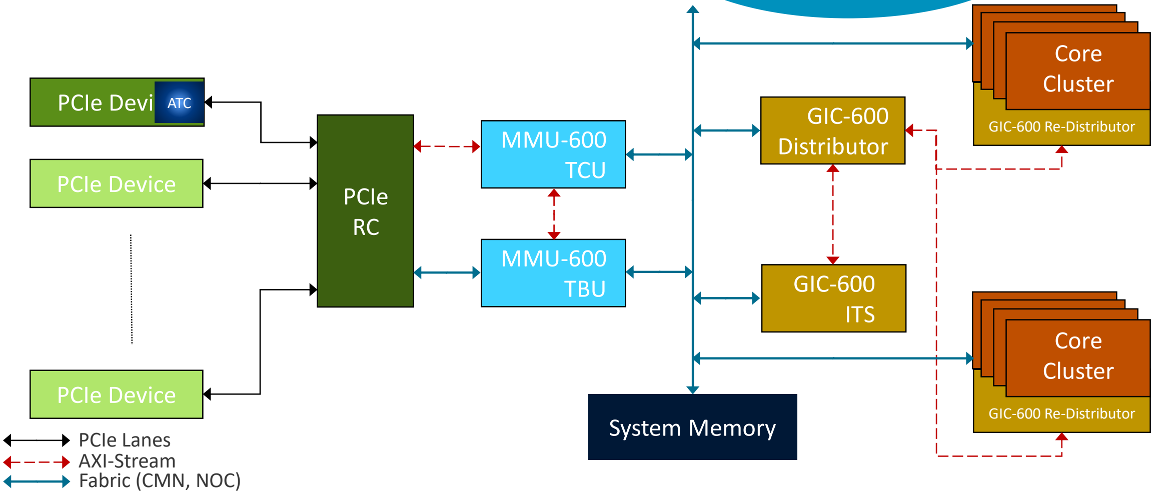
Increased scalability and performance

- Scales to millions of contexts and configurations

PCIe Integration in Arm Systems

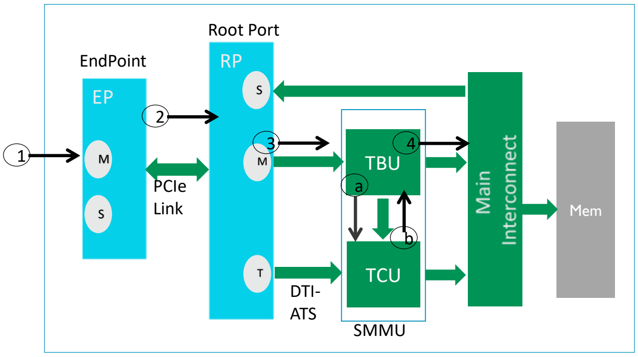
Key for PCIe Gen4:

1. ATS flow
2. PRI flow



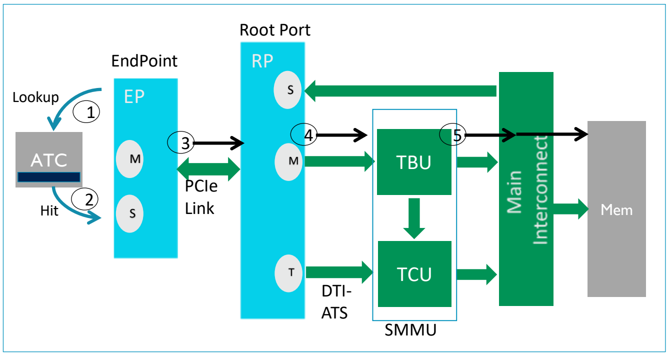
SMMU with PCIe operation – no ATS

1. Client logic generates a TLP with a un-translated address
2. EP sends this as a PCIe TLP to the RP
3. On receipt by the RP, since the packet is a data flow packet, this is sent on the "M" interface
   a) If the TBU does not have a suitable translation for the address received, it will issue a request to TCU
   b) The TCU will respond with the response for the TBU
4. The TBU then forwards the transaction to the memory



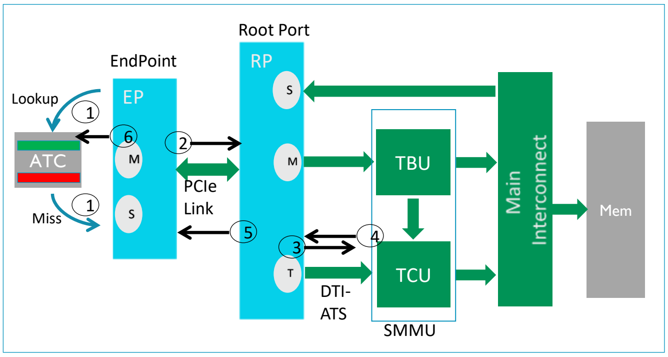
SMMU with PCIe operation – with ATS and ATC hit

1. Client logic generates a TLP with a virtual address
2. The client logic uses the translated Addr if available from the ATC
3. The EP sends this as a PCIe TLP that has translated address
4. On receipt by the RP, since the packet is a data-flow packet, this is sent on the "M" interface
5. The TBU then forwards the transaction to the memory via the main interconnect



SMMU with PCIe operation – with ATS and ATC miss

1. EP client generates a PCIe translation for a particular address that needs translation
2. Translation request goes out on the PCIe link to the RP
3. RP sends the translation request it received on the "T" interface to the TCU
4. The TCU then generates the response completion
5. The RP repacks the translation completion TLP back to the EP
6. Once the EP received this completion for the translation request it generated, it populates the local ATC



arm