FTF 2016
TECHNOLOGY FORUM

# PORTING SOFTWARE FROM POWER ARCHITECTURE® TO ARM®

## FTF-DES-N1841

BHUPESH SHARMA
TEAM LEAD, UEFI FIRMWARE, DIGITAL NETWORKING
FTF-DES-N1841
MAY 16, 2016

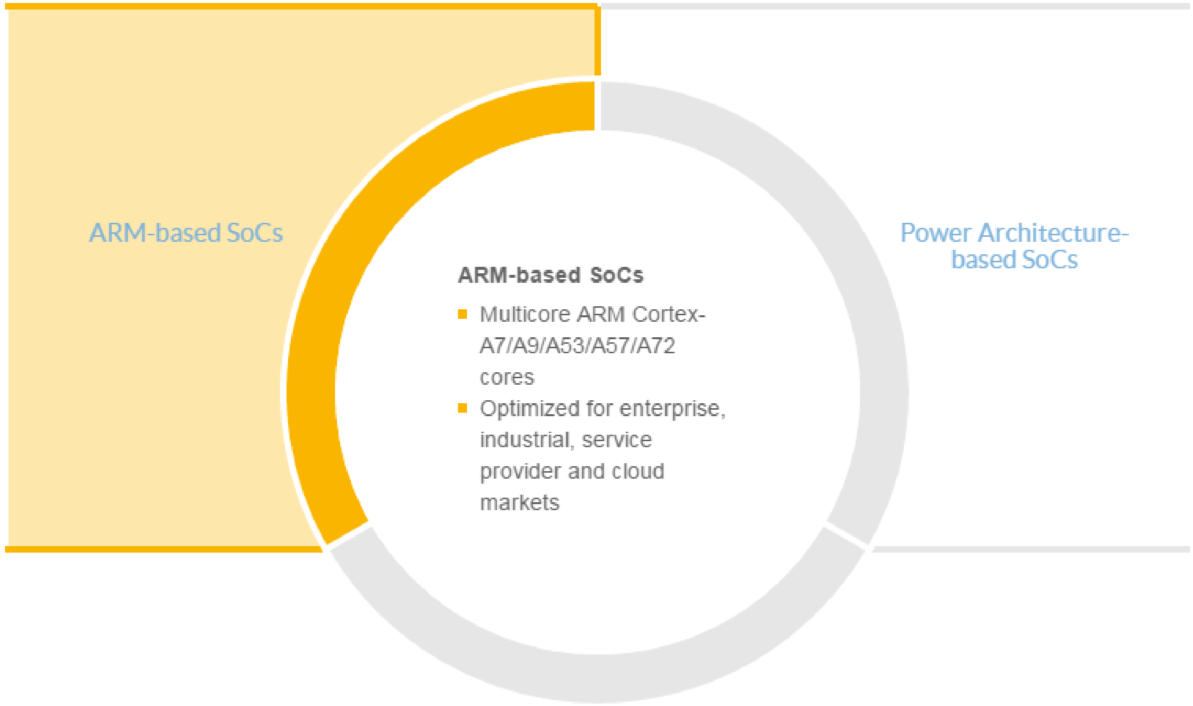
# AGENDA

- Overview of QorIQ Processors
- Comparison of environments: Power® vs ARM®
  - 32-bit v/s 64-bit
  - Instruction Set Architecture (ISA)
  - IP Ecosystem
  - Endianness
- ARM fundamentals for good software design
  - Memory Model
  - Exception Model
  - Security Model
- Software Support
- Summary

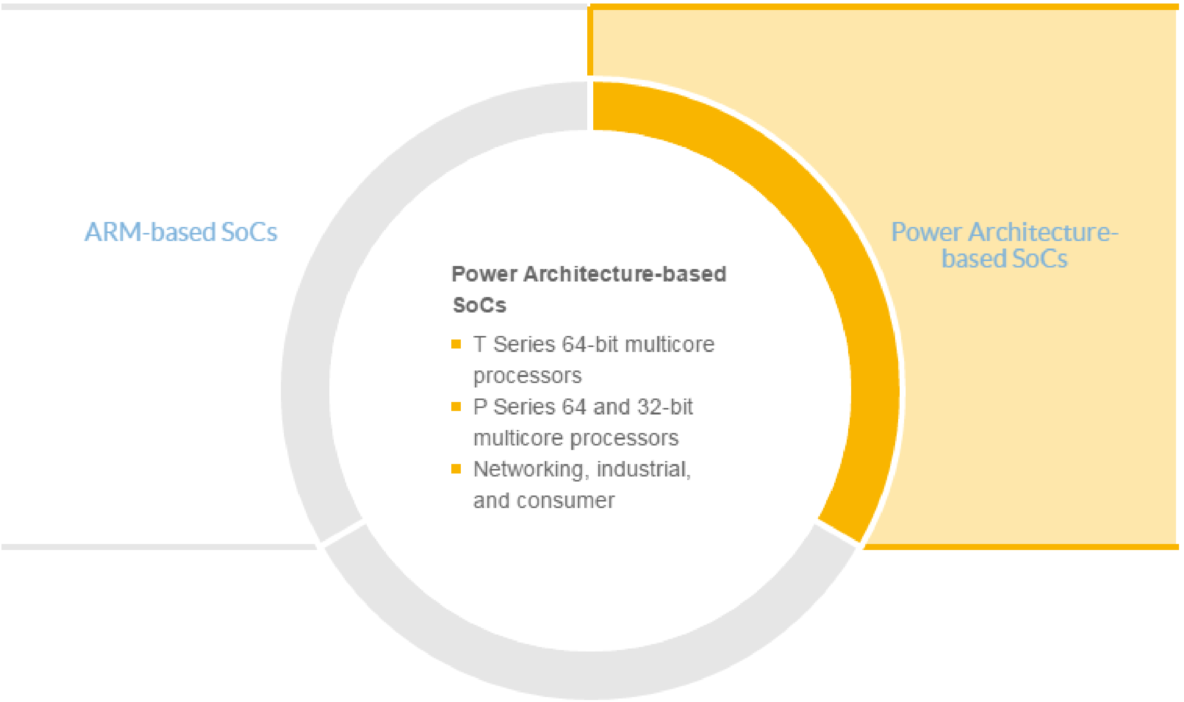
# OVERVIEW OF QORIQ PROCESSORS

# Overview of QorIQ Processors

QorIQ Processing Platforms: 64-bit Multicore SoCs




QorIQ Processing Platforms: 64-bit Multicore SoCs



# Power Architecture Technology - Core Overview

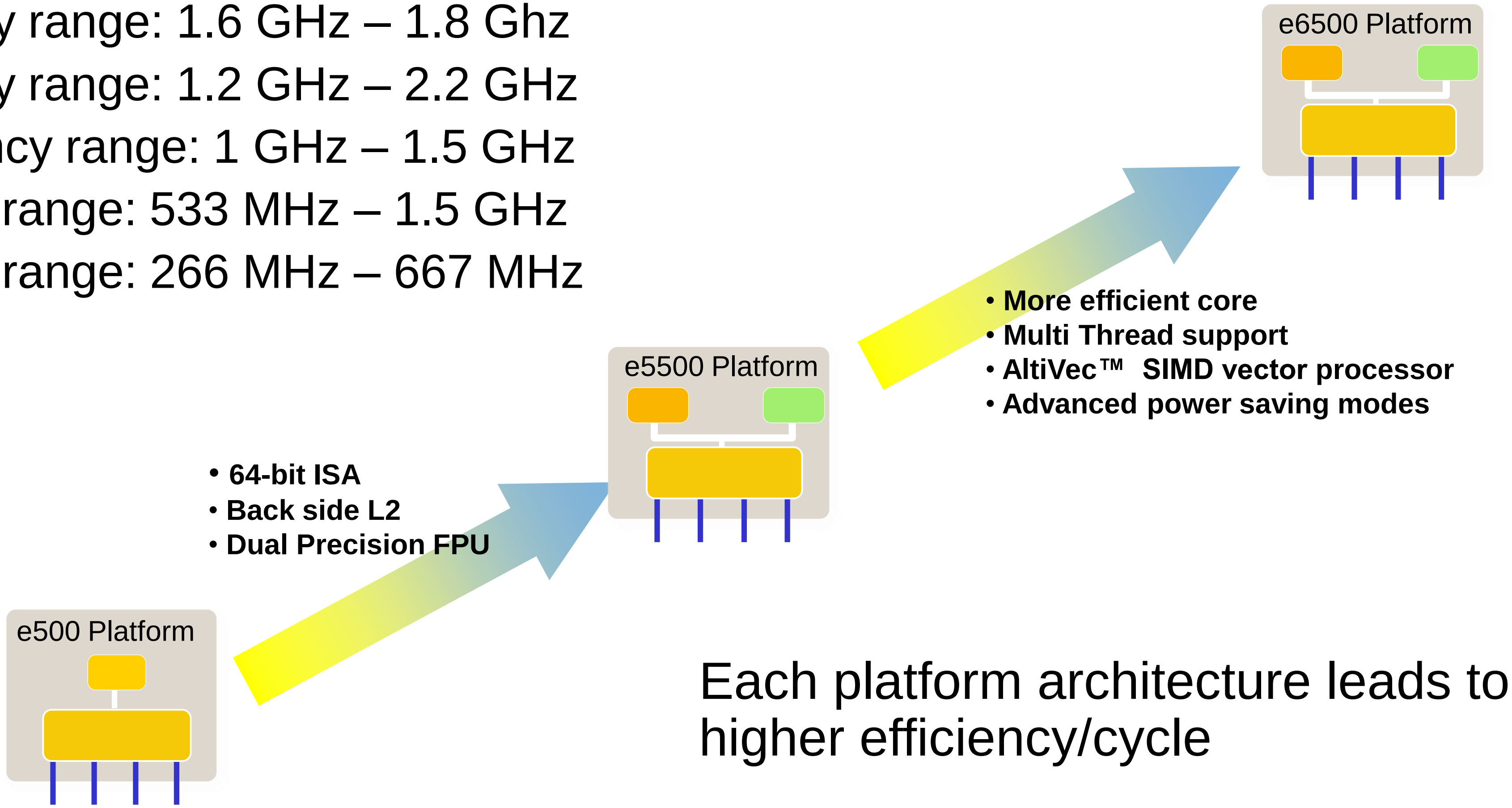
| Core | ISA / MHz | Platforms |
|---|---|---|
| e5500 Core<br>e6500 Core | Power Architecture ISA<br>1.2 GHz – 2.2Ghz | **e5500 Platforms**<br>QorIQ P5, T1 Platforms<br>**e6500 Platforms**<br>QorIQ T2, B4, T4 Platforms |
| e600 Core | Power Architecture ISA<br>600 MHz – 1.8 GHz | **e600 Platforms**<br>MPC86xx host processors<br>MPC74xx host processors |
| e500 Core<br>e500-mc Core | Power Architecture ISA<br>533 MHz – 1.5 GHz | **e500 Platforms**<br>PowerQUICC III MPC85xx<br>QorIQ P1 & P2020 Platforms<br>**e500-mc Platforms**<br>QorIQ P204x, P3 and P4 Platforms |
| e300 Core | Power Architecture ISA<br>266 MHz – 667 MHz | **e300 Platforms**<br>PowerQUICC II Pro MPC83xx<br>PowerQUICC II MPC82xx<br>MPC52xx microcontrollers<br>MPC51xx microcontrollers |
| e200 Core | Power Architecture<br>80 MHz – 475 MHz | **e200 Platforms**<br>MPC55xx auto microcontrollers |

# e300 ↔ e500 ↔ e500mc ↔ e5500 ↔ e6500 Cores

## Relative Optimized Performance

Frequency overlap allows for incremental performance boosts as required

- e6500 frequency range: 1.6 GHz – 1.8 Ghz
- e5500 frequency range: 1.2 GHz – 2.2 GHz
- e500mc frequency range: 1 GHz – 1.5 GHz
- e500 frequency range: 533 MHz – 1.5 GHz
- e300 frequency range: 266 MHz – 667 MHz




Each platform architecture leads to higher efficiency/cycle

# ARM Cores Device Family

| ARM Processor Family | | ARM Architecture | Core |
|---|---|---|---|
| Classic ARM | ARM7 | ARMv3 | ARM700 |
| | | | ARM710 |
| | | | ARM710a |
| | ARM11 | ARMv6 | ARM1136J |
| | | | ARM1156T2 |
| | | | ARM1176JZ |
| | | | ARM11MPCore |
| Embedded | Cortex-M | ARMv6-M | Cortex-M0 |
| | | | Cortex-M1 |
| | | ARMv7-M | CortexM3 |
| | | ARMv7E-M | Cortex-M4 |
| Real-Time | Cortex-R | ARMv7-R | Cortex-R4 |
| | | | Cortex-R5 |
| | | | Cortex-R7 |
| Application | Cortex-A | ARMv7-A | Cortex-A5 |
| | | | Cortex-A7 |
| | | | Cortex-A8 |
| | | | Cortex-A9 |
| | | | Cortex-A15 |
| 64-bit Core | Cortex-A5x | ARMv8-A | Cortex-A53 |
| | | | Cortex-A57 |
| | | | Cortex-A72 |

# QorIQ SoCs and Core Architectures

| Power Cores | QorIQ SoCs | | ARM Cores | QorIQ SoCs |
|---|---|---|---|---|
| e500 | • P1010<br>• BSC913x | 32bit arch | ARMv7 | • LS1021A |
| e5500 | • T1040<br>• P5040 | 64bit arch | ARMv8 (DPAA1) | • LS1043A |
| e6500 | • B4860<br>• T4240<br>• B5T10 | | ARMv8 (DPAA2) | • LS2080A<br>• LS1088A* |

*** These devices are under development**

# COMPARISON OF ENVIRONMENTS: POWER® VS ARM® - 32 BIT V/S 64-BIT

# Data Size and Instruction Sets

- **Both Power Architecture and ARM are based on RISC architecture**
  - Fixed instruction set length
  - Most instructions execute in a single cycle
  - Superscalar dual issue core with out-of-order execution and in-order completion
  - ARM: Every instruction can be conditionally executed

# Power Architecture: 32bit vs 64bit – SW Programmers View

## 32-bit Power machines

Supports 32bit execution mode
- 32-bit effective address
- 32-bit registers
- Instructions to manipulate 32-bit address and 32-bit registers

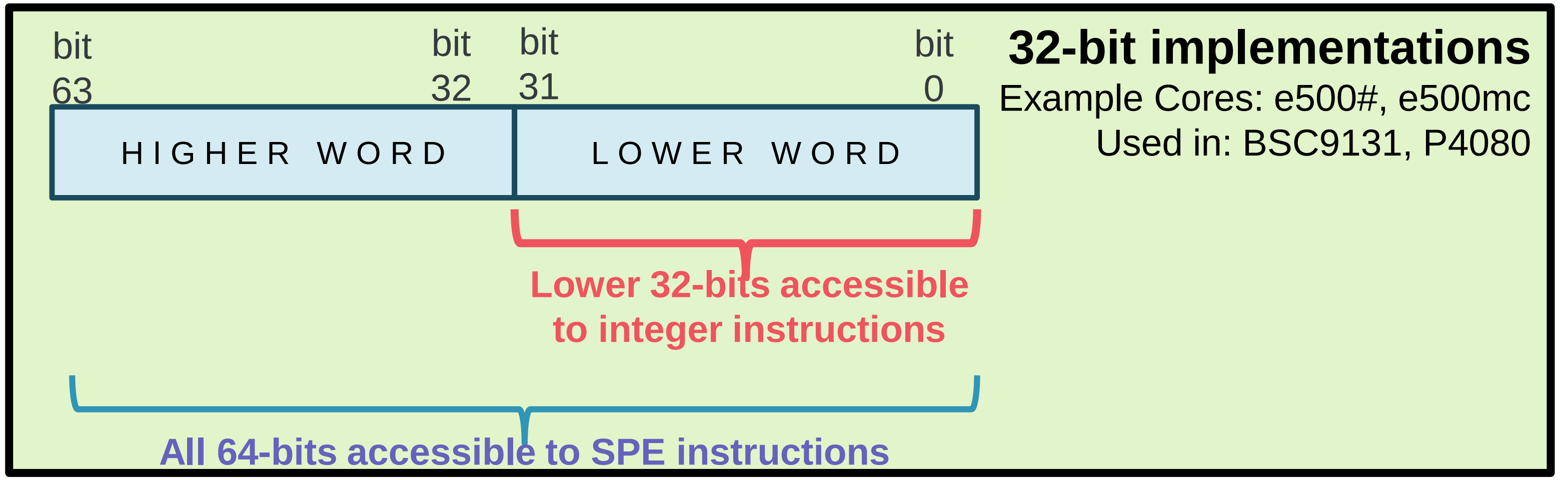
U-boot, Linux and apps execute in 32bit mode

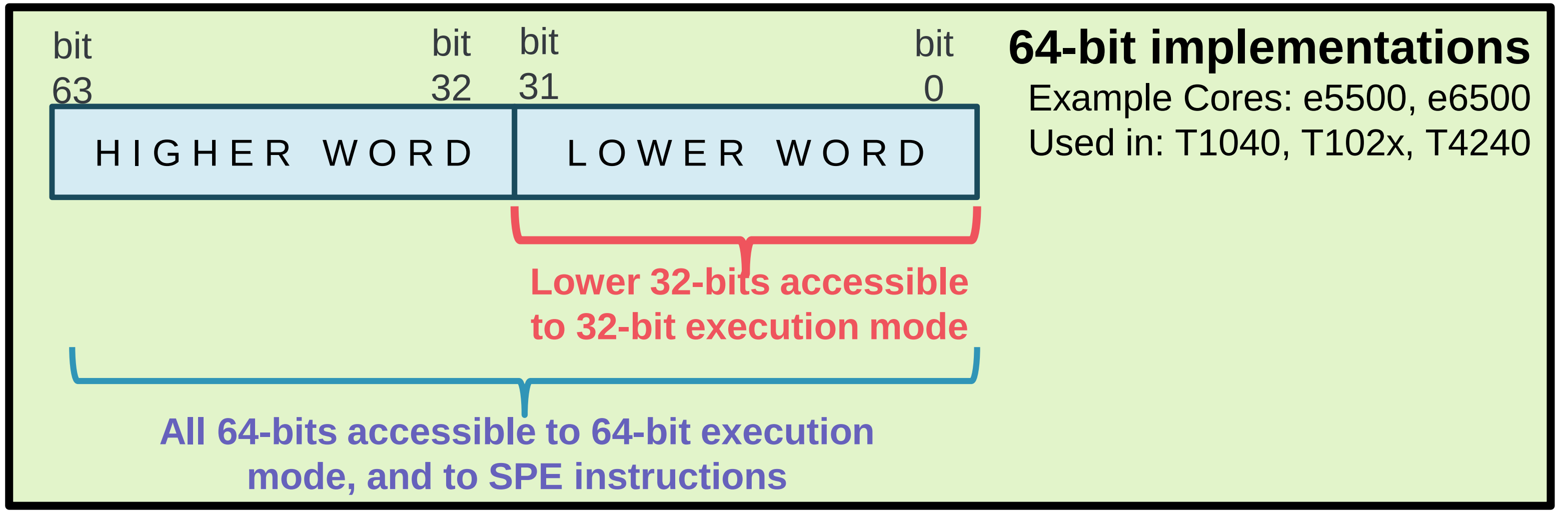
## 64-bit Power machines

Supports 32bit and 64bit execution modes
- 32-bit execution mode is like 32-bit implementation (shown on the left)
- 64-bit execution mode:
  - ~ 64-bit effective address
  - ~ 64-bit registers
  - ~ Instructions to manipulate 64-bit address and 64-bit registers

U-boot executes in 32bit mode
Linux executes in 64bit mode
Apps: 32b and 64b

# Power Architecture Registers: General Purpose Registers



# Power Architecture: Types of Registers

- **GPR:**
  - Used by integer instructions
  - All registers are user privileged
- **Floating Point registers:**
  - Used by floating point instructions
  - SPE (signal processing engine) don't use it: SPE uses GPRs
- **Vector Registers: AltiVec**
- **SPR (Special Purpose Register):**
  - Ex: Link Register
  - Instructions used to manipulate these registers: m[t|f]spr
  - Not all registers are privileged ones
- **MSR: Machine State Register**
- **Memory Mapped Registers –** used to manipulate L2 cache settings
- **Thread Management Registers** (eg: setting CPU-thread priority)
  - CPU threads
  - Hypervisor
- **Performance Monitor Register**: User RO

# ARM Architecture: 32bit vs 64bit - SW Programmers View

## 32-bit ARM machines

- Supports 32bit execution mode
- Bootloader, Linux and apps execute in 32bit mode

## 64-bit ARM machines

- Supports 32bit (AArch32) and 64bit (AArch64) execution states
- Bootloader executes in 64bit mode
  Linux executes in 64bit mode
  Apps: 32b and 64b

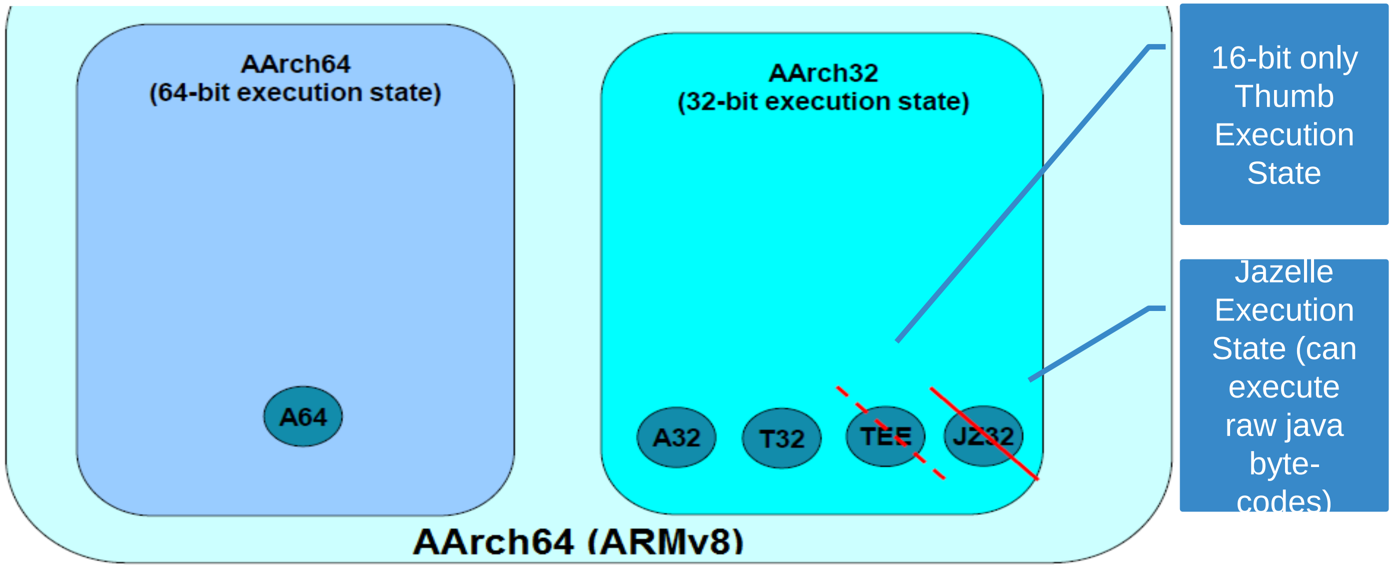
# ARM Architecture: 32bit vs 64bit - SW Programmers View

**AArch64** AArch64 state supports only a single instruction set, called A64. This is a fixed-width instruction set that uses 32-bit instruction encodings.

**AArch32** AArch32 state supports the following instruction sets:

- **A32** This is a fixed-width instruction set that uses 32-bit instruction encodings. It is compatible with the ARMv7 ARM instruction set.
- **T32** This is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings. It is compatible with the ARMv7 Thumb instruction set

- Rather than sharing the instruction decoder between 32-bit and 64-bit instruction sets, ARM implements a **separate decoder** for A64
  - This means that when running 64-bit software the 32-bit part of the machine does not need to be active.
- ***Limitation***:
  - Transition between 32-bit and 64-bit execution states can only occur on exception boundaries
    - i.e. it is not possible to interleave 32 bit and 64 bit code



# ARM Register Set (32-bit Implementations)

- **37 registers in total**
- **16 general purpose registers**
- **20 banked register**
- Registers are 32-bits long

- **The registers are arranged into several banks, with the accessible bank being governed by the processor mode.**

- **Some of the registers have special significance**
  - R13 – stack pointer (SP)
  - R14- link registers (LR)
  - R15 – Dedicated program counter (PC)

- **Status registers**
- 1 dedicated **Current** Program Status Register (CPSR)
- 5 dedicated **Saved** Program Status Register (SPSR)

- **There are banked SPs, LRs, and SPSRs for each privileged mode.**

**ARM state general purpose register and program counter**

| User/System | FIQ | IRQ | Abort | Undef | SVC |
|---|---|---|---|---|---|
| R0 | | | | | |
| R1 | | | | | |
| R2 | | | | | |
| R3 | | | | | |
| R4 | | | | | |
| R5 | | | | | |
| R6 | | | | | |
| R7 | | | | | |
| R8 | R8 | | | | |
| R9 | R9 | | | | |
| R10 | R10 | | | | |
| R11 | R11 | | | | |
| R12 | R12 | | | | |
| R13 (SP) | R13 (SP) | R13 (SP) | R13 (SP) | R13 (SP) | R13 (SP) |
| R14 (LR) | R14 (LR) | R14 (LR) | R14 (LR) | R14 (LR) | R14 (LR) |
| R15 (PC) | | | | | |
| CPSR | | | | | |
| | SPSR | SPSR | SPSR | SPSR | SPSR |

Current mode

Banked out registers

# ARM Register Set (64-bit Implementations)

- AArch64 has 31 general purpose registers (X0 – X30)
  – SP and PC are not general purpose registers.
- AArch64 Banked registers are banked by exception level
  – Used for exception return information and stack pointer
  – EL0 Stack Pointer can be used by higher exception levels after exception taken.

| | EL0 | EL1 | EL2 | EL3 |
|---|---|---|---|---|
| SP = Stack Ptr | SP_EL0 | SP_EL1 | SP_EL2 | SP_EL3 |
| ELR = Exception Link Register | | ELR_EL1 | ELR_EL2 | ELR_EL3 |
| Saved/Current Process Status Register | | SPSR_EL1 | SPSR_EL2 | SPSR_EL3 |

***Banked Registers***

31 general purpose registers accessible at all times

| | | | |
|---|---|---|---|
| X0 | X8 | X16 | X24 |
| X1 | X9 | X17 | X25 |
| X2 | X10 | X18 | X26 |
| X3 | X11 | X19 | X27 |
| X4 | X12 | X20 | X28 |
| X5 | X13 | X21 | X29 |
| X6 | X14 | X22 | X30* |
| X7 | X15 | X23 | |

***Un-Banked Registers***

| | X0-X7 | X8-X15 | X16-X23 | X24-X30 |
|---|---|---|---|---|
| 0 | R0 | R8_usr | R14_irq | R8_fiq |
| 1 | R1 | R9_usr | R13_irq | R9_fiq |
| 2 | R2 | R10_usr | R14_svc | R10_fiq |
| 3 | R3 | R11_usr | R13_svc | R11_fiq |
| 4 | R4 | R12_usr | R14_abt | R12_fiq |
| 5 | R5 | R13_usr | R13_abt | R13_fiq |
| 6 | R6 | R14_usr | R14_und | R14_fiq |
| 7 | R7 | R13_hyp | R13_und | No Register |

# COMPARISON OF ENVIRONMENTS: POWER® VS ARM® - ISA

# Instruction Set Architecture (ISA) – Power Architecture

| | |
|---|---|
| **Single ISA** | • Only one ISA is defined |
| **32bit execution mode** | • 32-bit effective address<br>• 32-bit registers<br>• Instructions to manipulate 32-bit address and 32-bit registers |
| **64bit execution mode** | • 64-bit effective address<br>• 64-bit registers<br>• Instructions to manipulate 64-bit address and 64-bit registers |

# Instruction Set Architecture (ISA) – ARM Architecture

**ARM Execution State: AArch32**

- Fixed length (32b) instructions
- Available in ARMv8 and earlier architectures
- ISA is A32 (ARMv7 and before: it's called ARM instructions)

**T32 (aka Thumb-2, applicable for AArch32)**

- A variable length (16b, 32b) instruction set
- ARMv7 & before, referred to as Thumb instructions and all instructions were 16-bit

**ARM Execution State: AArch64**

- Available in ARMv8 architecture only
- Instruction set is called A64
- Fixed length (32b) instructions

# ARM Vs PA Instruction Set Examples

**<operation>**
**<condition>** Is an optional field. It specifies the condition under which the instruction is executed.
**<Op2>** optional 2nd operand
**<Rd>** The destination register.
**<Rm>** The first operand register.

## Data processing instructions:

<operation><condition> Rd, Rm, <Op2>

```
ADDEQ   r4, r5, r6     ;r4 = r5 + r6
SUB     r5, r7, #4     ;r5 = (r7 - #4)
MOV     r4, #7         ;move immediate 7 into r4
```

## Memory access instructions:

<operation><size> Rd, [<address>]

```
LDR     r0, [r6, #4]   ; loading a 32 bit value
                       ; adding 4  to address in r6
                       ; loading the address result into r0

STRB    r4,[r7], #8    ; storing a byte
                       ; store the lower byte of r4 in to  the address
pointing to by r7
                       ; then update r7 with 8
```

## Program flow instructions:

<branch>{<condtion>} <label>

<branch>{<condtion>} <sub_routine_label>

```
B       func_1         ; branch
BL      func_2         ;branch with link
```

**<operation>**
**<rD>** The destination register.
**<rA rB>** Source or destination general purpose register
**SIMM/UIMM** Signed/Unsigned immediate 16 bit value

## Data processing instructions:

<operation> rD, rA, rB

```
Add     r4, r5, r6     ;r4 = r5 + r6
Subfic  r5, r7, 0x4    ;r5 = (0x4 – r7)
addi    r4, r0, 0x7    ;move immediate 7 into r4
```

## Memory access instructions:

<operation><size> Rd, [<address>]

```
lwz     r0, 0x4(r6)    ; loading a 32 bit value
                       ; adding 4  to address in r6
                       ; loading the address result into r0

stbu    r4,0x8(r7)     ; storing a byte
                       ; store the lower byte of r4 in to  the
address pointing to by r7+0x8
                       ; then update r7 with r7 + 0x8
```

## Program flow instructions:

<branch>{<condtion>} <label>

<branch>{<condtion>} <sub_routine_label>

```
b       func_1         ; branch
bl      func_2         ; branch with link
```

# COMPARISON OF ENVIRONMENTS: POWER® VS ARM® - IP ECOSYSTEM

# ARM v/s PA - IP Ecosystem

| IP | Power based QorIQ SoCs | ARM based QorIQ SoCs |
|---|---|---|
| Interrupt Controller | MPIC | GIC |
| IOMMU | PAMU | SMMU |
| Vector Instruction Processing | AltiVec | NEON |
| Bus | Corenet | CCN504, CCI400 |
| Data Path Arch | DPAA1 (Fman) | DPAA2 (WRIOP)<br>DPAA1 (Fman) |
| Timers | PowerPC timers<br>FlexTimer | ARM generic timers<br>FlexTimer |
| Watchdog | NXP's WDT | ARM's WDT |
| Platform cache (L3) | NXP's CPC | ARM CCN |

# ARM v/s PA - Misc Hardware Related Features

## Cache as SRAM

- Power Architecture supports CPC (Corenet Platform Cache), an L3 cache.
- CPC supports converting a part of cache into SRAM.
- Above SRAM is used during boot phase till DDR is initialized.
- **ARM: L3 cache doesn’t support above feature**

## Decorated Load / Store

- CPC supports decorated operation: High performance atomic “fire and forget” operations on memory
- **ARM L3 cache doesn’t have it.**

## LAW

- LAW stands for Local Access Window.
- Power architecture (using LAW) allows relocating the address of a target device.
- **ARM SoCs instead support a flat memory model.**

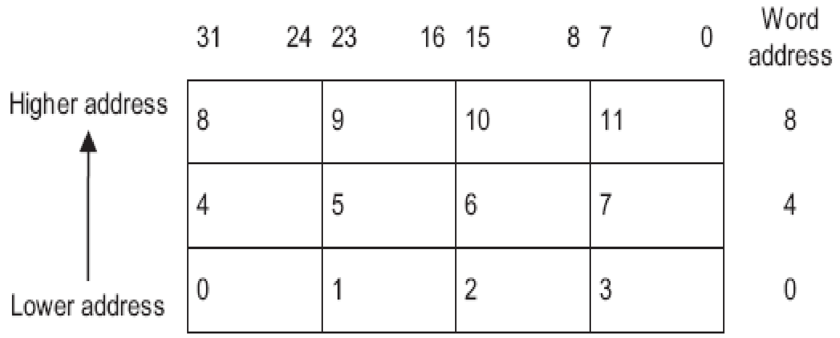
# COMPARISON OF ENVIRONMENTS: POWER® VS ARM® - ENDIANNESS

# Power Core Endianness

- Power processors are ***big-endian*** by default (can

- Big-endian memory systems
  - least significant byte is at lowest address.
  - Example:  byte 0 of the memory system connects to data lines 31 to 24

***<u>Big-endian</u>***



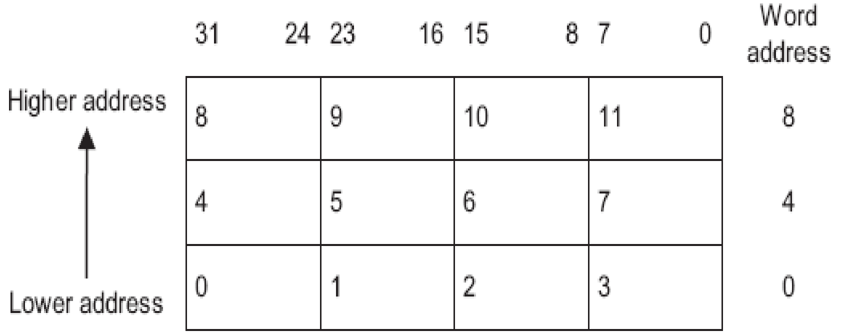
# ARM Core Endianness

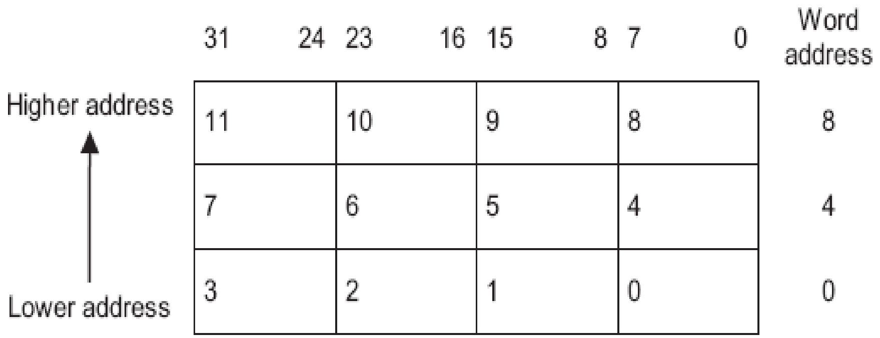
- ARM processors are ***little-endian*** by default (can be configured to access big-endian memory system).

- Big-endian memory systems
  - least significant byte is at lowest address.
  - Example: byte 0 of the memory system connects to data lines 31 to 24

- Little-endian memory systems
  - most significant byte is at lowest address
  - Example: byte 0 of the memory system connects to data lines 7 to 0.

***Big-endian***




***Little-Endian***

# Endianness – How Does It Affect a S/W Programmer?

**NW apps: Interpretation of network packets**

- Network packets follow network byte-order on wire – BE format
- If a networking program written originally for PPC platforms, doesn't take care of byte-ordering while interpreting a packet, it'll still work on BE machines but it may not work on LE machine.

**Device drivers: Accessing hardware registers**

- In QorIQ SoCs, most of the hardware registers are 32bit sized
- If a driver accesses a sub-field of the 32bit hardware register, the driver software needs attention when it migrates to ARM from PPC.

**Bitwise fields in structures**

- Bitwise members in *struct* may become a challenge for porting a software across architectures of opposite endianness

# Endianness – Rework S/W to Ease Transition to ARM

**NW apps: Interpretation of network packets**

- Use “ntoh()” family (ntohs, ntohl etc) while parsing the RX’ed packet.
- Use “hton()” family (htons, htonl etc) while forming a packet.
- Once above is done, the same source code compiles appropriately for the either core-endianness.

**Device drivers: Accessing hardware registers**

- Use ***accessors*** to access hardware registers
  - U-boot: IFC driver uses ifc_out32() to write IFC registers.
  - Accessors get compiled according to the underlying core endianness.

**Bitwise fields in structures**

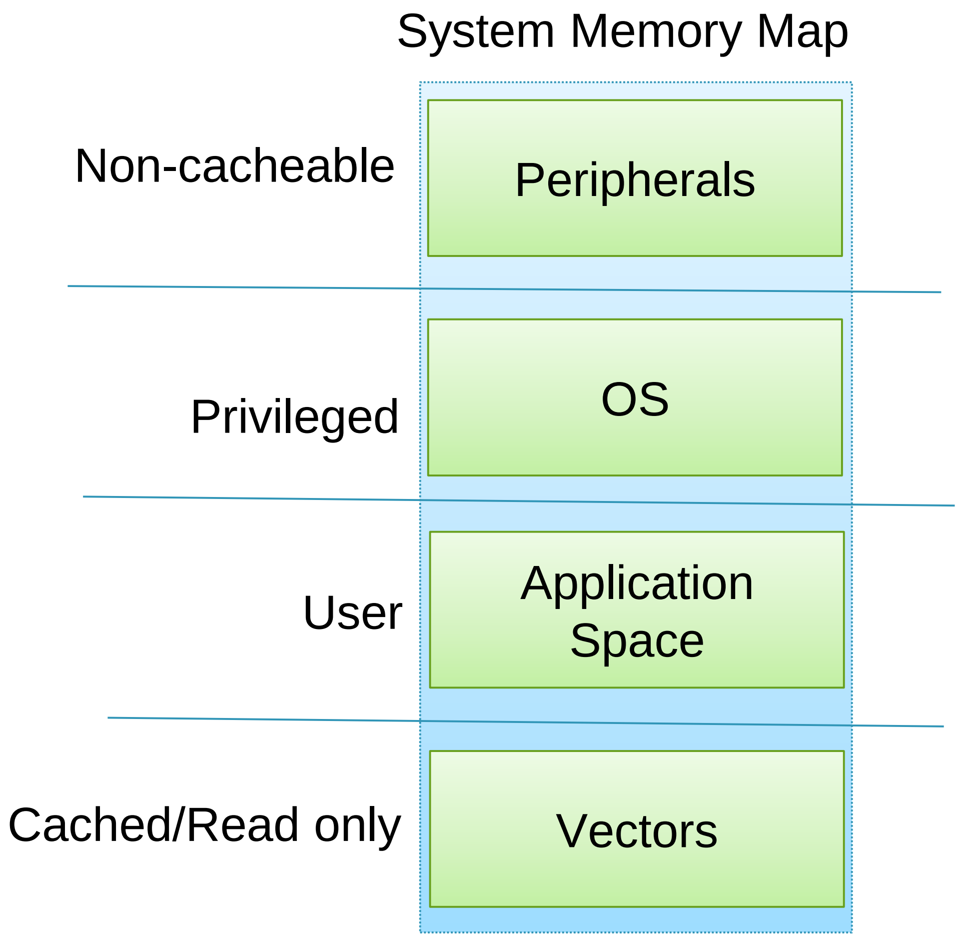
- Avoid using such *struct* when writing architecture independent code.

# ARM FUNDAMENTALS FOR GOOD SOFTWARE DESIGN - MEMORY MODEL

# Memory Model

- A system includes different memories and peripherals
  - The processor needs to be told how it should access different devices

- For each address region:
  - Access permissions (R/W permissions for User/Privileged modes)
  - Memory types (Caching/Buffering and access ordering rules for memory access)



# ARM Memory Types

- The ***type*** tells the attributes of the memory region
- ARMv8 Memory attributes:
  - **G**athering (merging multiple transactions),
  - **R**e-ordering (accesses in program-order),
  - **E**arly-Write-Ack-hint (only the end-point returns ack)

- In v6/v7: three memory types specified: Normal, Device and Strongly-ordered

| | |
|---|---|
| Normal | • Typically, memory used for program code and for data storage is ‘Normal’<br>• Sharable normal memory: Inner, Outer sharability domains<br>• Non-sharable normal memory: hardware generally doesn’t do coherency ops |
| Device | • Memory map accesses to system are defined as Device/ peripherals.<br>• Reads may result into side effect<br>• Memory attributes: nGnRE – no gathering, no re-ordering, early-ack |
| Strongly- ordered | • Examples: memory-mapped peripherals and I/O locations.(data used by legacy code)<br>• Memory attributes: “nGnRnE” – no gathering, no re-ordering, no early-ack |

Note: In ARMv8: ***Device*** and ***Strongly-Ordered*** have been  renamed to ***nGnRnE***, ***nGnRE*** respectively

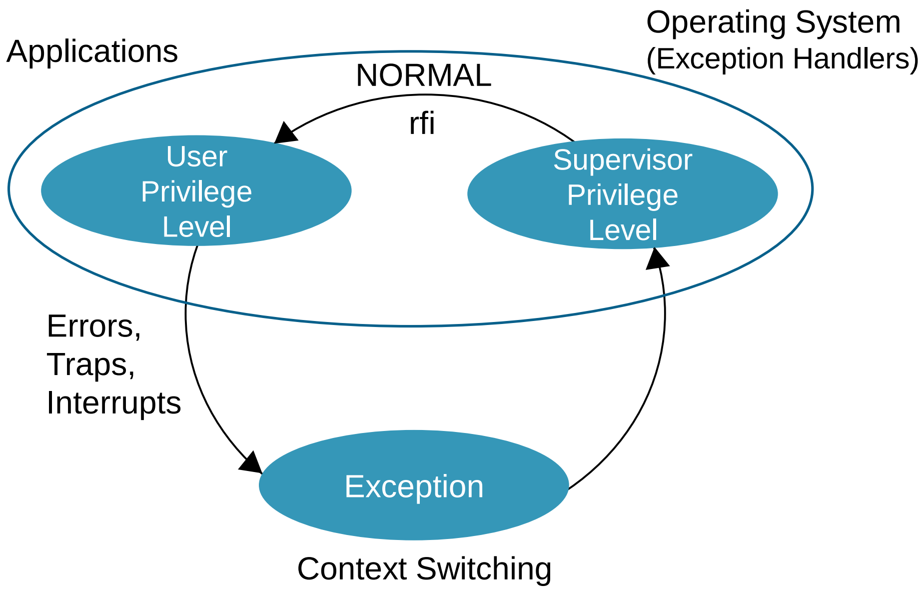
# Data & Stack Alignment

- **Data Alignment Requirements**
  - ARM cores supporting architecture v6 and later are capable of supporting unaligned accesses in hardware.
    - Data access can be unaligned
    - Address marked as “Normal” can be accessed unaligned
  - Load and store unit will access memory with aligned memory access.
  - ARM processor instruction and data is **little-endian (by default)**, but data side can be configured to access **big-endian** system.
- **Stack Alignment Requirements**
  - For AArch32 architectures:
    - Stack must be aligned to a 8-byte boundary.
  - For AArch64 architectures:
    - Stack must be aligned to a 16-byte boundary.

# ARM FUNDAMENTALS FOR GOOD SOFTWARE DESIGN - EXCEPTION MODEL

# Exceptions and Interrupts

- **Interrupts**
  - Action where processor saves current context and begins execution at predetermined interrupt handler
- **Exceptions**
  - Events which cause the processor to take an interrupt
    - Synchronous
    - Asynchronous



# PowerPC Exceptions Classes

| Type | Exception |
|---|---|
| Asynchronous/non maskable | Machine Check<br>System Reset |
| Asynchronous/maskable | External Interrupt<br>Decrementer |
| Synchronous/Precise | Instruction caused exception, excluding floating point imprecise exceptions |
| Synchronous/imprecise | Instruction caused imprecise exceptions<br>(Floating-point imprecise exception) |

# ARM processor Events (<= ARMv6)

ARM processor has 7 events that can halt the normal sequential execution of instructions.

| Event | Priority | I Bit | F Bit |
| --- | --- | --- | --- |
| Reset | 1 | 1 | 1 |
| Data Abort | 2 | 1 | 0 |
| FIQ | 3 | 1 | 1 |
| IRQ | 4 | 1 | 0 |
| Pre-fetch Abort | 5 | 1 | 0 |
| SWI | 6 | 1 | - |
| Undefined Instruction | 6 | 1 | - |

Event priority levels

Set /Cleared by ARM core (No S/W intervention required)

I and F bits represent the respective CPSR bits

Simultaneous multiple events :

- Current instruction will be completed no matter what event has been raised.

- Except when a data abort occurs on the first offending data address being accessed by LDM or STM.

# ARM processor Events (= ARMv7)

Two new ARM modes were introduced in ARMv7

**Monitor mode**

Monitor mode is a Secure mode

S/W running in Monitor mode has access to both the Secure and Non-secure copies of system registers.

**Hypervisor mode**

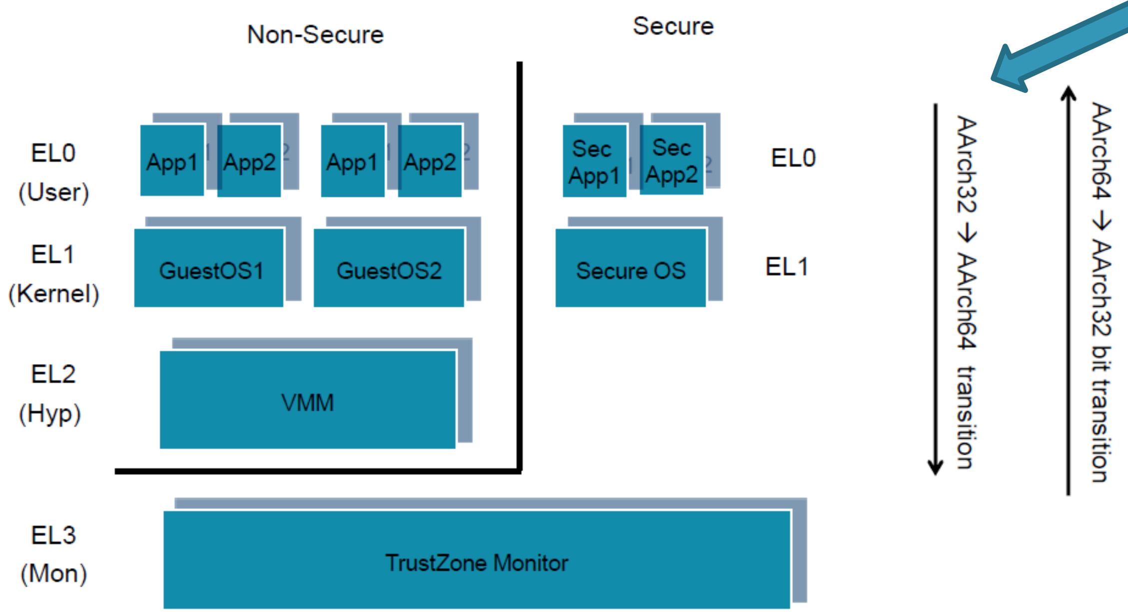
Hyp mode is a Non-secure EL2 mode.

Used to provide virtualization support.

# ARM Processor Events (= ARMv8)

- Architecture based around "Exception Levels" and Privilege Levels
  - EL0/EL1/EL2/EL3 and PL0/PL1/PL2/PL3

- Exception levels broadly designed for one software "layer"
  - EL0 – Application
  - EL1 – OS Kernel
  - EL2 – Hypervisor (non-secure only)
  - EL3 – Secure Monitor




- EL0
  - Lowest software execution privilege,
  - Execution at EL0 is called unprivileged execution.
- Increased values of n, from 1 to 3, indicate
- Increased software execution privilege.
- EL2 provides support for virtualization.
- EL3 provides support for two security states.

On taking an **exception to a higher exception level**, the execution state:

• Can either:
— Remain the same.
— Increase from AArch32 state to AArch64 state.

• Cannot decrease from AArch64 state to AArch32 state.

On returning from an **exception to a lower exception level**, the execution state:

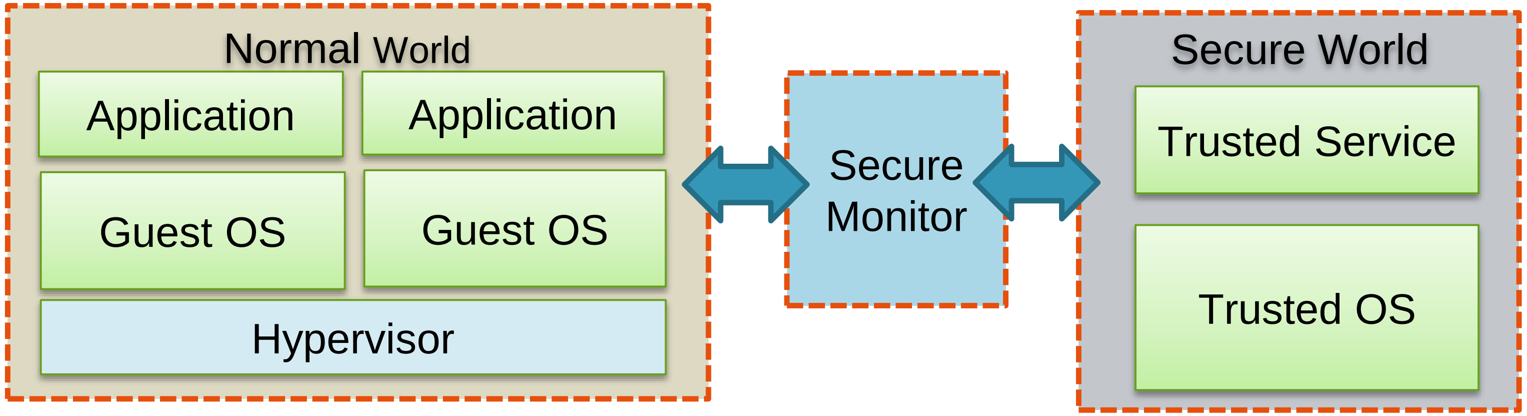
• Can either:
— Remain the same.
— Decrease from AArch64 state to AArch32 state.

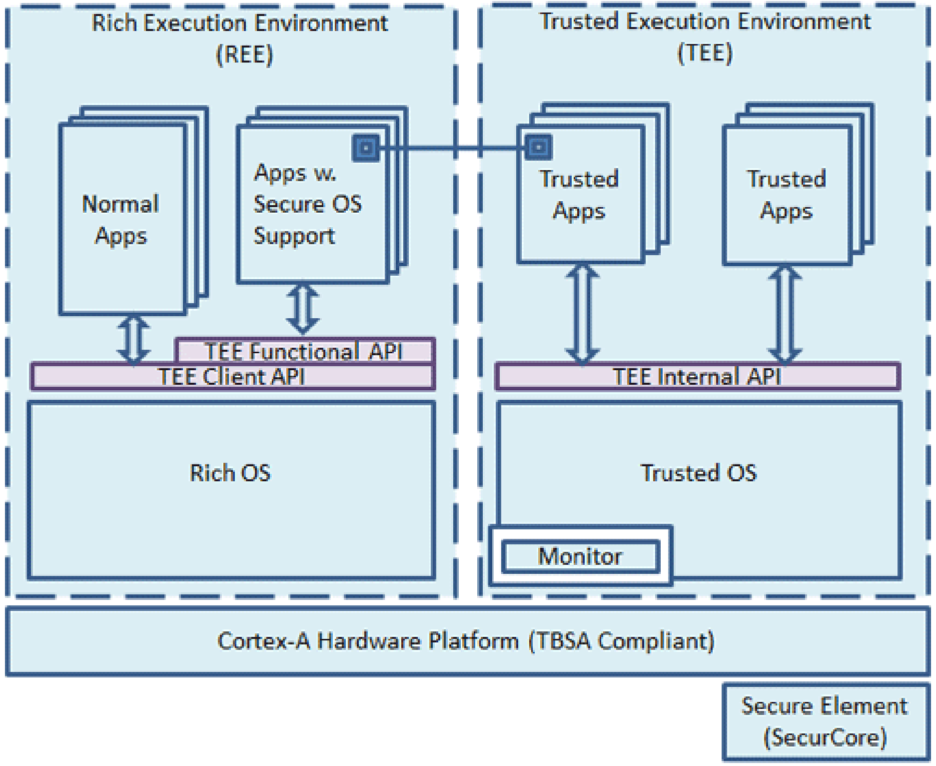
• Cannot increase from AArch32 state to AArch64 state.

# ARM FUNDAMENTALS FOR GOOD SOFTWARE DESIGN - SECURITY MODEL

# ARM Trustzone

- Security of the system is achieved by:
  - Partitioning all of the SoC’s hardware and software resources so that they exist in one of two worlds
    - ***Secure world*** for the security subsystem, and
    - ***Normal world*** for everything else



# SOFTWARE SUPPORT

# Power vs ARM: Should QorIQ Developers Worry?

| | | |
|---|---|---|
| **Packaging** | • Power and ARM SoC: SDKs of both are packaged using Yocto | Familiar host and target experience |
| **Development Environment** | • SDKs use GNU based compiler tools<br>• GCC tool-chains also support BE and ILP32 for ARM<br>• CodeWarrior® Debugger supports Power and ARM | |
| **Common Drivers** | • QorIQ LS software drivers already support both ARM and Power SoCs.<br>• You need not worry about endianness ☺ | Enablement SW takes care of architectural differences |
| **Quality** | • NXP is aggressive in upstreaming the platform and IP support code.<br>• Extensive testing for ARM and PowerPC based SoCs | Quality software on all QorIQ SoCs |

**Smooth transition of developers between Power and ARM QorIQ SoCs**

# SUMMARY

# Summary

QorIQ family includes both PowerPC and ARM based SoCs

There are architectural differences between PowerPC and ARM cores/SoCs

Developers moving from one architecture to the other should know these differences

QorIQ SDK takes care of above differences

**Ease of use for driver development:**
**Hassle-free transition between the two architectures**

SECURE CONNECTIONS
FOR A SMARTER WORLD

# ATTRIBUTION STATEMENT