ARMv8-A Architecture Overview

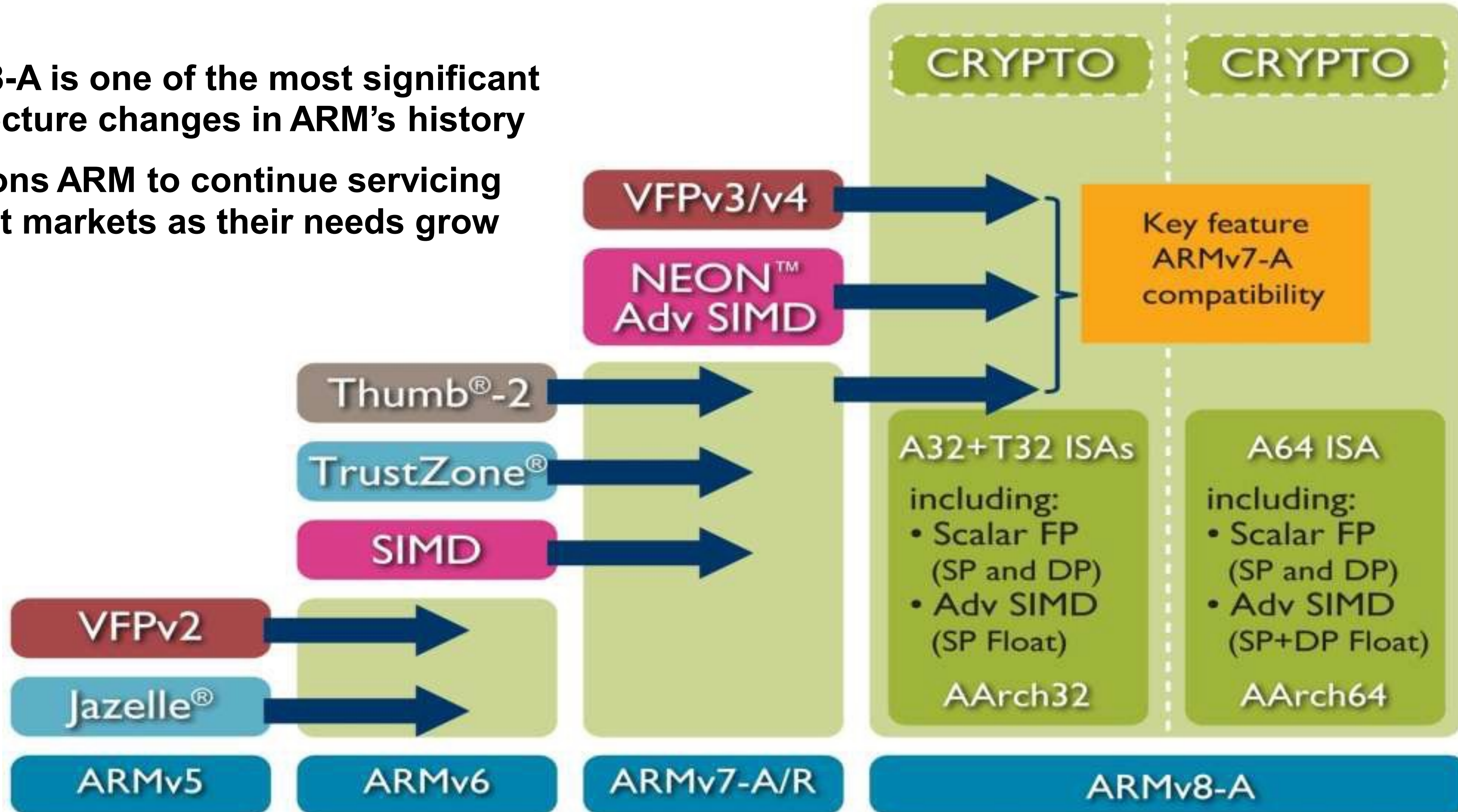
ARM®

ARMv8 terminology reference

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

Development of the ARM Architecture

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



ARM®

What’s new in ARMv8-A?

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

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

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

What's new in ARMv8.1-A

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

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

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

- **And other enhancements to the memory system architecture, such as Privileged Access Never (PAN) state bit**
  - Prevents kernel-mode (or hypervisor) from accessing memory allocated to user-mode

ARM®

Agenda

- **Privilege levels**

AArch64 Registers

A64 Instruction Set

AArch64 Exception Model

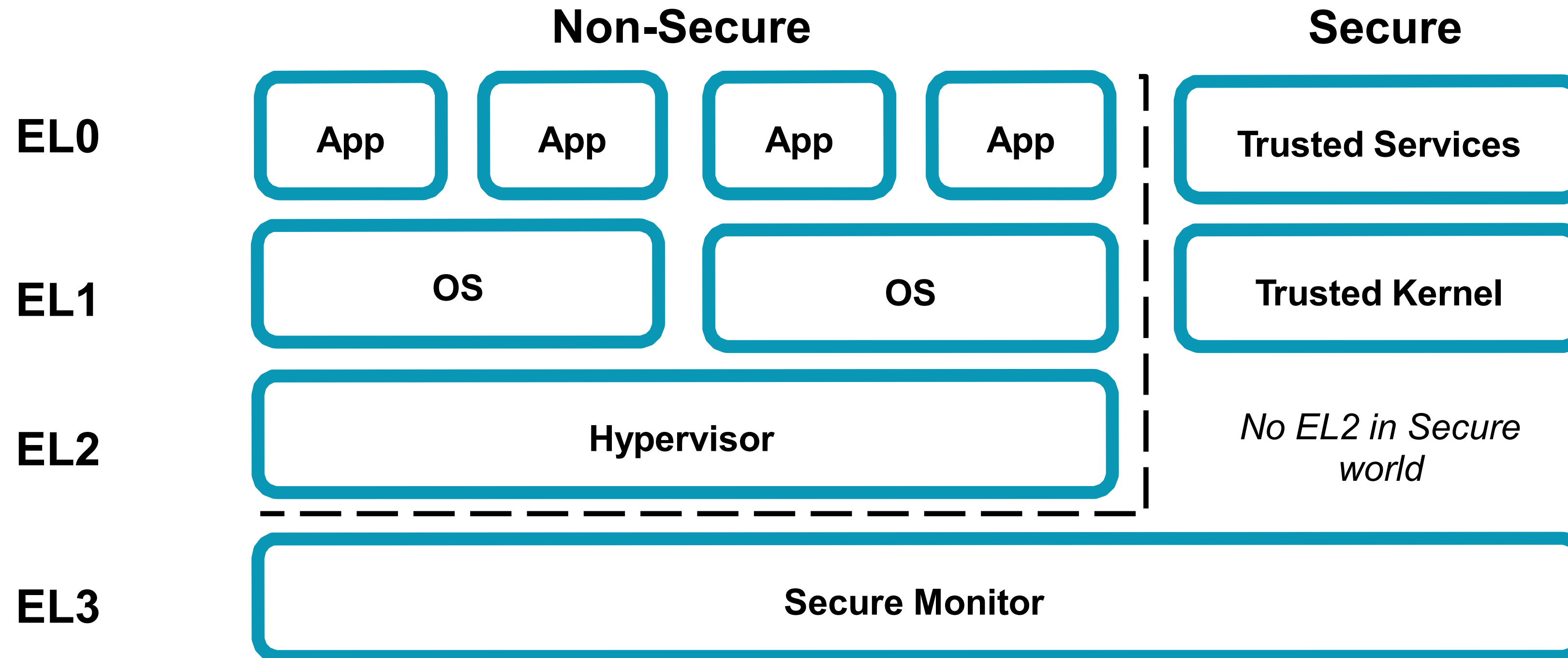
AArch64 Memory Model

AArch64 Cortex A5x

ARM®

AArch64 privilege model

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

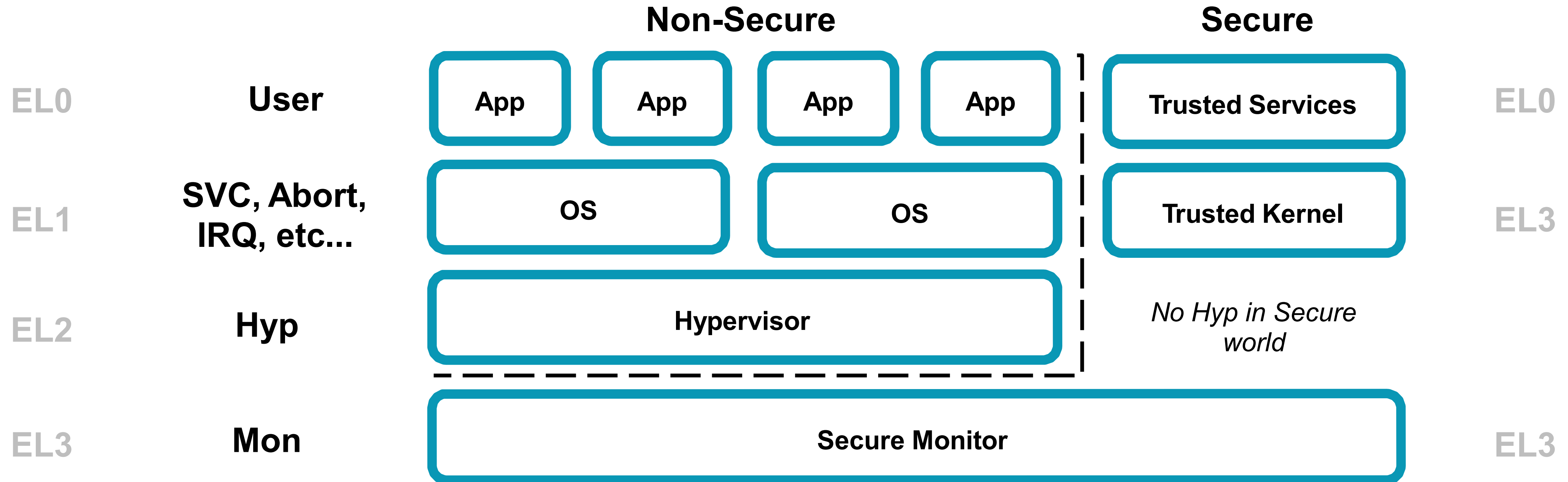


- **EL2 and EL3 are optional**
  - A processor may not implement EL2/3 if Security or Virtualization are not required

ARM®

AArch32 privilege model

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

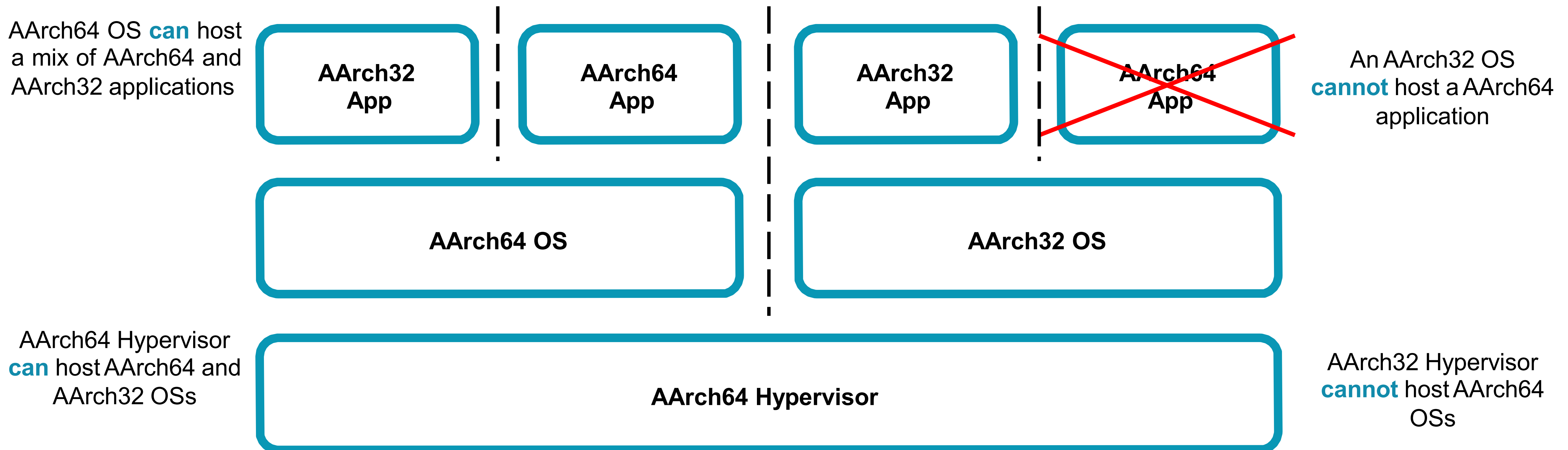


- **When EL3 is using AArch32, in the Secure world the EL1 modes are treated as EL3**
  - No effect on the Normal world

ARM®

Moving between AArch32 & AArch64

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



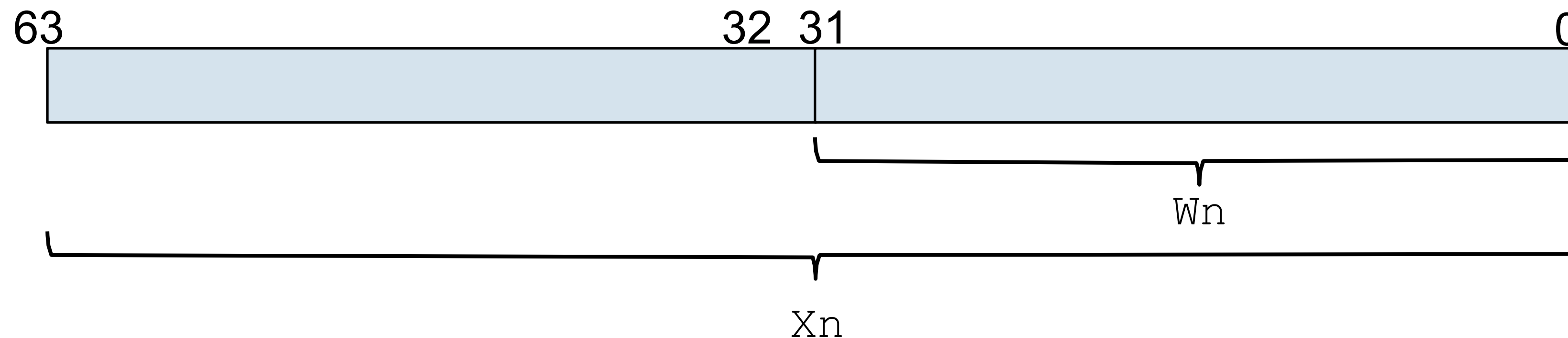
ARM®

Agenda

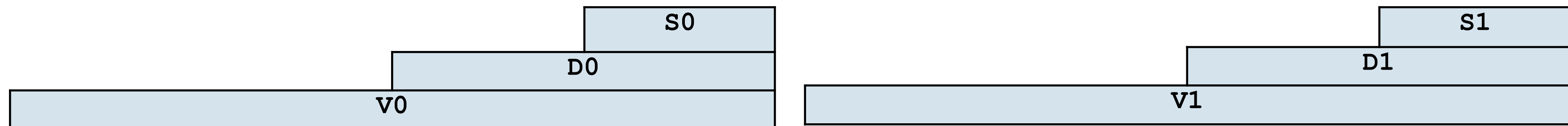
- Privilege levels
- **AArch64 Registers**
- A64 Instruction Set
- AArch64 Exception Model
- AArch64 Memory Model
- AArch64 Cortex A5x

Register banks

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



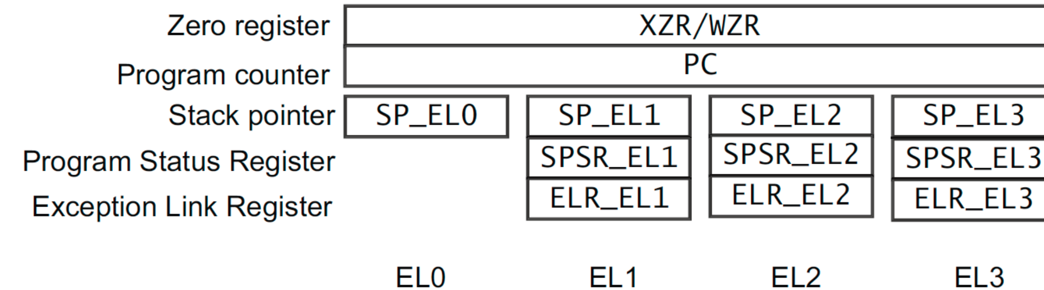
- **Separate register file for floating point, SIMD and crypto operations - `Vn`**
  - 32 registers, each 128-bits
    - Can also be accessed in 32-bit (Sn) or 64-bit (Dn) forms



ARM®

Other registers

- **AArch64 introduces the "zero" register – `XZR` and `WZR`**
  - Reads as 0, writes are ignored
- **The PC is not a general-purpose register, cannot be directly referenced**



- **There are separate link registers for function calls and exceptions**
  - **`X30`** – Updated by branch with link instructions (**`BL`** & **`BLR`**)
    Use **`RET`** instruction to return from sub-routines
  - **`ELR_ELn`** – Updated on exception entry
    Use **`ERET`** instruction to return from exceptions
- **Each exception level has its own stack pointer**
  - **`SP_EL0`**, **`SP_EL1`**, **`SP_EL2`** and **`SP_EL3`**
  - The SPs are not general purpose registers
  - Stack pointers must always be 128-bit aligned (bits 3:0 = b0000)
    - Hardware checking of SP alignment can be enabled

ARM®

Processor state

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

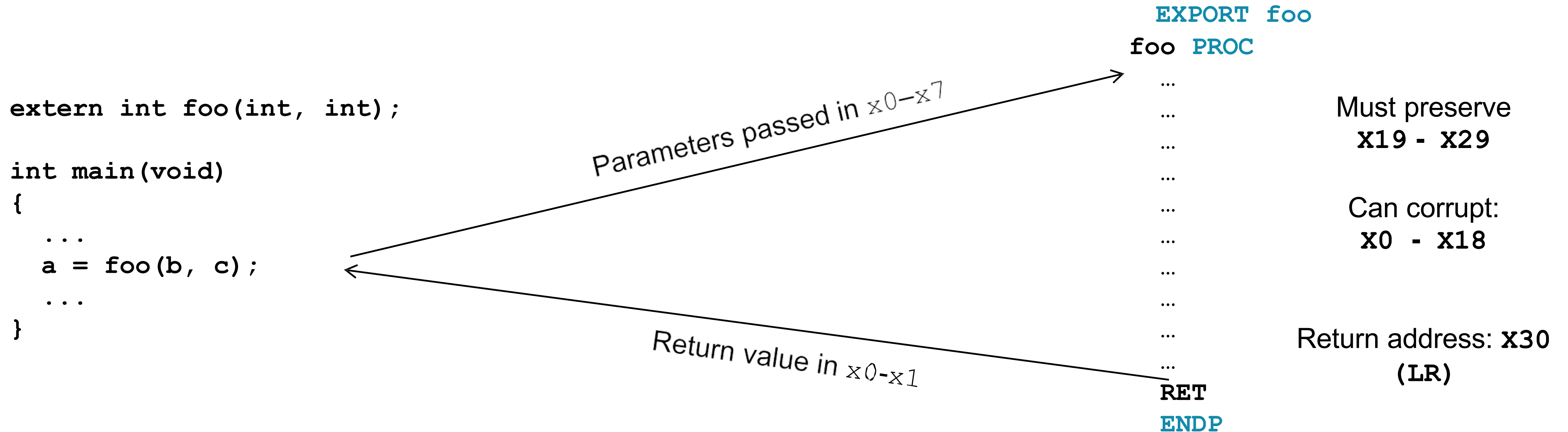
Fields	Description
N, Z, C and V	ALU flags
Q	Sticky overflow (AArch32 only)
DAIF	Exception mask bits
SPSel	SP selection (EL0 or ELn), not applicable to EL0
CurrentEL	The current exception level
E	Data endianness (AArch32 only)
IL	Illegal flag. When set, all instructions treated as UNDEFINED
SS	Software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

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

ARM®

Procedure call standard (1)

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



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

ARM®

Procedure call standard (2)

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

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

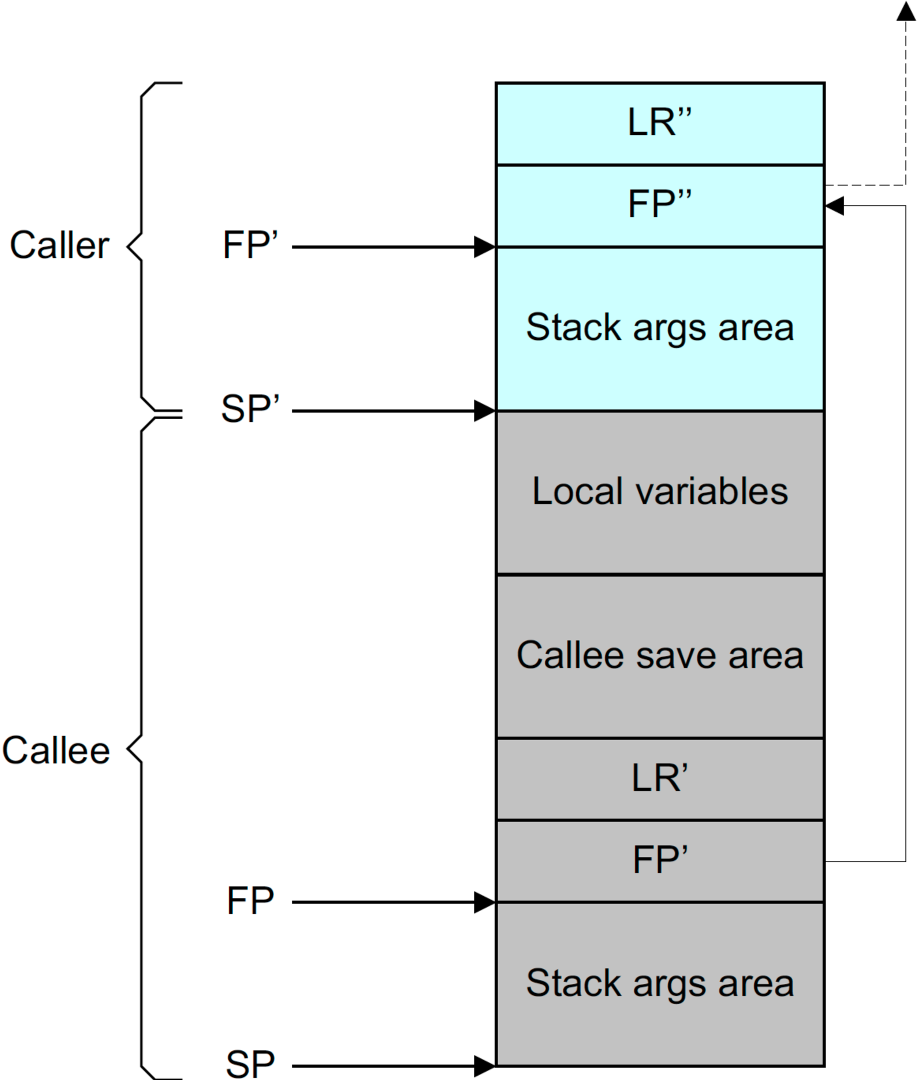
Procedure call standard (3)

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

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

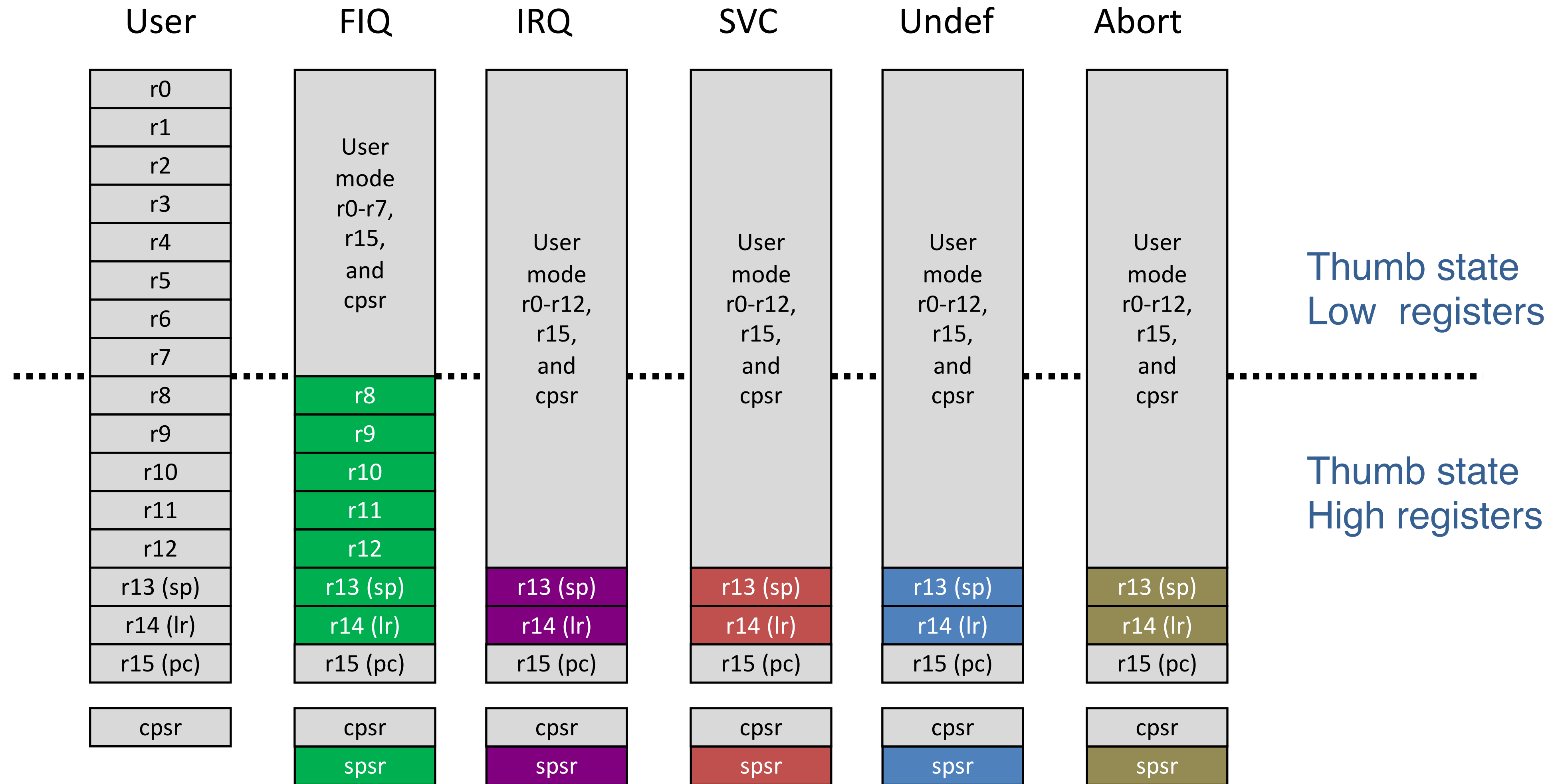
ARM®

Stack Frame



ARM®

V7 Register Organization Summary



ARM®

AArch64 ↔ AArch32 register mappings

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

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

ARM®

System control

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

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

    `TTBR0_EL2` – can be accessed from EL2 and EL3

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

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


ARM®

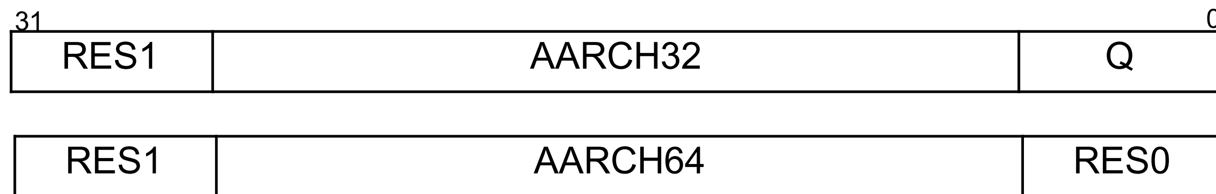
Some important System Registers

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

ARM®

REServed bits and the v8 Architecture

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



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

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

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

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

ARM®

System Register contents at reset

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

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

ARM®

Agenda

Privilege levels

AArch64 Registers

- **A64 Instruction Set**

AArch64 Exception Model

AArch64 Memory Model

AArch64 Cortex A5x

ARM®

A64 overview

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

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

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

ARM®

V7 vs. V8 Instruction formats

V7

ADD (register, ARM)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}
```

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–7	6–5	4	3–0
cond	0	0	0	0	1	0	0	S	Rn	Rd	imm5	type	0	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

V8

ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value, and writes the result to the destination register.

31	30	29	28	27	26	25	24	23–22	21	20–16	15–10	9–5	4–0
sf	0	0	0	1	0	1	1	shift	0	Rm	imm6	Rn	Rd
	op	S											

32-bit (sf == 0)

```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit (sf == 1)

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```


ARM®

Conditional execution

- **A64 does not allow instructions to be conditionally executed**
  - Except for branch instructions
  - Unlike A32, which allows for most instructions to include a condition code, for example `ADDEQ R0, R1, R2`
  - Unlike T32, which supports the `IT` (If Then) instruction

- **A64 has conditional operations**
  - These instructions are always executed, but their result depends on the ALU flags

- **Some data processing instructions will set the ALU flags after execution**
  - Mnemonics appended with ‘`S`’, for example `SUBS`
  - Some encodings have preferred syntax for disassembly to aid in clarity

```
SUBS   X0, X1, X2          ; X0 = (X1 - X2), and set ALU flags
TST    X0, #(1 << 20)      ; Alias of ANDS XZR, X0, #(1 << 20)
CMP    X0, #5              ; Alias of SUBS XZR, X0, #5
```


ARM®

Condition codes

Condition Code	Description	Flags Tested
EQ	Equal	Z == 1
NE	Not Equal	Z == 0
CS / HS	Unsigned Higher or Same	C == 1
CC / LO	Unsigned Lower	C == 0
MI	Minus	N == 1
PL	Positive or Zero	N == 0
VS	Overflow	V == 1
VC	No Overflow	V == 0
HI	Unsigned Higher	C == 1 && Z == 0
LS	Unsigned Lower or Same	C == 0 && Z == 1
GE	Greater Than or Equal	N == V
LT	Less Than	N != V
GT	Greater Than	Z == 0 && N == V
LE	Less Than or Equal	Z == 1 || N != V
AL	Always	--

NZCV → Negative, Zero, Carry, Overflow

ARM®

V7 Instruction Set Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Instruction Type
Condition				0	0	I	OPCODE				S	Rn				Rs				OPERAND-2												Data processing
Condition				0	0	0	0	0	0	A	S	Rd				Rn				Rs				1	0	0	1	Rm				Multiply
Condition				0	0	0	0	1	U	A	S	Rd HIGH				Rd LOW				Rs				1	0	0	1	Rm				Long Multiply
Condition				0	0	0	1	0	B	0	0	Rn				Rd				0	0	0	0	1	0	0	1	Rm				Swap
Condition				0	1	I	P	U	B	W	L	Rn				Rd				OFFSET												Load/Store - Byte/Word
Condition				1	0	0	P	U	B	W	L	Rn				REGISTER LIST																Load/Store Multiple
Condition				0	0	0	P	U	1	W	L	Rn				Rd				OFFSET 1				1	S	H	1	OFFSET 2				Halfword Transfer Imm Off
Condition				0	0	0	P	U	0	W	L	Rn				Rd				0	0	0	0	1	S	H	1	Rm				Halfword Transfer Reg Off
Condition				1	0	1	L	BRANCH OFFSET																								Branch
Condition				0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn				Branch Exchange
Condition				1	1	0	P	U	N	W	L	Rn				CRd				CPNum				OFFSET								COPROCESSOR DATA XFER
Condition				1	1	1	0	Op-1				CRn				CRd				CPNum				OP-2			0	CRm				COPROCESSOR DATA OP
Condition				1	1	1	0	OP-1			L	CRn				Rd				CPNum				OP-2			1	CRm				COPROCESSOR REG XFER
Condition				1	1	1	1	SWI NUMBER																								Software Interrupt

AArch64 cryptographic instructions

Instruction		Descriptrion
PMULL	Vd.1Q, Vn.1D, Vm.1D	Polynomial Multiply Long (Vector)
PMULL2	Vd.1Q, Vn.2D, Vm.2D	Polynomial Multiply Long (Vector, Part 2)
AESE	Vd.16B, Vn.16B	AES Single Round Encrypt
AESD	Vd.16B, Vn.16B	AES Single Round Decrypt
AESMC	Vd.16B, Vn.16B	AES Mix Columns
AESIMC	Vd.16B, Vn.16B	AES Inverse Mix Columns
SHA256H	Qd, Qn, Vm.4S	SHA256 Hash Update Accelerator
SHA256H2	Qd, Qn, Vm.4S	SHA256 Hash Update Accelerator (Part 2)
SHA256SU0	Vd.4S, Vn.4S	SHA256 Schedule Update
SHA256SU1	Vd.4S, Vn.4S, Vm.4S	SHA256 Schedule Update (Part 2)
SHA1C	Qd, Sn, Vm.4S	SHA1 Hash Update Accelerator (Choose)
SHA1P	Qd, Sn, Vm.4S	SHA1 Hash Update Accelerator (Parity)
SHA1M	Qd, Sn, Vm.4S	SHA1 Hash Update Accelerator (Majority)
SHA1H	Sd, Sn	SHA1 Hash Update Accelerator (Rotate Left by 30)
SHA1SU0	Vd.4S, Vn.4S, Vm.4S	SHA1 Schedule Update Accelerator
SHA1SU1	Vd.1Q, Vn.1D, Vm.1D	SHA1 Schedule Update Accelerator (Part 2)

ARM®

A64 Instruction Summary (partial)

Arithmetic Instructions			
ADC{S}	rd, rn, rm	rd = rn + rm + C	
ADD{S}	rd, rn, op2	rd = rn + op2	S
ADR	Xd, $\pm rel_{21}$	Xd = PC + $rel^{\pm}$	
ADRP	Xd, $\pm rel_{33}$	Xd = $PC_{63:12}$:0_{12} + $rel^{\pm}_{33:12}$:0_{12}	
CMN	rd, op2	rd + op2	S
CMP	rd, op2	rd − op2	S
MADD	rd, rn, rm, ra	rd = ra + rn × rm	
MNEG	rd, rn, rm	rd = − rn × rm	
MSUB	rd, rn, rm, ra	rd = ra − rn × rm	
MUL	rd, rn, rm	rd = rn × rm	
NEG{S}	rd, op2	rd = −op2	
NGC{S}	rd, rm	rd = −rm − ∼C	
SBC{S}	rd, rn, rm	rd = rn − rm − ∼C	
SDIV	rd, rn, rm	rd = rn $\bar{\div}$ rm	
SMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn $\bar{\times}$ Wm	
SMNEGL	Xd, Wn, Wm	Xd = − Wn $\bar{\times}$ Wm	
SMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn $\bar{\times}$ Wm	
SMULH	Xd, Xn, Xm	Xd = $(Xn \bar{\times} Xm)_{127:64}$	
SMULL	Xd, Wn, Wm	Xd = Wn $\bar{\times}$ Wm	
SUB{S}	rd, rn, op2	rd = rn - op2	S
UDIV	rd, rn, rm	rd = rn ÷ rm	
UMADDL	Xd, Wn, Wm, Xa	Xd = Xa + Wn × Wm	
UMNEGL	Xd, Wn, Wm	Xd = − Wn × Wm	
UMSUBL	Xd, Wn, Wm, Xa	Xd = Xa − Wn × Wm	
UMULH	Xd, Xn, Xm	Xd = $(Xn \times Xm)_{127:64}$	
UMULL	Xd, Wn, Wm	Xd = Wn × Wm	

Logical and Move Instructions			
AND{S}	rd, rn, op2	rd = rn & op2	
ASR	rd, rn, rm	rd = rn $\bar{\gg}$ rm	
ASR	rd, rn, $\#i_6$	rd = rn $\bar{\gg}$ i	
BIC{S}	rd, rn, op2	rd = rn & ∼op2	
EON	rd, rn, op2	rd = rn ⊕ ∼op2	
EOR	rd, rn, op2	rd = rn ⊕ op2	
LSL	rd, rn, rm	rd = rn ≪ rm	
LSL	rd, rn, $\#i_6$	rd = rn ≪ i	
LSR	rd, rn, rm	rd = rn ≫ rm	
LSR	rd, rn, $\#i_6$	rd = rn ≫ i	
MOV	rd, rn	rd = rn	S
MOV	rd, #i	rd = i	
MOVK	rd,$\#i_{16}${, sh}	$rd_{sh+15:sh}$ = i	
MOVN	rd,$\#i_{16}${, sh}	rd = $\sim(i^{\emptyset} \ll sh)$	
MOVZ	rd,$\#i_{16}${, sh}	rd = $i^{\emptyset} \ll sh$	
MVN	rd, op2	rd = ∼op2	
ORN	rd, rn, op2	rd = rn $\mid$ ∼op2	
ORR	rd, rn, op2	rd = rn $\mid$ op2	
ROR	rd, rn, $\#i_6$	rd = rn ⋙ i	
ROR	rd, rn, rm	rd = rn ⋙ rm	
TST	rn, op2	rn & op2	

Atomic Instructions			
CAS{A}{L}	rs, rt, [Xn]	if (rs = $[Xn]_N$) $[Xn]_N$ = rt	1
CAS{A}{L}{B,H}	Ws, Wt, [Xn]	if (Ws_{N0} = $[Xn]_N$) $[Xn]_N$ = Wt_{N0}	1
CAS{A}{L}P	rs,rs2,rt,rt2,[Xn]	if (rs2:rs = $[Xn]_{2N}$) $[Xn]_{2N}$ = rt2:rt	1
LDao{A}{L}{B,H}	Ws, Wt, [Xn]	Wt=$[Xn]^{\emptyset}_N$; $[Xn]_N$=ao($[Xn]_N$,Ws_{N0})	1
LDao{A}{L}	rs, rt, [Xn]	rt = $[Xn]_N$; $[Xn]_N$ = ao($[Xn]_N$, rs)	1
STao{A}{L}{B,H}	Ws, [Xn]	$[Xn]_N$ = ao($[Xn]_N$, Ws_{N0})	1
STao{A}{L}	rs, [Xn]	$[Xn]_N$ = ao($[Xn]_N$, rs)	1
SWP{A}{L}{B,H}	Ws, Wt, [Xn]	Wt = $[Xn]^{\emptyset}_N$; $[Xn]_N$ = Ws_{N0}	1
SWP{A}{L}	rs, rt, [Xn]	rt = $[Xn]_N$; $[Xn]_N$ = rs	1

Conditional Instructions			
CCMN	rn, $\#i_5$, $\#f_4$, cc	if(cc) rn + i; else N:Z:C:V = f	
CCMN	rn, rm, $\#f_4$, cc	if(cc) rn + rm; else N:Z:C:V = f	
CCMP	rn, $\#i_5$, $\#f_4$, cc	if(cc) rn − i; else N:Z:C:V = f	
CCMP	rn, rm, $\#f_4$, cc	if(cc) rn − rm; else N:Z:C:V = f	
CINC	rd, rn, cc	if(cc) rd = rn + 1; else rd = rn	
CINV	rd, rn, cc	if(cc) rd = ∼rn; else rd = rn	
CNEG	rd, rn, cc	if(cc) rd = −rn; else rd = rn	
CSEL	rd, rn, rm, cc	if(cc) rd = rn; else rd = rm	
CSET	rd, cc	if(cc) rd = 1; else rd = 0	
CSETM	rd, cc	if(cc) rd = ∼0; else rd = 0	
CSINC	rd, rn, rm, cc	if(cc) rd = rn; else rd = rm + 1	
CSINV	rd, rn, rm, cc	if(cc) rd = rn; else rd = ∼rm	
CSNEG	rd, rn, rm, cc	if(cc) rd = rn; else rd = −rm	

ARM®

General purpose and dedicated registers

- 31 general purpose + 2 dedicated registers, each 64-bit wide

same register

	GENERAL PURPOSE	DEDICATED	NOTES
Architectural names	`R0, R1, R2, …, R30`	`SP, ZR`	• names rarely used in practice
64-bit views	`X0, X1, X2, …, X30`	`SP, XZR`	• "`x`" stands for e**X**tended word • all 64 bits are "used"
32-bit views	`W0, W1, W2, …, W30`	`WSP, WZR`	• "`w`" stands for **W**ord • only **bottom** 32 bits are "used"

- Example: add two registers

INITIAL VALUES
X0=0x2222222222222222
X1=0x3333333333333333
X2=0xffffffffffffffff

`ADD X2, X0, X1`

AFTER 64-BIT ADD
X0=0x2222222222222222
X1=0x3333333333333333
X2=0x5555555555555555

All 64 bits of X2 are written

`ADD W2, W0, W1`

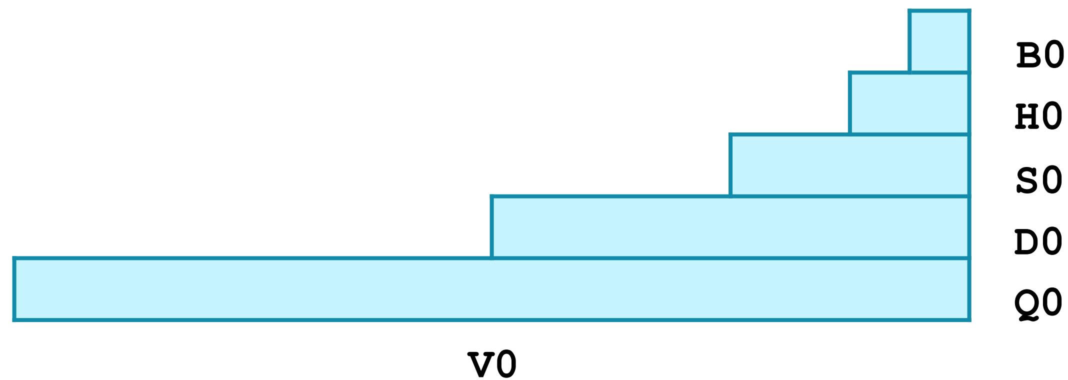
AFTER 32-BIT ADD
X0=0x2222222222222222
X1=0x3333333333333333
X2=0x0000000055555555

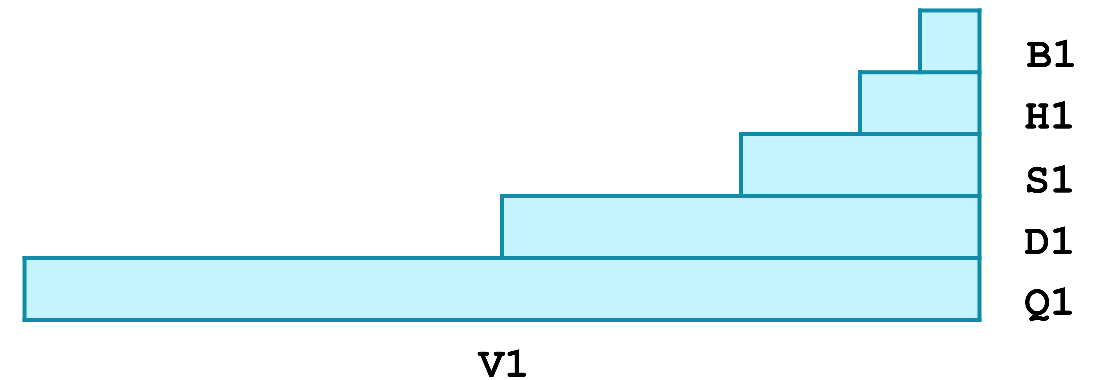
Upper 32 bits of X0, X1 are ignored.
Upper 32 bits of X2 are cleared on write

ARM®

Scalar FP and SIMD registers

- **Separate set of 32 registers, each 128-bit wide**
  - Architecturally named `V0` – `V31`
  - Used by scalar floating-point and SIMD instructions
- **The instruction syntax uses qualified register names**
  - `Bn` for byte, `Hn` for half-word, `Sn` for single-word, `Dn` for double-word, `Qn` for quad-word

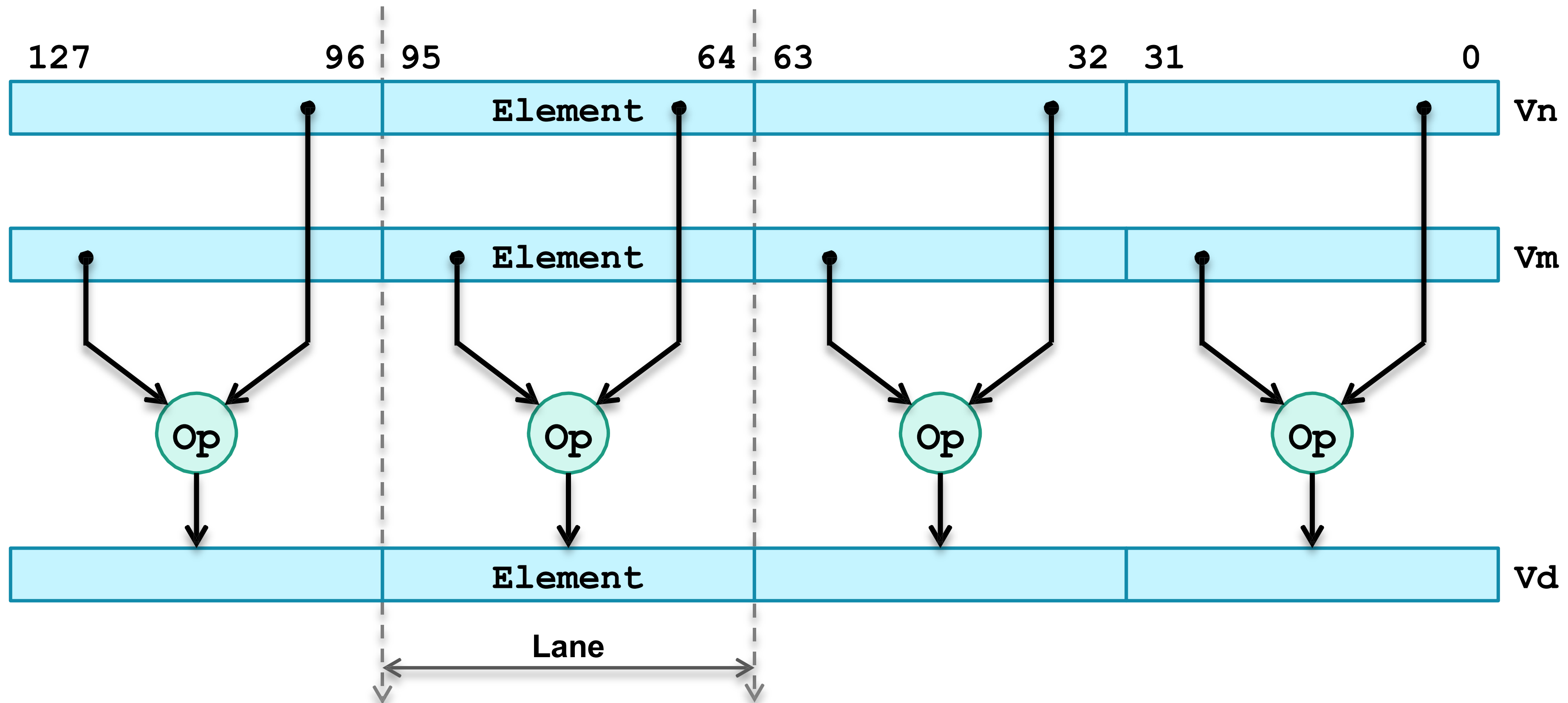




ARM®

SIMD operations

- **Operand registers are treated as being vectors of individual elements**
- **Operations are performed simultaneously on a number of "lanes"**
  - In this example each lane contains a pair of 32-bit elements, one from each of the operand 128-bit vector registers



ARM®

AAPCS64: Role of integer registers

Register	Alternative name	Role
R0		Return value (for integers and pointers)
R0 … R7		Arguments in function calls (for integers and pointers)
R8		Indirect result location register. Used in C++ for returning non-trivial objects (set by the caller).
R9 … R15		Temporary registers (trashed across calls)
R16, R17	IP0, IP1	The intra-procedure-call temporary registers. The linker may use these in PLT code. Can be used as temporary registers between calls
R18		Platform register
R19 … R28		Callee-saved registers: register preserved across calls
R29	FP	Frame pointer. Copy of SP before function stack allocation
R30	LR	Link register. BL and BLR instructions save return address in it
SP		Stack pointer

ARM®

AAPCS64: Role of floating-point registers

Register	Role
V0	Return value (for floating-point values)
V0 … V7	Arguments in function calls (for floating-point values)
V8 … V15	Callee-saved registers. The bottom 64 bits of these registers are preserved across calls. Upper 64 bits must be saved by the caller if needed.
V16 … V31	Temporary registers (trashed across calls)

ARM®

Specifying register load size

Load Size	Extension	Xn	
8-bit	Zero	`--`	
	Sign	`LDRSB`	
16-bit	Zero	`--`	
	Sign	`LDRSH`	
32-bit	Zero	`--`	
	Sign	`LDRSW`	
64-bit	Zero	`LDR`	

- There is no encoding for a zero-extended load of less than 64-bits to an `Xn` register
  - Writing to a `Wn` register automatically clears bits [63:32], which accomplishes the same thing

Store Size	Xn	Wn
8-bit	`--`	`STRB`
16-bit	`--`	`STRH`
32-bit	`--`	`STR`
64-bit	`STR`	`--`

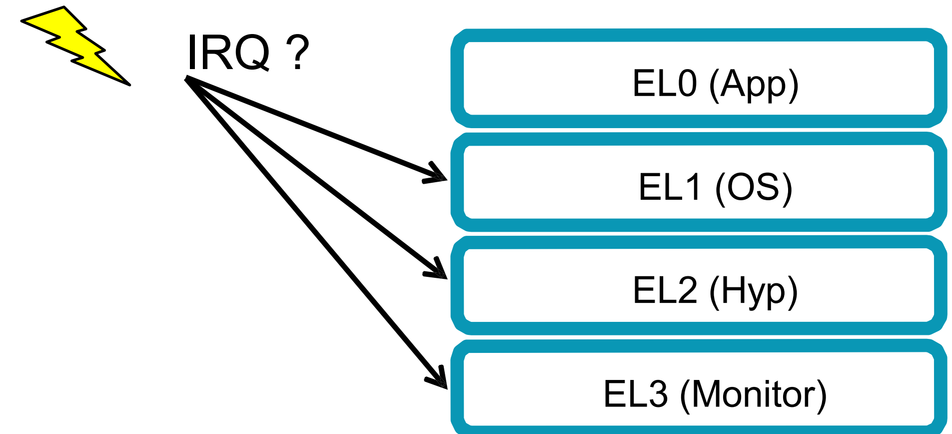
Agenda

- Privilege levels
- AArch64 Registers
- A64 Instruction Set
- **AArch64 Exception Model**
- AArch64 Memory Model
- AArch64 Cortex A5x

ARM®

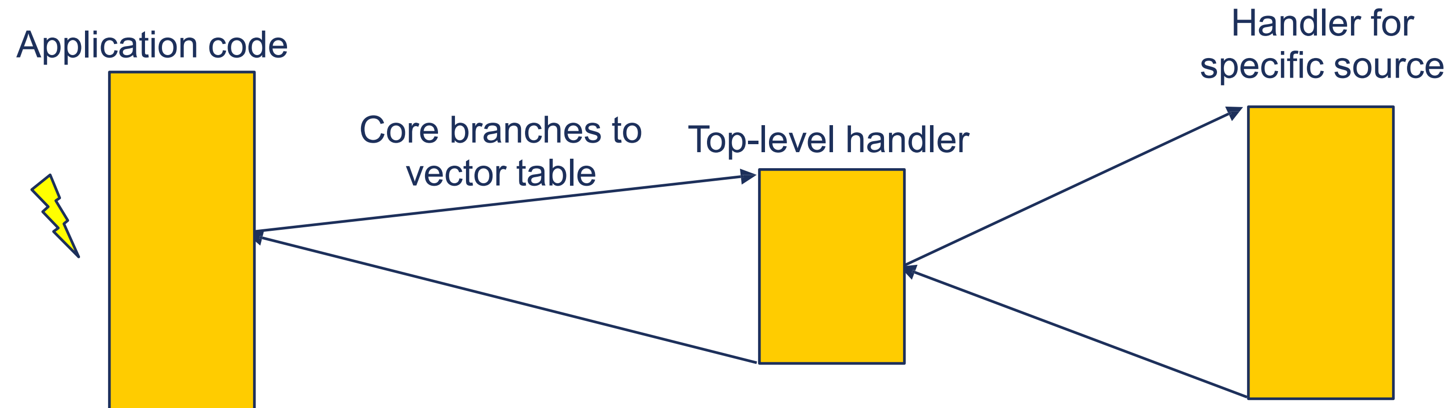
AArch64 exceptions

- **In AArch64 exceptions are split between:**
  - Synchronous
    - Data aborts from MMU, permission/alignment failures, service call instructions, etc.
  - Asynchronous
    - IRQ/FIQ
    - SError (System Error)
- **On taking an exception the EL can stay the same OR goes higher**
  - Exceptions are never taken to EL0
- **Synchronous exceptions are normally taken in the current EL**
- **Asynchronous exceptions can be routed to a higher EL**
  - **SCR_EL3** specifies exceptions to be routed to EL3
  - **HCR_EL2** specifies exceptions to be routed to EL2
  - Separate bits to control routing of IRQs, FIQs and SErrors



ARM

Taking an exception



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

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

ARM®

Agenda

Privilege levels

AArch64 Registers

A64 Instruction Set

AArch64 Exception Model

- **AArch64 Memory Model**

AArch64 Cortex A5x

ARM®

Memory types

- **Address locations must be describedin terms of a *type***
  - The "type" tells the processor how it can access that location
    - Access ordering rules
    - Speculation
- **Normal**
  - Used for code and data
  - Processor allowed to re-order, re-size and repeat accesses
  - Speculative accesses allowed
- **Device**
  - Used for peripherals
  - Accesses could have side effects, so there are more restrictions on what optimizations a processor can perform
  - Speculative data accesses not allowed
- **Other attributes can also be specified**
  - For example, whether a region is executable, shareable and cacheable

ARM®

Alignment

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

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

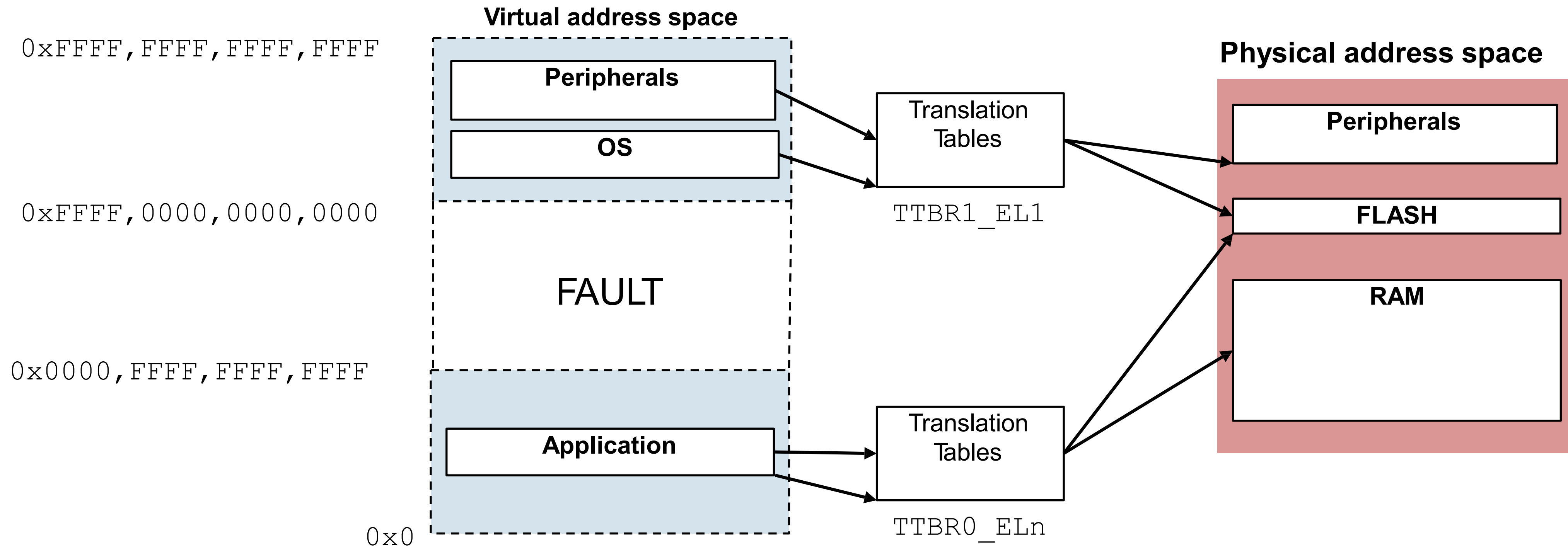
- **Unaligned data accesses to addresses marked as Device will always trigger an exception**
  - Synchronous data abort

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

ARM®

Virtual address space

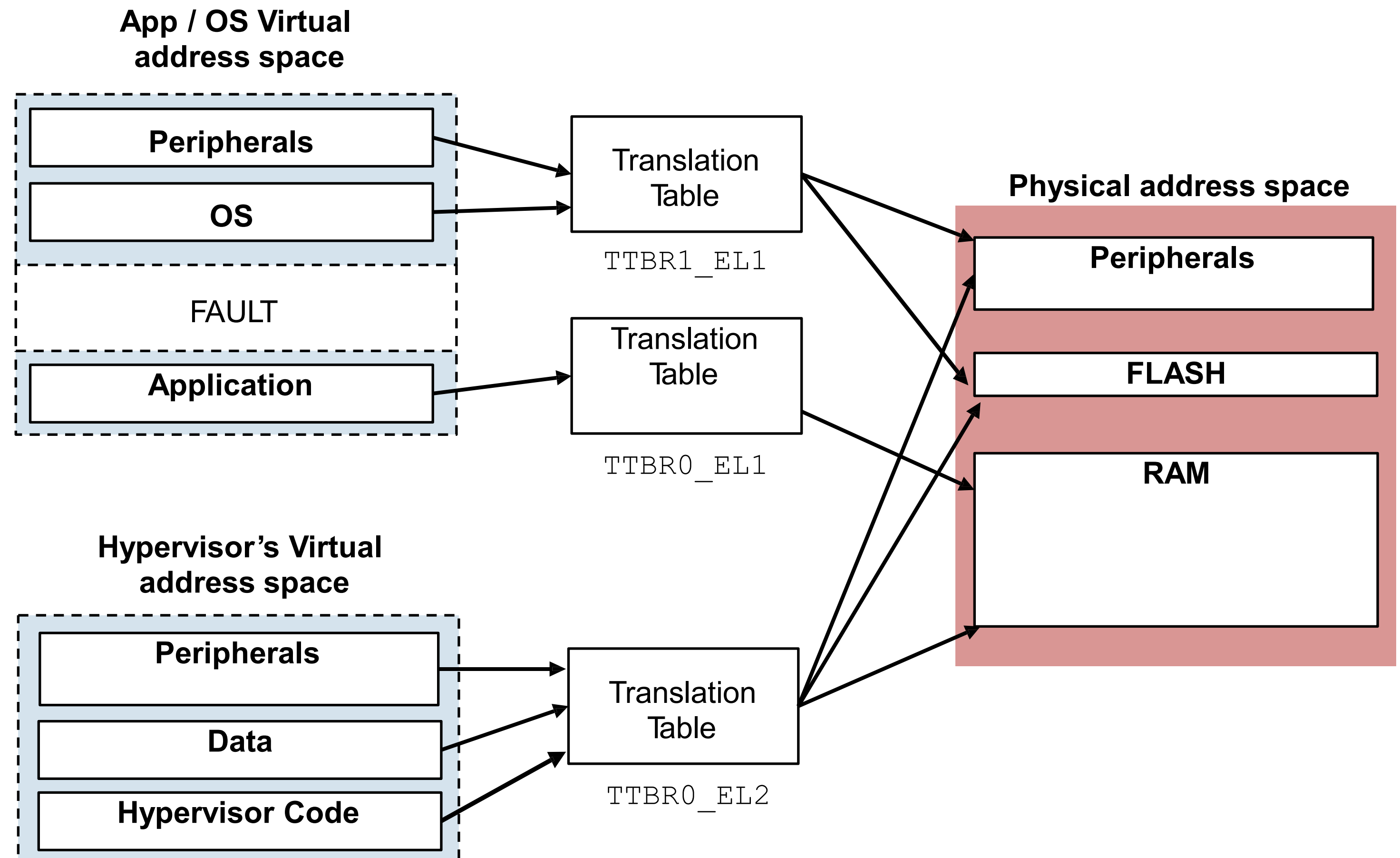
- **Virtual addresses are 64-bit wide, but not all addresses are accessible**
  - Virtual memory address space split between two translation tables
    - Each covering a configurable size, up to 48 bits of address space (**`TCR_ELn`**)
  - Addresses not covered by either translation table generate translation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



ARM

Multiple virtual address spaces

- A system may define multiple virtual address spaces:
- OS and applications
  - TTBR0_EL1
  - TTBR1_EL1
  - TCR_EL1
- Hypervisor
  - TTBR0_EL2
  - TCR_EL2
- Secure Monitor
  - TTBR0_EL3
  - TCR_EL3



ARM®

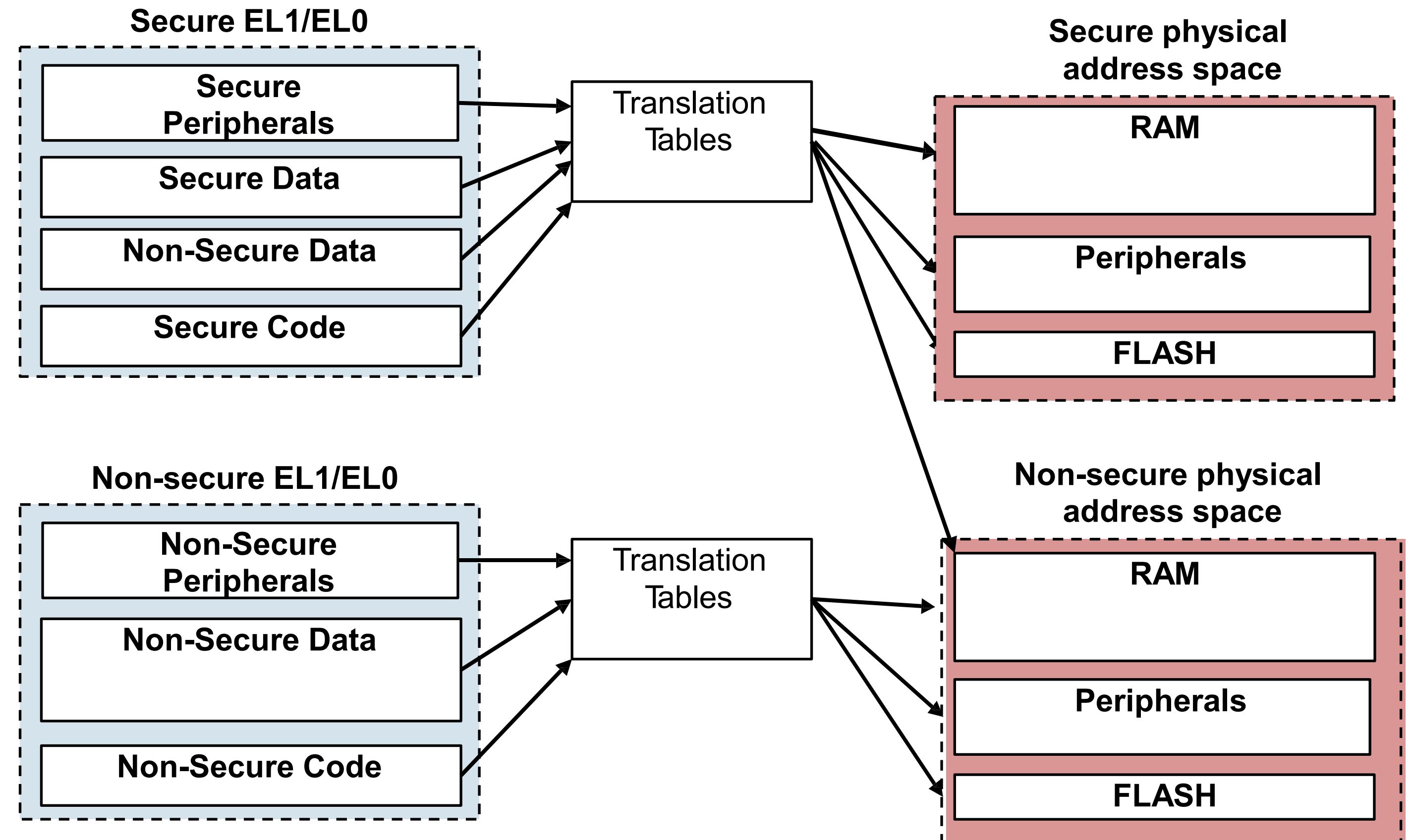
Physical address spaces

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

- **These are in theory completely separate:**
  - SP:0x8000 != NP:0x8000
  - But most systems treat Secure/Non-Secure as an attribute for access control

- **Normal world can only access the non-secure physical address space**

- **Secure world can access BOTH physical address spaces**
  - Controlled through translation tables



ARM®

Agenda

Privilege levels

AArch64 Registers

A64 Instruction Set

AArch64 Exception Model

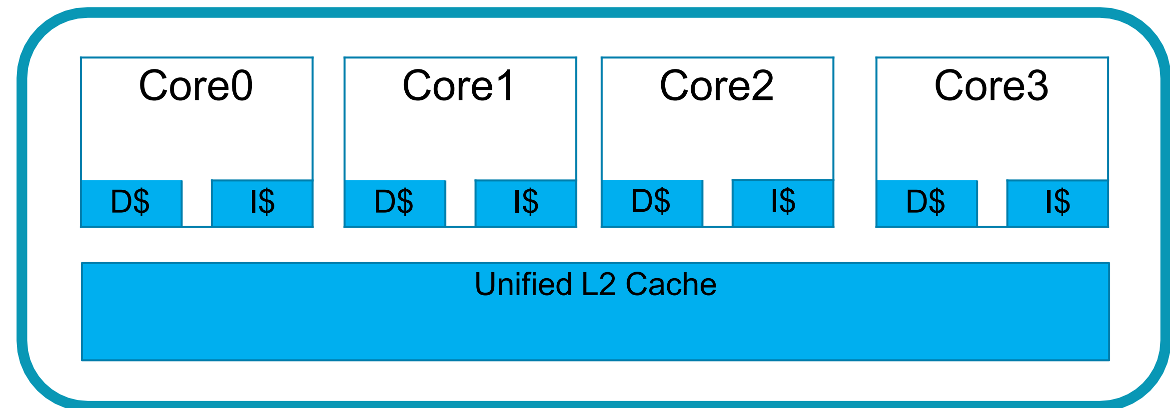
AArch64 Memory Model

- **AArch64 Cortex A5x**

ARM®

MPCore configurations

- **Many implementations of ARM processors have a multi-core configuration**
  - Multiple cores contained within the same block
- **Each core has its own MMU configuration , register bank, internal state and Program Counter**
  - Core0 might be executing in Non-secure, AArch32 EL0 while Core1 is executing in Secure, AArch64 EL1
- **Cores can be powered and brought in and out of reset independently**
  - ID registers allow discovery of core affinity
- **Each core has separate L1 data and instruction caches**
  - Hardware will maintain coherency between L1 data caches for certain memory types
  - Some cache and TLB instructions are broadcast to other cores
  - All cores share a common physical memory map



ARM®

ARM Cortex-A5x MP Cores

Interrupt System (GIC)

Cortex-A57

L1 I Cache | L1 D Cache

L2 Cache

Interrupt System (GIC)

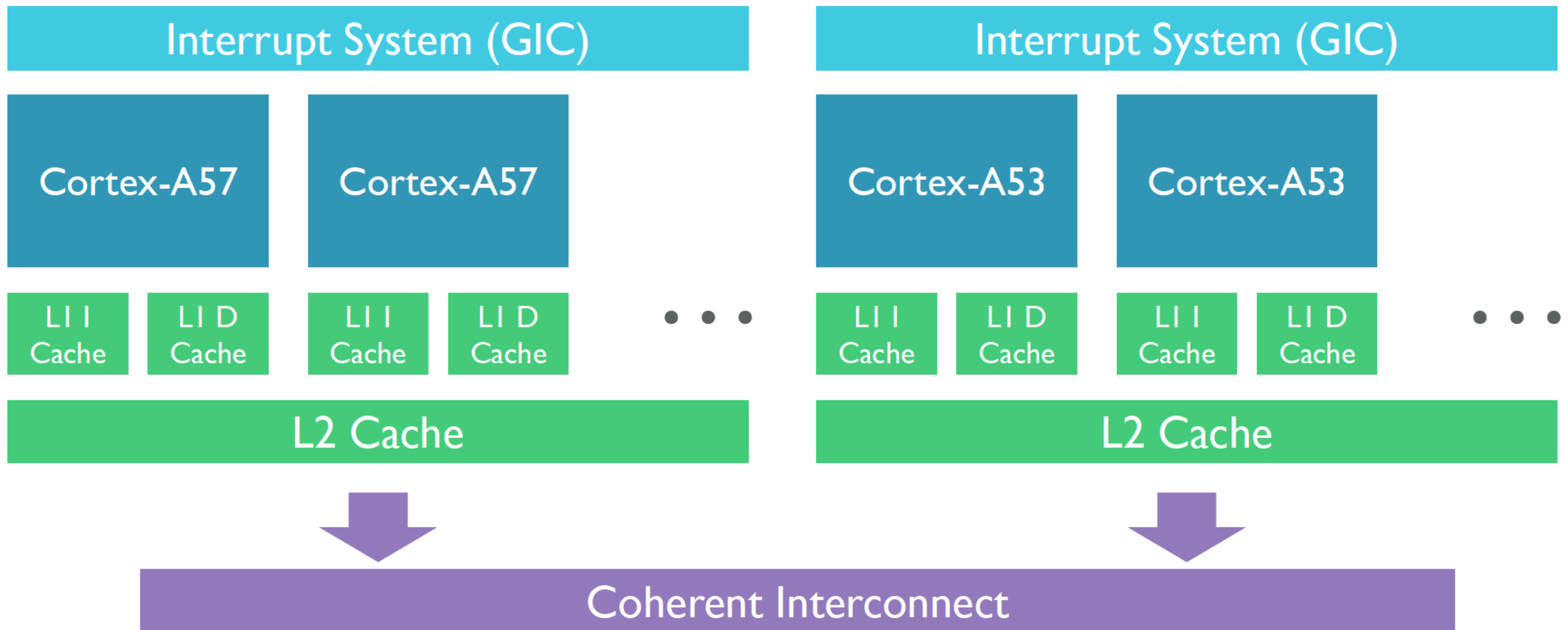
Cortex-A53

L1 I Cache | L1 D Cache

L2 Cache

ARM®

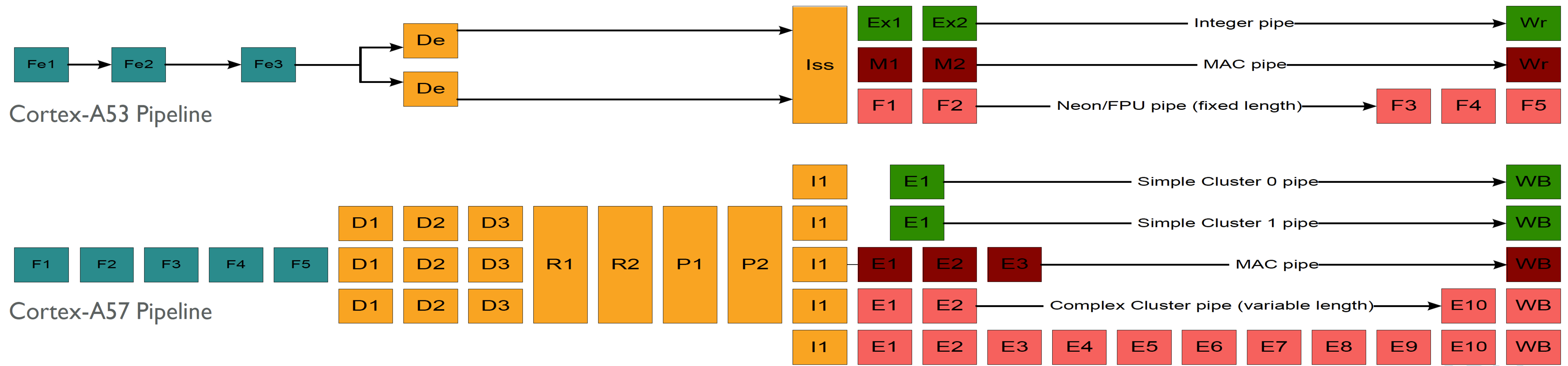
ARM Cortex A5x Multi-cluster (big.LITTLE)



ARM

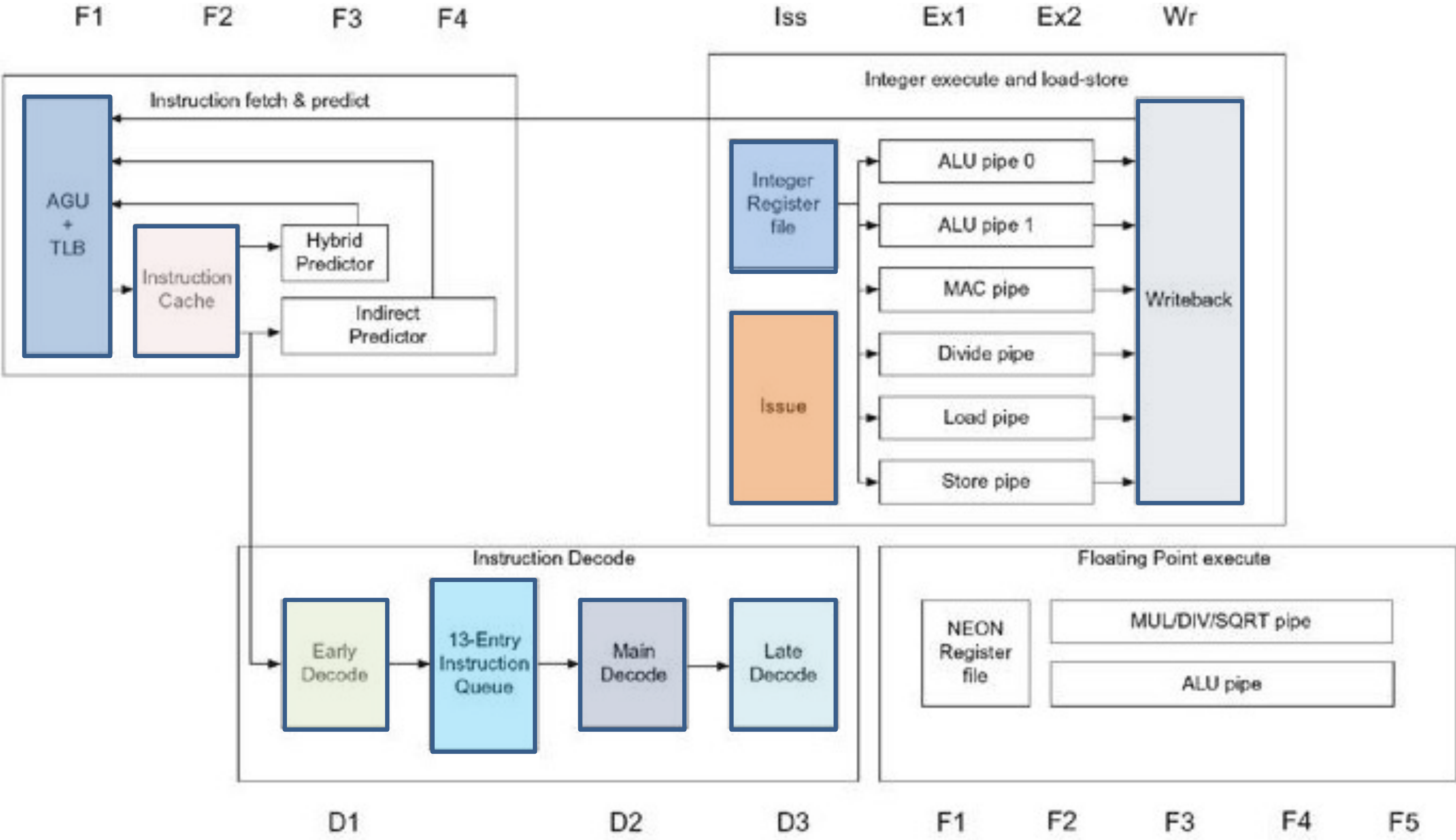
ARM Cortex A5x Power efficiency differences

- Cortex-A53: design focused on energy efficiency (while balancing performance)
  - 8-11 stages, In-Order and limited dual-issue
- Cortex-A57: focused on best performance (while balancing energy efficiency)
  - 15+ stages, Out-of-Order and multi-issue, register renaming
  - Average 40-60% performance boost over Cortex-A9 in general purpose code
    - Instructions per cycle (IPC) improvements



ARM®

Detailed Cortex-A53 Pipeline



ARM®

Appendix

ARM®

ARMv8-A software support

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

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

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

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

ARM®

ARMv8-A software support continued

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

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