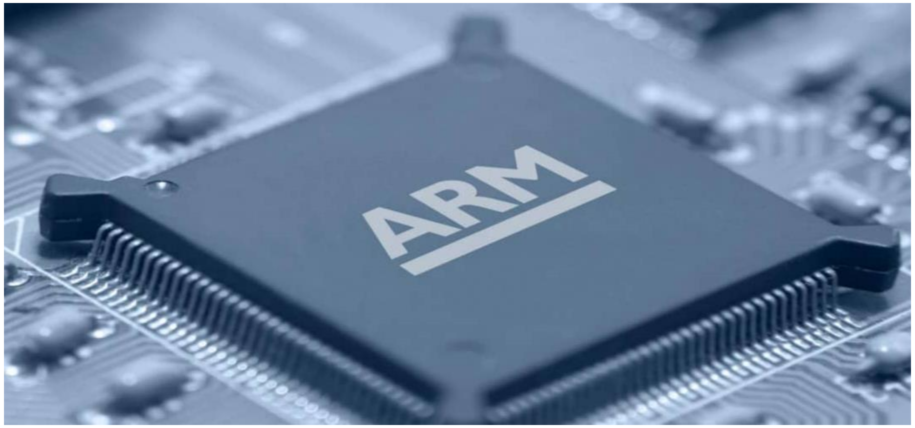
ARM (Advanced RISC Machine)

Basic - Quick Start to Understand



OCTOBER 2019

KENJI IIJIMA

Objective/Disclaimer

Goal:

- {For author} To build basic understanding of ARM architecture and technology to ramp up as FAE to support on behalf of ARM based SOC vendors from October 2019
- {To reader} To have basic ideas of key ARM technology/functionality with product portfolio

Disclaimer:

- This is non-NDA base (Readers may distribute to anyone without consensus of the author).
- The author may publish it to any social media or specific individuals to enhance his/her publicity.
- The author has no responsibility of correctness of documents despite the best effort to make as precise as possible.
- This document does not make readers know whole things about ARM architecture. For further understanding, recommended to read ARM Architecture Reference Manual, etc

Prerequisite background knowledge

- Basic understanding of microprocessor system
- Super basic concept of assembly code

Potential items to add if this document is upgraded:

- ARM instruction sets
- More details of ARM microarchitecture
- List of evolutions of ARM over generation (i.e. upgraded features of Cortex-A72 from A53)
- Introduction of some vendors ARM based product (i.e. Qualcomm, NXP, ST Micro, Microchip etc)

Contents

ARM Overview

ARM Architecture Basic

ARM Key technologies

ARM Misc

Contents

ARM Overview

- Corporate Overview
- Guidance to understand product portfolio

ARM Architecture Basic

ARM Key technologies

ARM Misc

ARM Corporate Overview

ARM = Advanced RISC Machine (Previously Acorn RISC Machine)

ARM Holdings

- Founded in 1990
- HQ: Cambridge UK
- Key persons: Chairman = Masayoshi Son (Softbank), CEO = Simon Segars

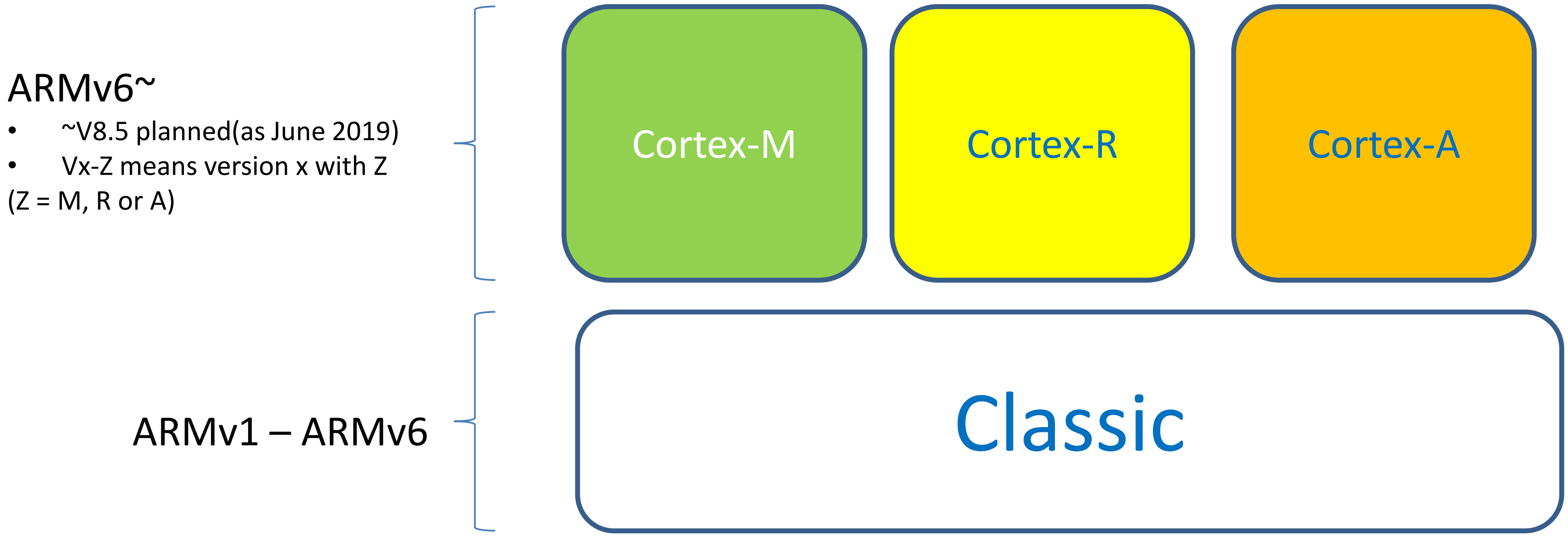
ARM deliverable in the market

- License of microprocessor/GPU designs
- Software Dev Tool

3 types of licenses

1. Core: Taking IP of architecture to build into their SoC
2. Built-on Cortex Tech license (BoC): Partner with ARM to modify designs together
3. Architectural license: Take design to re-architecture to their own based on ARM instruction set
4. Mali license: GPU design

ARM Architecture Categories



ARMv6~

- ~V8.5 planned(as June 2019)
- Vx-Z means version x with Z

(Z = M, R or A)

Cortex-M

Cortex-R

Cortex-A

ARMv1 – ARMv6

Classic

Classic = Fundamental architectures a

- When there was no differentiation between M/R/A
- Base functions for M/R/A

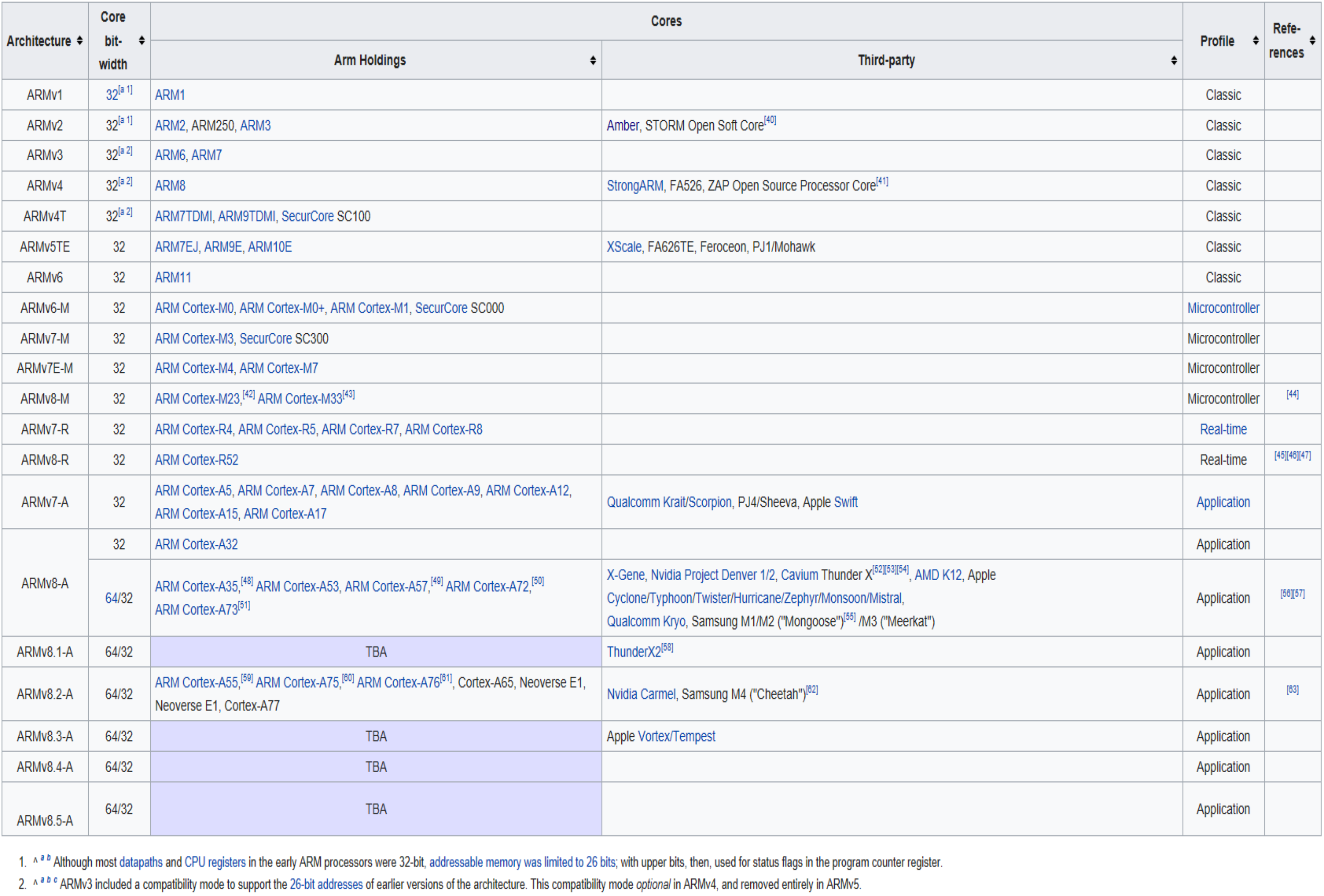
Cortex-M: Microcontroller profile

Cortext-R: Real-time profile

Cortex-A: Application profile

“Classic” is still reflected after ARMv6~

ARM Version Matrix (Copied from Wiki)



Architecture	Core bit-width	Cores		Profile	Refe-rences
		Arm Holdings	Third-party		
ARMv1	32[a 1]	ARM1		Classic	
ARMv2	32[a 1]	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core[40]	Classic	
ARMv3	32[a 2]	ARM6, ARM7		Classic	
ARMv4	32[a 2]	ARM8	StrongARM, FA526, ZAP Open Source Processor Core[41]	Classic	
ARMv4T	32[a 2]	ARM7TDMI, ARM9TDMI, SecurCore SC100		Classic	
ARMv5TE	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	Classic	
ARMv6	32	ARM11		Classic	
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Microcontroller	
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Microcontroller	
ARMv7E-M	32	ARM Cortex-M4, ARM Cortex-M7		Microcontroller	
ARMv8-M	32	ARM Cortex-M23,[42] ARM Cortex-M33[43]		Microcontroller	[44]
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7, ARM Cortex-R8		Real-time	
ARMv8-R	32	ARM Cortex-R52		Real-time	[45][46][47]
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Krait/Scorpion, PJ4/Sheeva, Apple Swift	Application	
ARMv8-A	32	ARM Cortex-A32		Application	
	64/32	ARM Cortex-A35,[48] ARM Cortex-A53, ARM Cortex-A57,[49] ARM Cortex-A72,[50] ARM Cortex-A73[51]	X-Gene, Nvidia Project Denver 1/2, Cavium Thunder X[52][53][54], AMD K12, Apple Cyclone/Typhoon/Twister/Hurricane/Zephyr/Monsoon/Mistral, Qualcomm Kryo, Samsung M1/M2 ("Mongoose")[55] /M3 ("Meerkat")	Application	[56][57]
ARMv8.1-A	64/32	TBA	ThunderX2[58]	Application	
ARMv8.2-A	64/32	ARM Cortex-A55,[59] ARM Cortex-A75,[60] ARM Cortex-A76[61], Cortex-A65, Neoverse E1, Neoverse E1, Cortex-A77	Nvidia Carmel, Samsung M4 ("Cheetah")[62]	Application	[63]
ARMv8.3-A	64/32	TBA	Apple Vortex/Tempest	Application	
ARMv8.4-A	64/32	TBA		Application	
ARMv8.5-A	64/32	TBA		Application	

1. ^ a b Although most datapaths and CPU registers in the early ARM processors were 32-bit, addressable memory was limited to 26 bits; with upper bits, then, used for status flags in the program counter register.
2. ^ a b c ARMv3 included a compatibility mode to support the 26-bit addresses of earlier versions of the architecture. This compatibility mode *optional* in ARMv4, and removed entirely in ARMv5.

ARM Evolution – Classic (~v5TE)

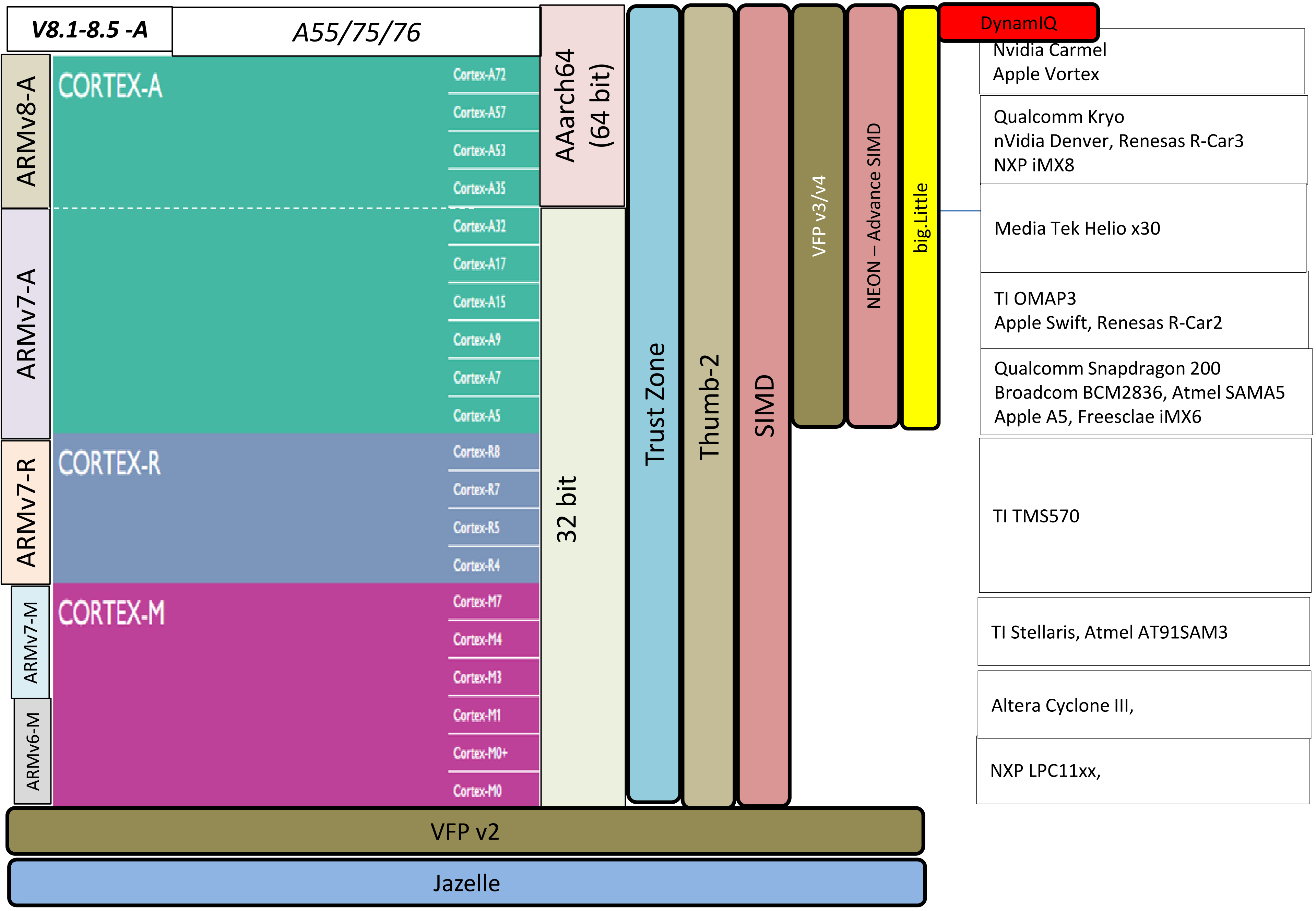
Version (Architecture)	Core (ARM)	Core (3rd party)	Key feature {Previous version +, from v2}
ARMv1	ARM1		1983-1985 26 bit addressing
ARMv2	ARM2		32 bit arithmetic instruction output Co-processor support
ARMv2a	ARM2aS, ARM3		On chip cache Load & Store instructions
ARMv3	ARM6, ARM600..		1990- (First design since ARM Limited established) MMU, Write-buffer 32 bit addressing mode, CPSR, SPSR
ARMv3M	ARM7, ARM700		64 bit signed/unsigned arithmetic instruction
ARMv4	StrongARM, ARM8		Half-word signed/unsigned Load/Store Privilege Mode
ARMv4T	ARM7TDMI, ARM710		Thumb instruction
ARMv5T	ARM9-S		BLX, CLZ, BRK instructions
ARM5TE	ARM10TDMI, ARM1020E	XScale	DSP instructions

ARM Architecture Categories – After v6

	Cortex-M	Cortex-R	Cortex-A
Profile	Microcontroller (MCU)	Real-Time	Application Processor
Example usage	Core controller in FPGA Smart Meter PLC/Sequencer	Hard Disk Controller Network Equipment Printer Co-processor inside Application SOC	Smart Phone/Tablet Computer Set-Top-Box Cloud/Edge computing
Mission	Simple & low cost Peripherals in same chip	Reduced latency for high priority task Exclusive HW resource for time critical	Feature rich & High performance Media processing with large data processing Power Management → Efficiency
Features	Limited Instruction sets 3~ stages of pipelines (from M4) SIMD/DSP/FPU Thumb2 available	**<u>Addition to Cortex-M</u>** 8 – 11 stages of pipelines Interrupt vector Availability of exclusive bus for reduced latency (i.e. LLPP, low latency peripheral port) → Direct read/write to IO (otherwise, read → modify → write) Out-of-Order Muti-core with lock-step (core independent, own program, own bus IF, Interrupt)	**<u>Addition to Cortex-R</u>** Widest range of instruction sets (NEON 128b, VP, etc) 11 stages or deeper pipelines Muti-core with big.Little → DynamIQ 64bit (v8~) → >4GB of RAM Virtualization MMU *Note: Cortex-R's lock-step is not suitable for application as it loses flexibility of dynamic core resource allocation with sacrifice of real-time

Note:
With evolution of ARM, this table gets less precise.
This is only benchmark of classifications between M/R/A

ARM Class Matrix with Technology Milestone



V8.1-8.5 -A

A55/75/76

ARMv8-A

ARMv7-A

ARMv7-R

ARMv7-M

ARMv6-M

CORTEX-A

Cortex-A72

Cortex-A57

Cortex-A53

Cortex-A35

Cortex-A32

Cortex-A17

Cortex-A15

Cortex-A9

Cortex-A7

Cortex-A5

CORTEX-R

Cortex-R8

Cortex-R7

Cortex-R5

Cortex-R4

CORTEX-M

Cortex-M7

Cortex-M4

Cortex-M3

Cortex-M1

Cortex-M0+

Cortex-M0

AArch64 (64 bit)

32 bit

Trust Zone

Thumb-2

SIMD

VFP v3/v4

NEON – Advance SIMD

big.Little

DynamIQ

Nvidia Carmel
Apple Vortex

Qualcomm Kryo
nVidia Denver, Renesas R-Car3
NXP iMX8

Media Tek Helio x30

TI OMAP3
Apple Swift, Renesas R-Car2

Qualcomm Snapdragon 200
Broadcom BCM2836, Atmel SAMA5
Apple A5, Freesclae iMX6

TI TMS570

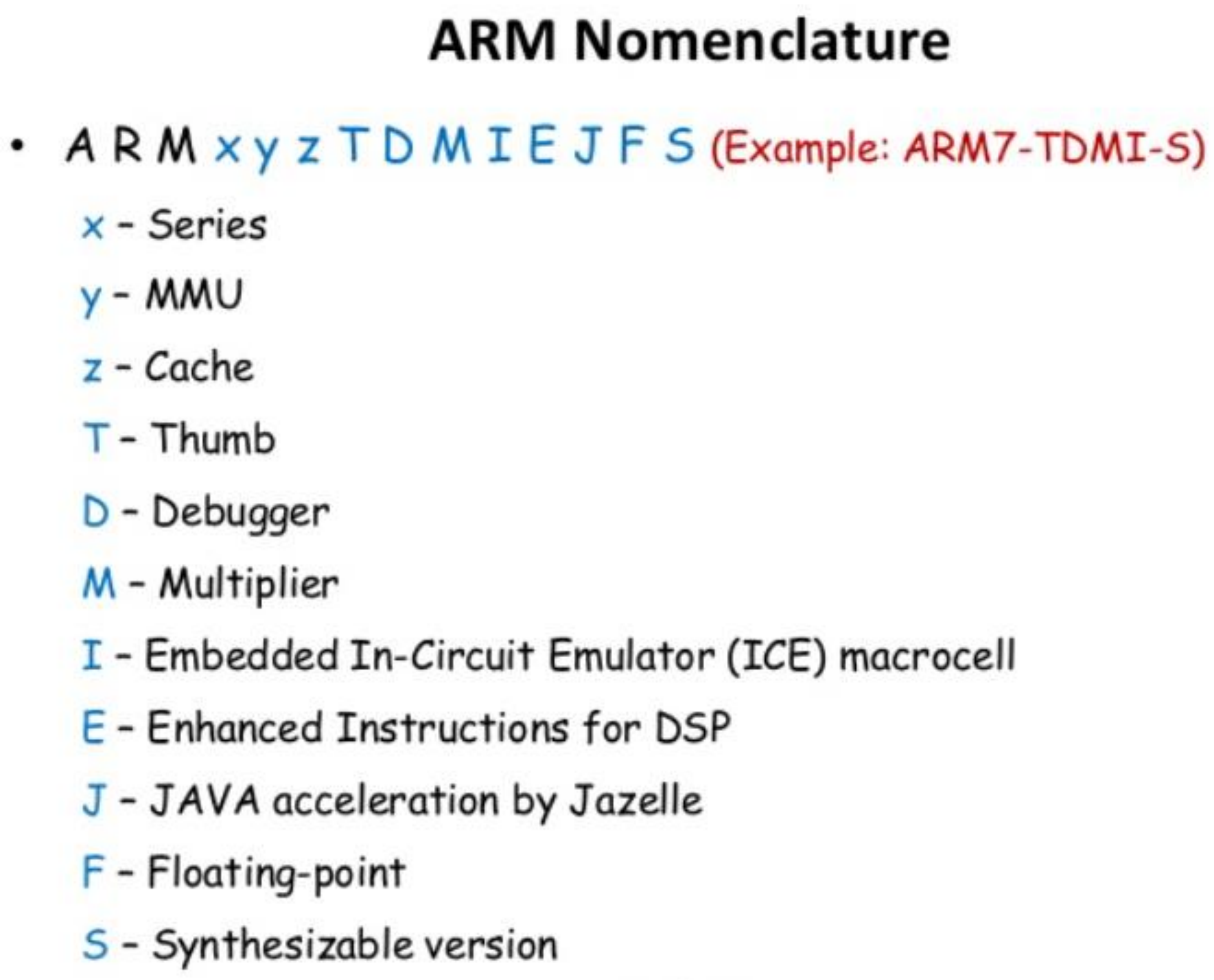
TI Stellaris, Atmel AT91SAM3

Altera Cyclone III,

NXP LPC11xx,

VFP v2

Jazelle



ARM Nomenclature

- A R M x y z T D M I E J F S (Example: ARM7-TDMI-S)

  x - Series

  y - MMU

  z - Cache

  T - Thumb

  D - Debugger

  M - Multiplier

  I - Embedded In-Circuit Emulator (ICE) macrocell

  E - Enhanced Instructions for DSP

  J - JAVA acceleration by Jazelle

  F - Floating-point

  S - Synthesizable version

Guidance to recognize features of ARM processor

Contents

ARM Overview

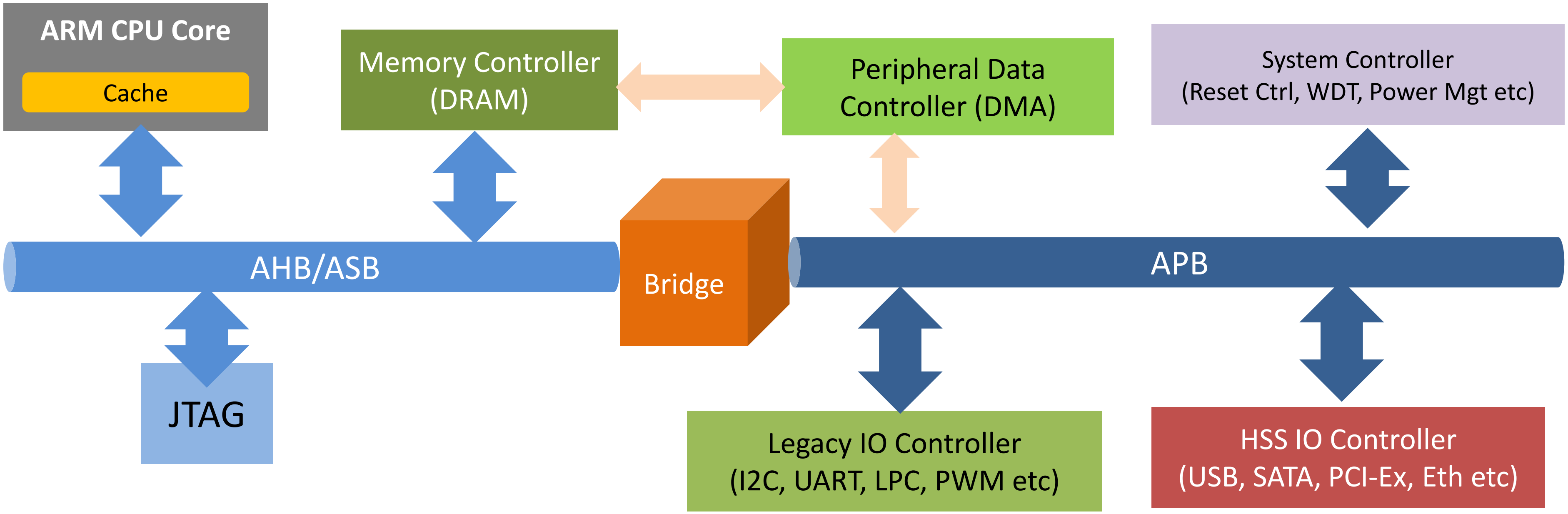
ARM Architecture Basic

- Architecture Overview
- General Registers in ARM
- Processor Mode and Privilege Level
- ARM Co-processor

ARM Key technologies

ARM Misc

ARM Architecture Block – High Level


ARM CPU Core
Cache
Memory Controller (DRAM)
Peripheral Data Controller (DMA)
System Controller (Reset Ctrl, WDT, Power Mgt etc)
AHB/ASB
Bridge
APB
JTAG
Legacy IO Controller (I2C, UART, LPC, PWM etc)
HSS IO Controller (USB, SATA, PCI-Ex, Eth etc)

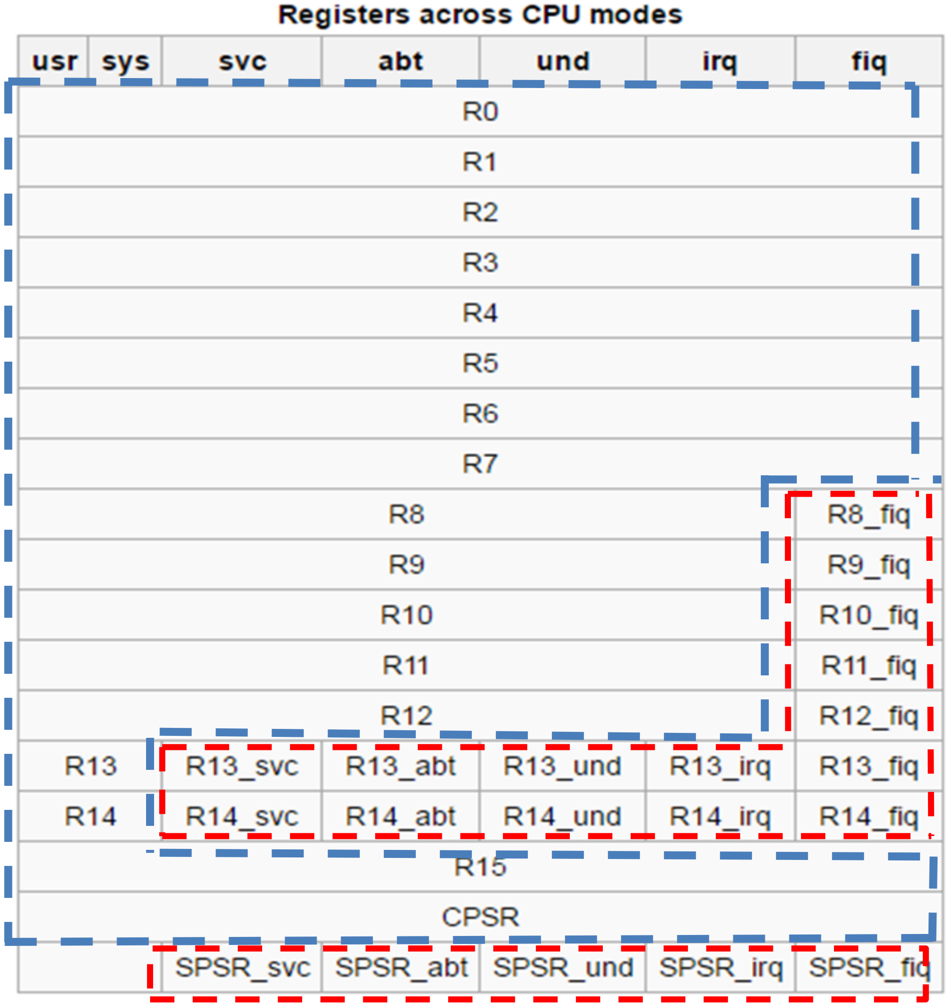
2 main segments split by bridge

- CPU core/memory: Run by AHB or ASB
- Peripheral IOs: Run by APB

CPU buses: Run with Advanced Microcontroller Bus Arch (AMBA)

- Advanced High Performance Bus (AHB) or ASB (Advanced System Bus)
  - Full Duplex with higher bandwidth
- Advanced Peripheral Bus
  - Lower BW
  - Focusing on memory/IO access
  - Simpler signal format

ARM Register Architecture



Registers across CPU modes

usr	sys	svc	abt	und	irq	fiq
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8_fiq
R9						R9_fiq
R10						R10_fiq
R11						R11_fiq
R12						R12_fiq
R13		R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14		R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15						
CPSR						
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

🟦 Accessible from user mode (Program user space)

🟥 Only system mode (non-user mode)

15 Registers

CPSR (Count Program Status Register)

SPSR (Saving program status register)

Each line with 32 bits

R0-R7: GPR for all CPU modes

R8-R12: User mode and FIQ mode use

R13/14: All modes may use

R15* program counter (PC)

In user mode:

R13 = stack point (sp) register

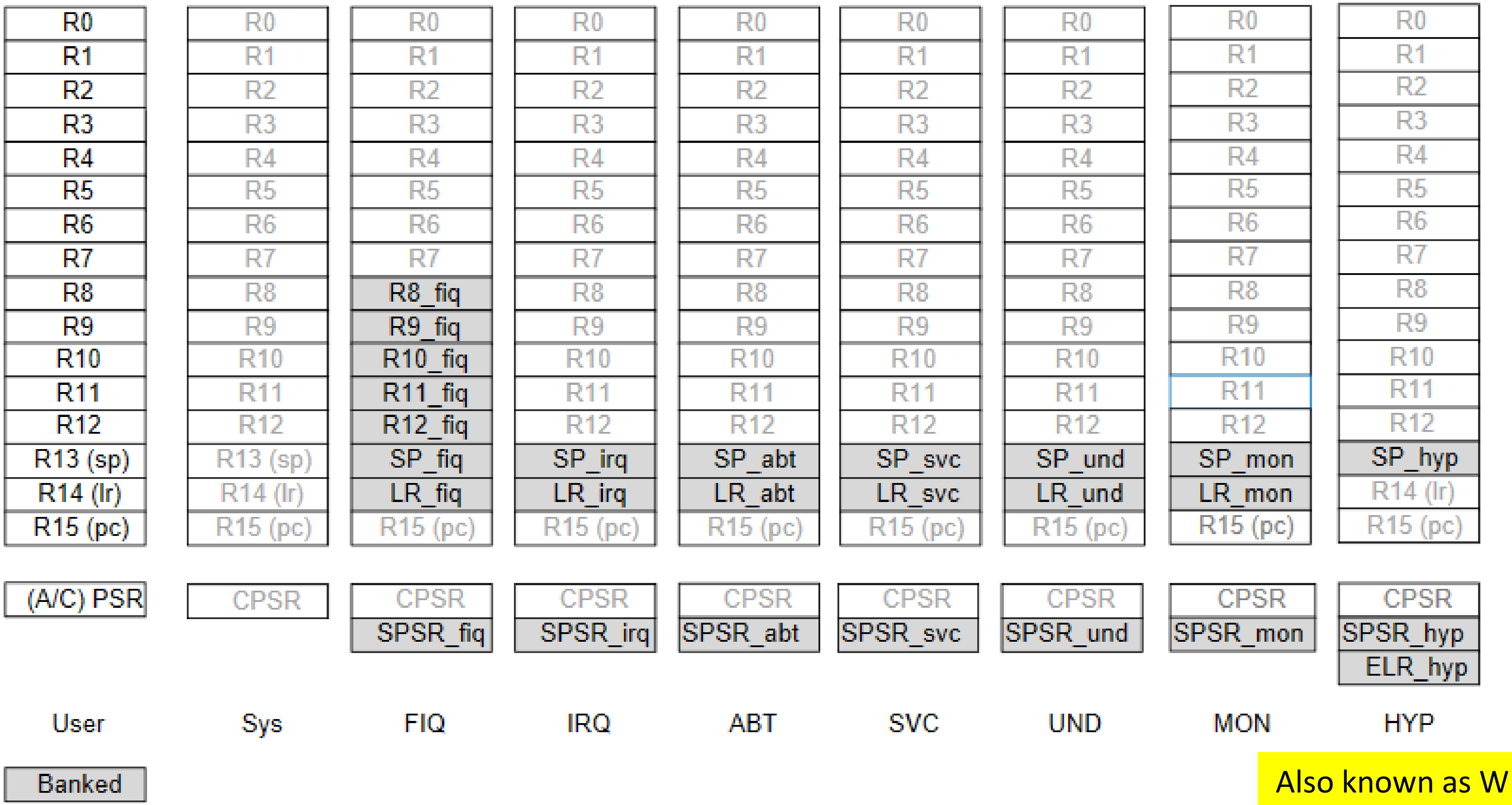
- Base address in memory where stack points starts in case of jumping to sub-routine

R14 = Link Register (lr)

- Storing return address of PC that it moves back at end of sub-routine

SPSR is used to store current CPSR value when exception happens and CPU gets into modes other than usr/sys
During exception modes, CPSR used to handle these modes operation
When returning to usr/sys mode, CPSR polls from SPSR to recover the original status

ARM Register – Looking from another angle



User	Sys	FIQ	IRQ	ABT	SVC	UND	MON	HYP
R0	R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8_fiq	R8	R8	R8	R8	R8	R8
R9	R9	R9_fiq	R9	R9	R9	R9	R9	R9
R10	R10	R10_fiq	R10	R10	R10	R10	R10	R10
R11	R11	R11_fiq	R11	R11	R11	R11	R11	R11
R12	R12	R12_fiq	R12	R12	R12	R12	R12	R12
R13 (sp)	R13 (sp)	SP_fiq	SP_irq	SP_abt	SP_svc	SP_und	SP_mon	SP_hyp
R14 (lr)	R14 (lr)	LR_fiq	LR_irq	LR_abt	LR_svc	LR_und	LR_mon	R14 (lr)
R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)	R15 (pc)
(A/C) PSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_fiq	SPSR_irq	SPSR_abt	SPSR_svc	SPSR_und	SPSR_mon	SPSR_hyp
								ELR_hyp

Banked

Also known as W Register

System Mode: Uses r0 – r12 & r13 (SP), r14(lr), r15 (PC), same as usr mode

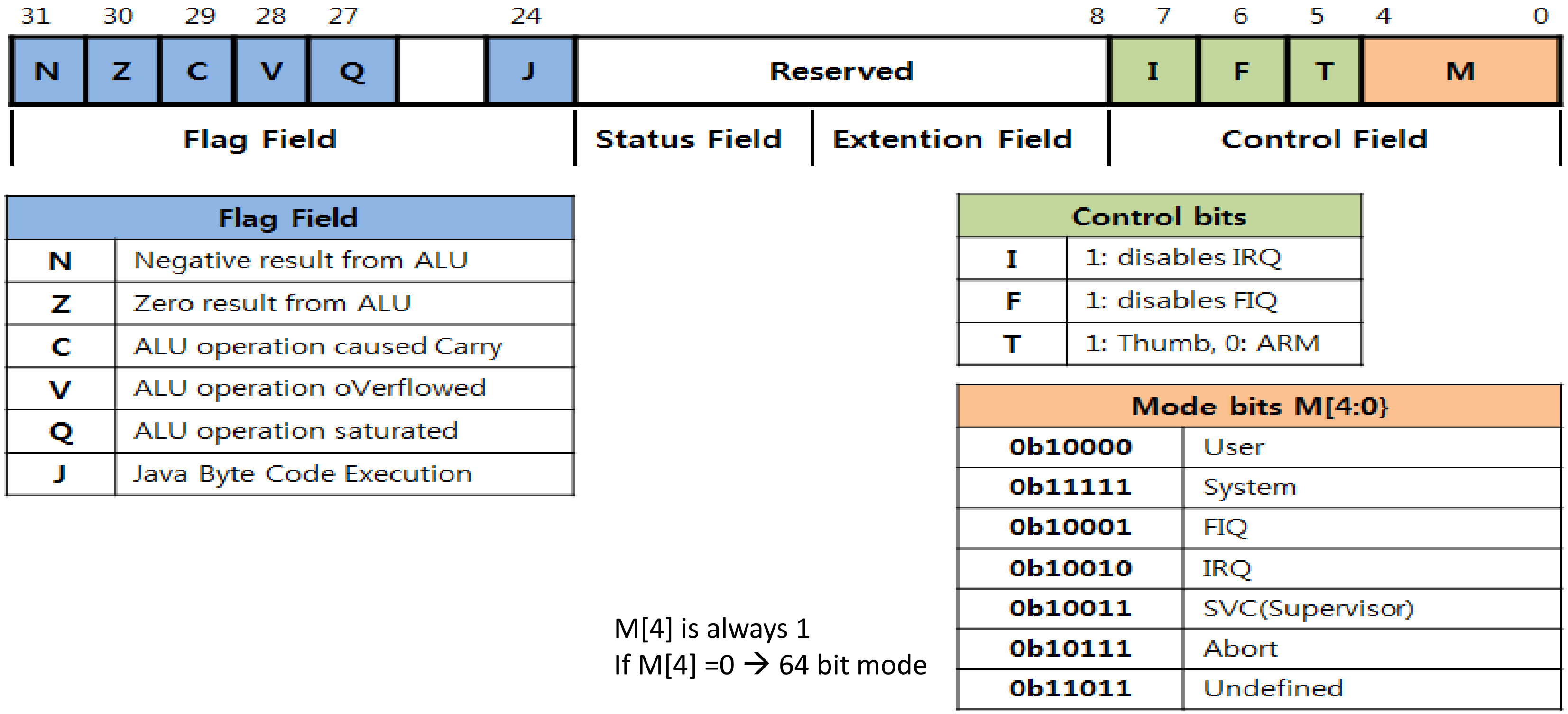
FIQ mode: r0-r7 and r15 for user mode & r8 –r14 used as registers for FIQ

IRQ/SVC/UND/ABT mode: r13 and r14 used as registers for these modes

SPSR used in exception/privilege modes other than usr to save CPSR values

For HYP: PC stored in ELR_hyp instead of r14.

Count Program Status Register (CPSR)



Flag Field	
N	Negative result from ALU
Z	Zero result from ALU
C	ALU operation caused Carry
V	ALU operation oVerflowed
Q	ALU operation saturated
J	Java Byte Code Execution

Control bits	
I	1: disables IRQ
F	1: disables FIQ
T	1: Thumb, 0: ARM

Mode bits M[4:0}	
0b10000	User
0b11111	System
0b10001	FIQ
0b10010	IRQ
0b10011	SVC(Supervisor)
0b10111	Abort
0b11011	Undefined

M[4] is always 1
If M[4] =0 → 64 bit mode

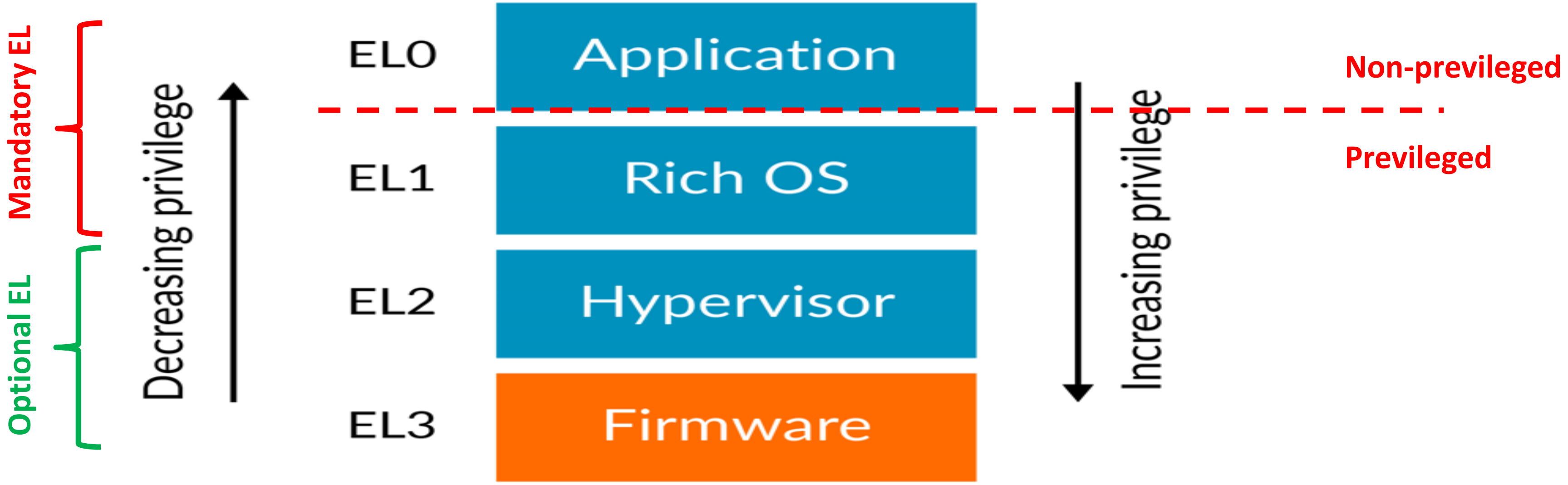
Mode bits [4:0]:
When NOT “user(b0)”/”system(31b)” ➔ Exception mode ➔ Write current CPSR value to SPSR

Additions not shown in diagram:
Mode bits: Hypervisor mode [0b11010], Monitor mode [0b10110]
{New] Bit 9, “E”: Endianness. If 0 → Little endian, else big endian
[ARMv8.0~] Bit23 = SSBS (Speculative Store Bypass Safe)
[ARMv8.1~] Bit22 = PAN (Privileged Access Never)
[ARMv8.4~] Bit21 = DIT (Data Independent Timing)

CPU Mode of Operation – Bit[4:0] of CPSR

Mode	mnemonic	ARMv~	CPSR[4:0]	Privilege	Statement
User	usr	All	0b10000	EL0	Only non-privileged mode ARM usual program execution state for most app program When interrupted, switch to svc (privileged)
System	sys	V4~	0b11111	EL1	Privileged mode but still using the same register sets as user mode. Introduced to overcome problem with calling multiple sub-routines (rick of overwriting PC in r14)
Faster Interrupt Request	fiq	All	0b10001	EL2	Higher priority interrupt and can mask other INT. 7 registers (r8-r14) used to avoid pushing out of handler & faster context switching Can be executed only with ARM assembly
Interrupt Request	Irq	All	0b10010	EL2	General purpose interrupt (i.e. periodic timers) Only r14 and r15 to return to main routine (usr) afterward
Supervisor	Svc	All	0b10011	EL2	Mode when CPU is reset or SW interrupt received
Abort	Abt	All	0b10111	EL2	When abort happened (i.e. unsuccessful memory data access) → one of exception mode
Undefined	und	All	0b11011	EL2	When undefined instruction is given and cannot handle → exception occurs
Hypervisor	hyp	If hypervisor supported	0b11010	EL2	When switching between guest OS kernels
Monitor	monitor	If security supported	0b10110	EL3	Associated with Trusted Zone When switching between secure and non-secure mode Safely saving state when leaving secure space operation

ARM – Privilege Level

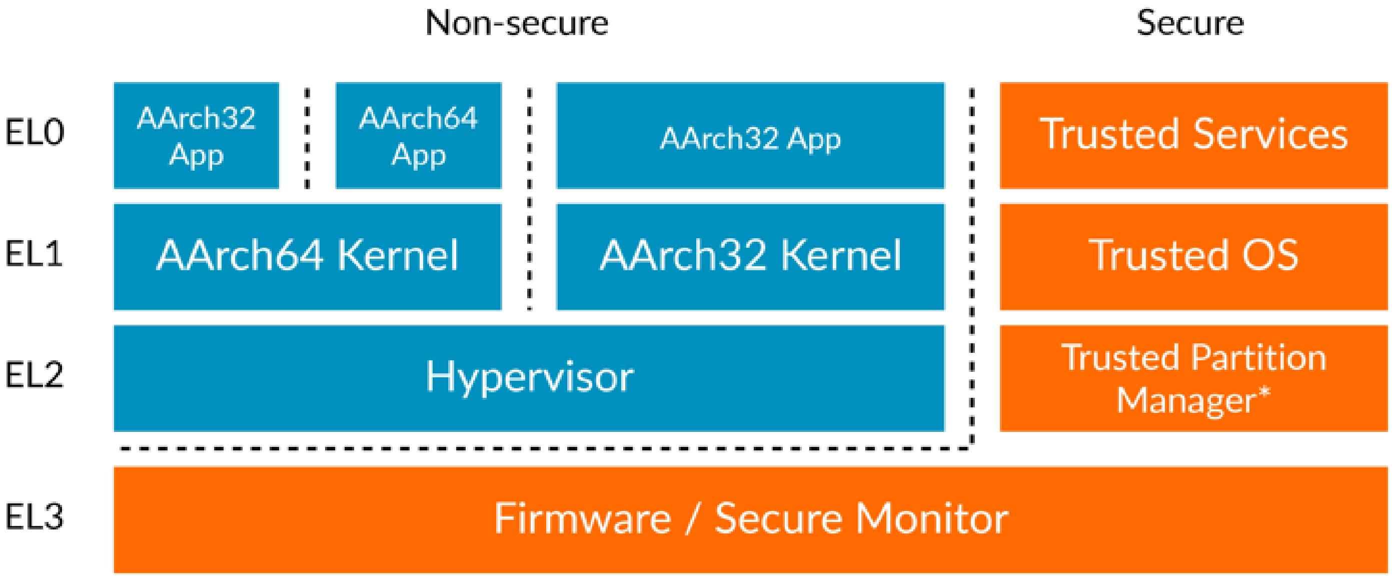


Privilege Level (PL) Definition {Also known as Exception Level}:
Level of access right to system and CPU resource allocated to software.
Higher number → More privilege
i.e) OS kernel has more right of configuration than application program

Type of privilege:

1. Memory privilege
   - MMU controls the read/write access of memory region based on PL
2. Register Access
   - Register setting is granted with PL. Some registers are not accessible with lower PL
   - To indicate accessibility of PL, register names may include suffix to indicate PL, such as SCTLR_EL1
3. Instruction sets
   - Some instruction sets are not granted to lower PL. For example, TZ is granted to specific

ARM – Privilege Level – Cont.



Non-secure

Secure

EL0

AArch32 App

AArch64 App

AArch32 App

Trusted Services

EL1

AArch64 Kernel

AArch32 Kernel

Trusted OS

EL2

Hypervisor

Trusted Partition Manager*

EL3

Firmware / Secure Monitor

Hypervisor at PL2 can switch between guest OSs (PL2)

Secure State (Trust Zone)

- Most privilege level (EL3) having access to most resource etc
- APP, OS, partition running on security zone have PL3 equivalent privilege instead of EL0-2.

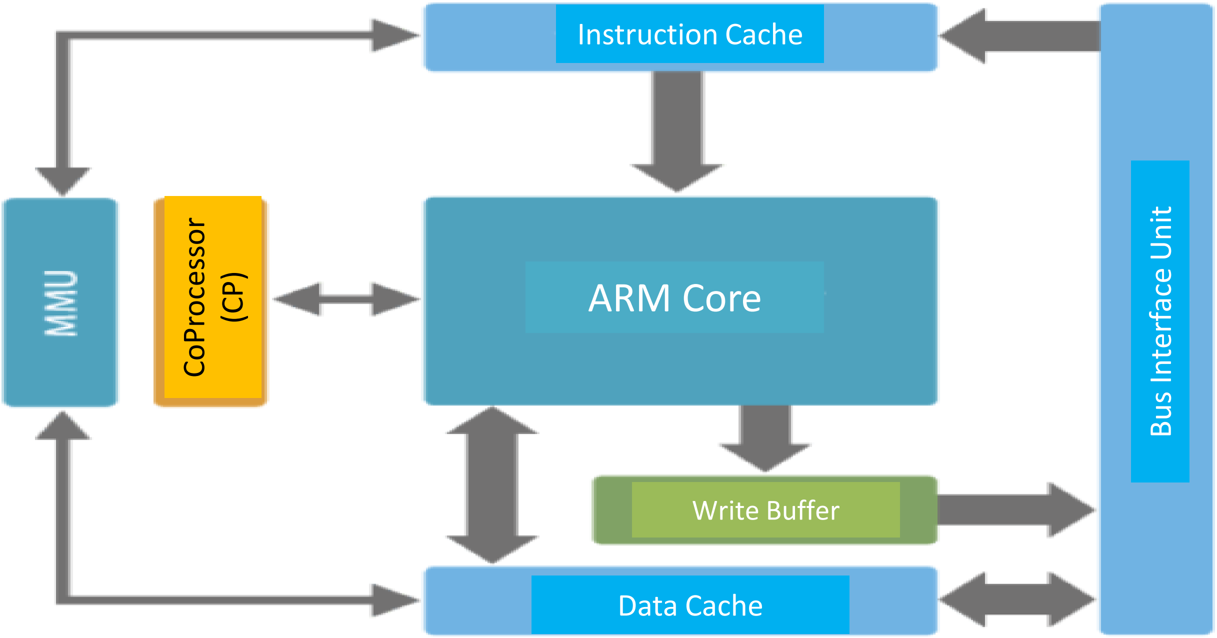
Non-Secure State (Blue in above diagram)

- Access only to limited (non-secure) resource (memory/register) and takes only non secure interrupt

Trust Zone explains how transition between Non-secure and Secure is done

Note: 64 bit execution state (AArch64 App) cannot run on AArch32 kernel

ARM Co-Processor



ARM Architecture uses Co-processors for extensive functionality (Except ARM-M)
Co-Processor uses dedicated instructions {MRC, MCR} and have own registers

Access to co-processor is outside of ARM core memory map
➔ (i.e. with 32 bit) part of 4GB mapping is not used by co-processor

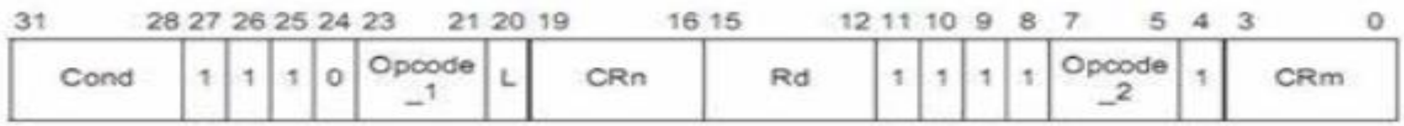
Example of co-processor (CP15): There are 16 co-processors in total

- CP15: For system configuration and control
- CP11: Double precision floating-point
- CP10: Single-precision floating point

} NEON uses both

ARM CP Access Instructions

CP Access Instruction Structure



31 28	27	26	25	24	23 21	20	19 16	15 12	11	10	9	8	7 5	4	3 0
Cond	1	1	1	0	Opcode _1	L	CRn	Rd	1	1	1	1	Opcode _2	1	CRm

Parameter	Contents
Cond	{Optional} Conditional code
CoProc	Name of co-processor which instruction is executed {For CP15, P15}
opcode1	3 bit opcode for co-processor. Mapped to CRn. (i.e. if 0 → load from or store to co-processor)
opcode2	{Optional} 3 bit opcode for co-processor. Mapped to CRm
Rd	Registers in ARM core which it is written into (MRC)/read form (MCR)
CRn	Co-processor register
CRm	Co-processor register

2 types of instructions to access to CP

- MRC: Reading from CP registers, *MRC{ cond } coproc,# opcode1,Rd,CRn,CRm {, # opcode2 }*
- MCR: Writing into CP registers, *MCR{ cond } coproc,#opcode1,Rd,CRn,CRm {, # opcode2 }*

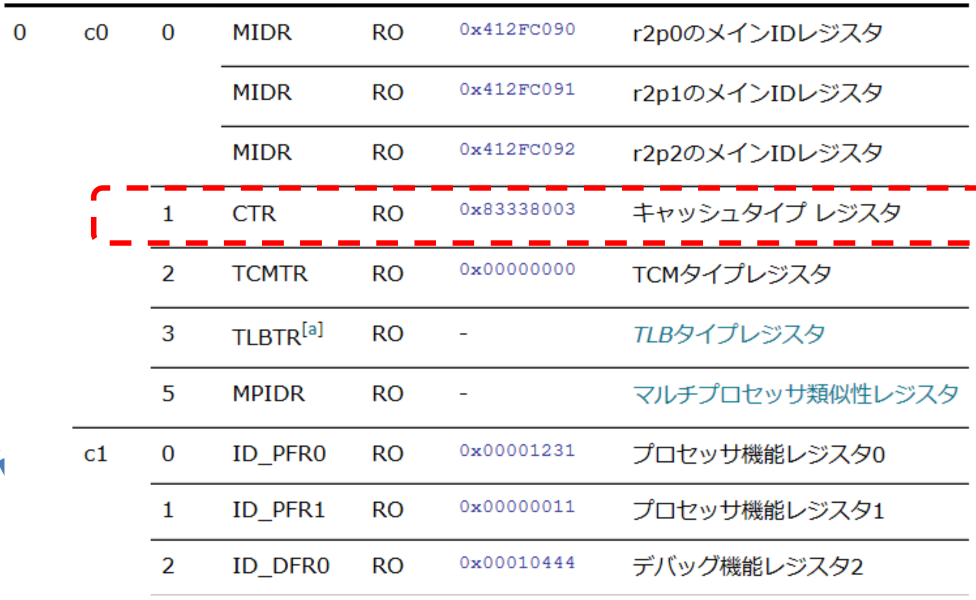
Example: MRC p15,0,r0,c1,c0,0

Reading from CP15 (P15 and 0) into ARM registe @ r0 (r0). Registers to load from are C1 (config) and C0 (ID)

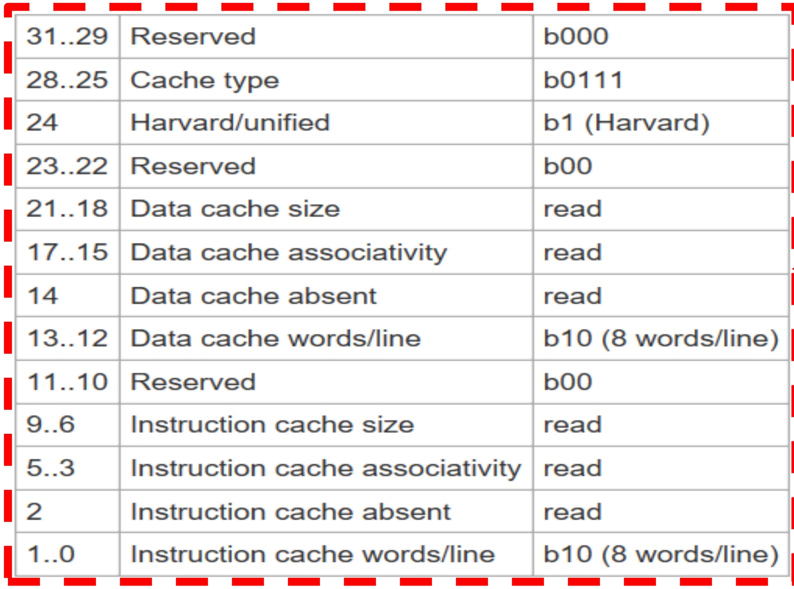
CP Example – CP15

CoProc = P15

Register (CRn)	Parameter
0	ID register
1	Config
2	Cache Control
3	Write Buffer
5	Access Permit
6	Base Address & Size
7	Cache Op
9	Cache Lock
15	Test



Op1	CRm	Op2	Name	Type	Default Value	Description (Sorry in Japanese)
0	c0	0	MIDR	RO	0x412FC090	r2p0のメインIDレジスタ
			MIDR	RO	0x412FC091	r2p1のメインIDレジスタ
			MIDR	RO	0x412FC092	r2p2のメインIDレジスタ
		1	CTR	RO	0x83338003	キャッシュタイプ レジスタ
		2	TCMTR	RO	0x00000000	TCMタイプレジスタ
		3	TLBTR[a]	RO	-	TLBタイプレジスタ
		5	MPIDR	RO	-	マルチプロセッサ類似性レジスタ
	c1	0	ID_PFR0	RO	0x00001231	プロセッサ機能レジスタ0
		1	ID_PFR1	RO	0x00000011	プロセッサ機能レジスタ1
		2	ID_DFR0	RO	0x00010444	デバッグ機能レジスタ2



31..29	Reserved	b000
28..25	Cache type	b0111
24	Harvard/unified	b1 (Harvard)
23..22	Reserved	b00
21..18	Data cache size	read
17..15	Data cache associativity	read
14	Data cache absent	read
13..12	Data cache words/line	b10 (8 words/line)
11..10	Reserved	b00
9..6	Instruction cache size	read
5..3	Instruction cache associativity	read
2	Instruction cache absent	read
1..0	Instruction cache words/line	b10 (8 words/line)

Registered looked in the order of

1. CRn: First level co-processorregister
2. Opcode 1
3. CRm: Second level co-processor
4. Opcode2

Contents

ARM Overview

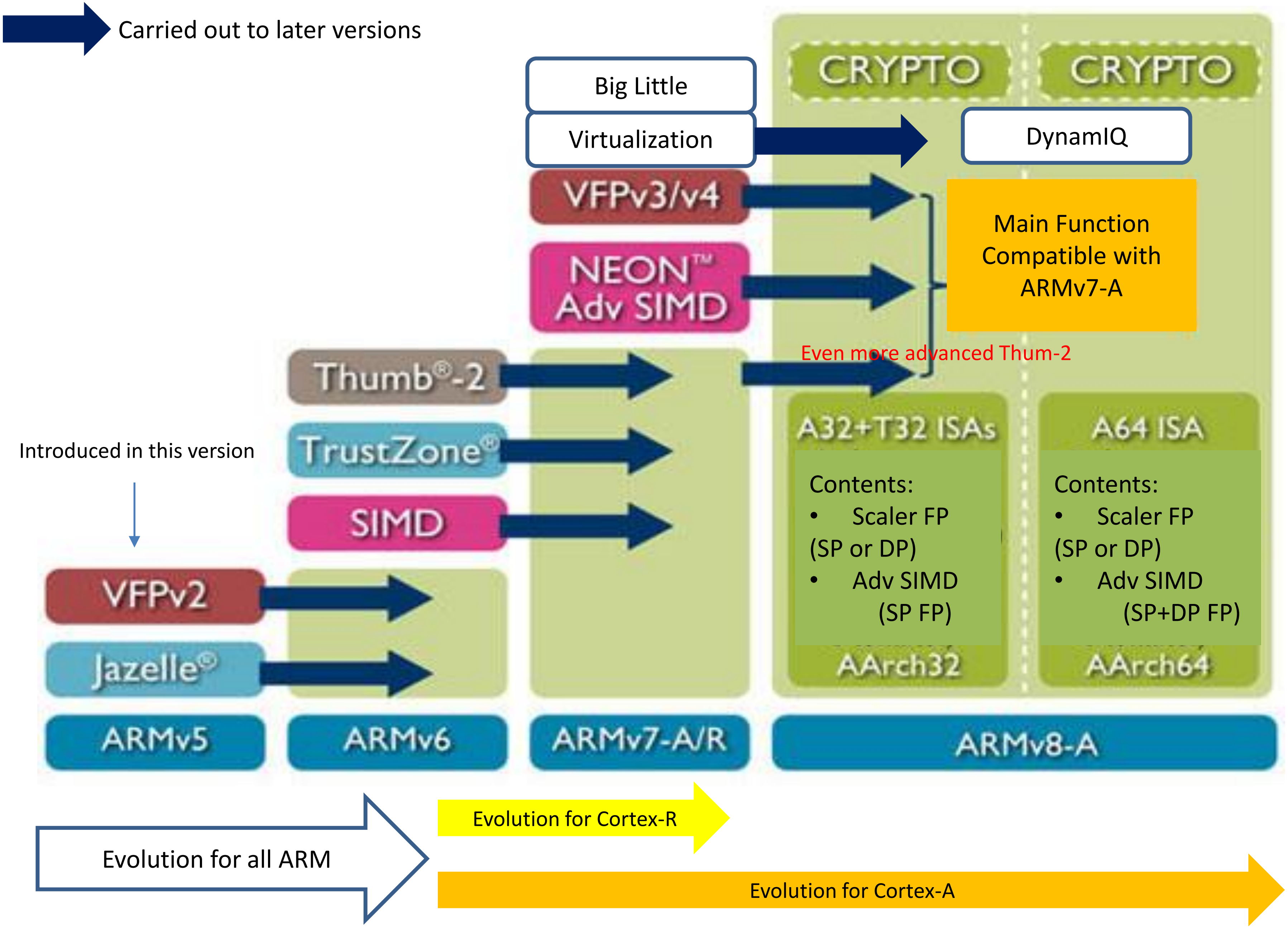
ARM Architecture Basic

ARM Key technologies

- AArch64 (64 bit mode)
- VFP
- ARM SIMD & NEON
- ARM Virtualization
- TEE and ARM Trusted Zone
- Thumb Instruction
- Jazelle
- BigLittle & DynamIQ

ARM Misc

ARM Key Technologies Evolutions



Carried out to later versions

Big Little

Virtualization

DynamIQ

Main Function Compatible with ARMv7-A

Even more advanced Thum-2

Introduced in this version

Contents:
- Scaler FP (SP or DP)
- Adv SIMD (SP FP)

Contents:
- Scaler FP (SP or DP)
- Adv SIMD (SP+DP FP)

Evolution for all ARM

Evolution for Cortex-R

Evolution for Cortex-A

ARM 64 bit – AArch64 Overview

Available from ARMv8 (From Cortex-A57)

64 bit execution state (A64 instructions)

Wider memory access ➔ AArch32 limited to 4GB

To do AArch64, mode switch is necessary

(Cannot execute A32 or T32 instructions)

Registers are extended to 64 bit [X-registers]

Can access to both 64bit and 32bit registers

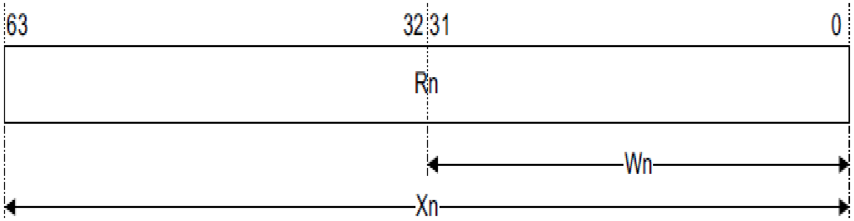
AArch32 cannot use X-Register[63:32] ➔ stuffed with 0

MMU:

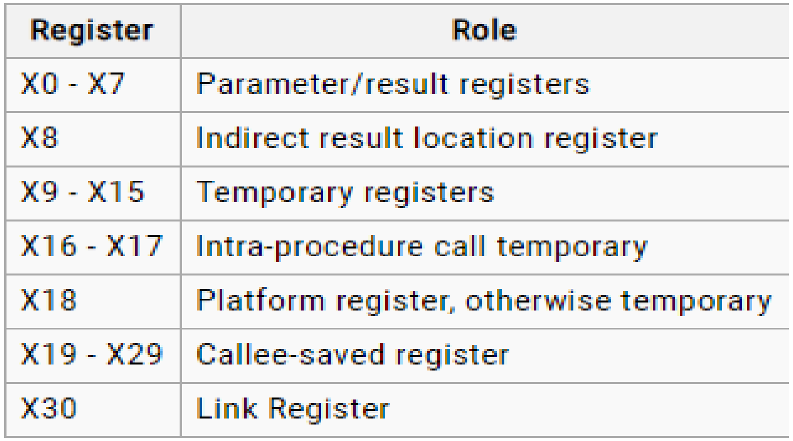
Virtual address up to 48 bits (8bits for tag and 40 bits for TBR)

ARM – AArch64 Registers (GPR)

GPR allocation between AARch64/32(ARMv8~)

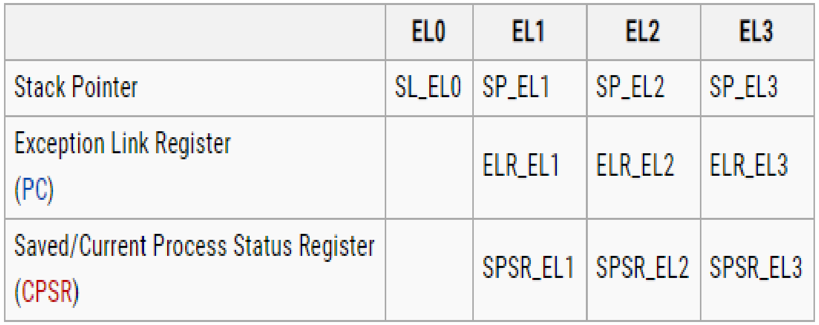


Convention of AArch64 GPR (X Register)



Register	Role
X0 - X7	Parameter/result registers
X8	Indirect result location register
X9 - X15	Temporary registers
X16 - X17	Intra-procedure call temporary
X18	Platform register, otherwise temporary
X19 - X29	Callee-saved register
X30	Link Register

Bank Registers for Exception/Privilege states in AArch64



	EL0	EL1	EL2	EL3
Stack Pointer	SL_EL0	SP_EL1	SP_EL2	SP_EL3
Exception Link Register (PC)		ELR_EL1	ELR_EL2	ELR_EL3
Saved/Current Process Status Register (CPSR)		SPSR_EL1	SPSR_EL2	SPSR_EL3

GPR Allocation {ARMv8~}:

AArch32 Mode:

- Use W[31:0] == X[31:0]
- X[63:32] stuffed with 0

AArch64 Mode:

- Use X[64:0]

AArch64 handler code can request access to AArch32 registers with mapping to 32b GPR

Until ARMv7, GPR called R[31:0]

Convention of X Registers

- Usually use X0-X18 and X30
- Each has 64bits

Exception Registers

- SP exists in all levels to indicate the base address of to store in case of migrating to higher EL
- Mapping with 64 → 32 shift
  - SPSR_svc maps to SPSR_EL1.
  - SPSR_hyp maps to SPSR_EL2.
  - ELR_hyp maps to ELR_EL2.

ARM – AArch64 Instructions

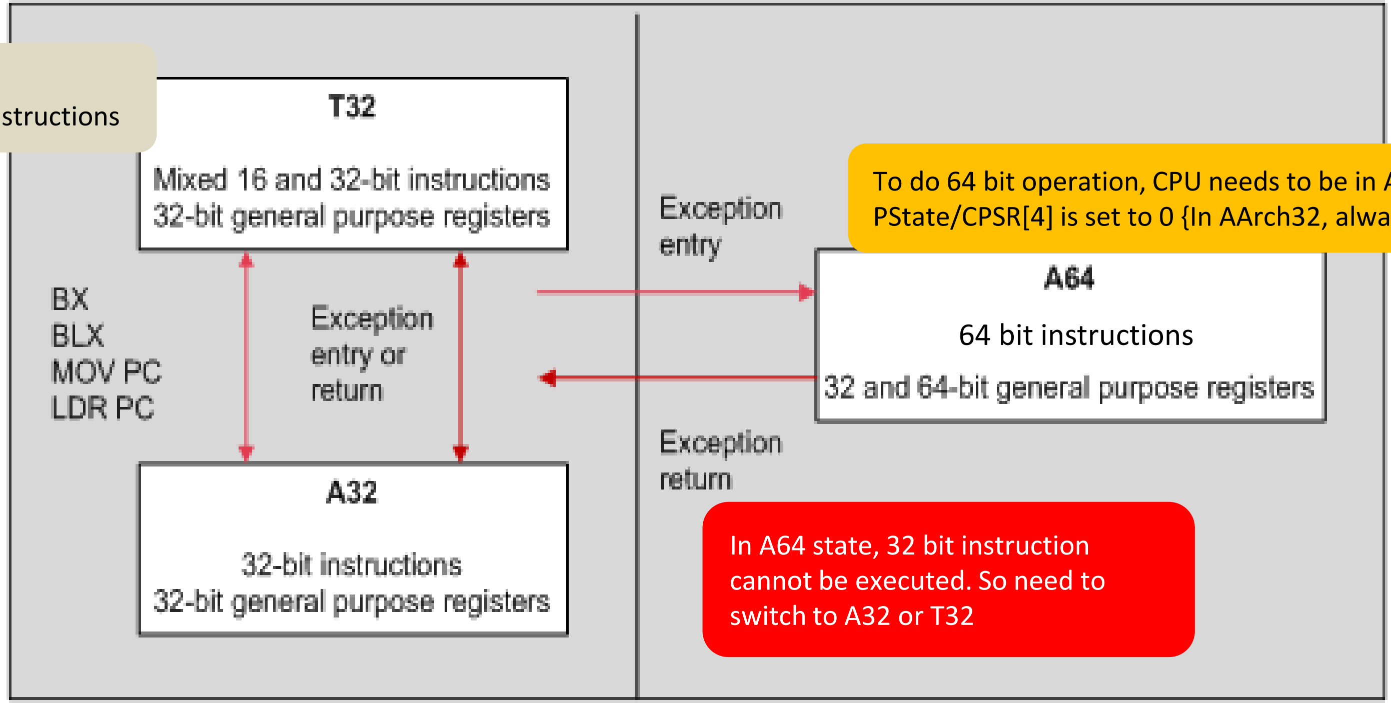
Advancement with AArch 64 from 32:

- 64 bit wide integer registers access
- 64bit data operation
- 64 bit sized addressing to memory

In case of A32/T32 → AArch32 & backward compatible to ~ARMv7

- Note: New 32b instructions introduced in ARMv8 cannot be carried back to ARMv7

Switching between AArch32 (A32/T32) and AArch64 (A64)



T32
Mixed 16 and 32-bit instructions
32-bit general purpose registers
BX
BLX
MOV PC
LDR PC
Exception entry or return
A32
32-bit instructions
32-bit general purpose registers
Exception entry
Exception return
A64
64 bit instructions
32 and 64-bit general purpose registers

T32: Thumb-2
Mix of 16 and 32 bits instructions

To do 64 bit operation, CPU needs to be in A64 state PState/CPSR[4] is set to 0 {In AArch32, always 1}

In A64 state, 32 bit instruction cannot be executed. So need to switch to A32 or T32

ARM VFP (Vector Floating Point) Instructions

FPU co-processor extension to ARM (~ARMv7)

***No more co-processor from ARMv8 {Likewise to Intel ® Pentium??}*

VFP is for short vector mode → Performance not as good as SIMD

- VFP benefit: when dealing with only single FP
  - CPI is shorter → faster output
  - Smaller code size

Version	#FPU Regs/bit	ARM version	Comment
VFPv2	16/64	v5TEJ, v6	
VFPv3/3-D32	32/64	v7 (A8, A9)	Mostly backward compatible with VFPv2
VFPv3-D16	16/64	R4, R5 and Tegra 2 (A9)	
VFPv3-F16			Half-precision (16b)
VFPv4/4-D32	32/64	v7(A12, A15) A7 if used with NEON	
VFPv4-D16	16/64	A5, A7 (not used with NEON)	
VFPv5-D16-M		M7	(if single&double precision option available)

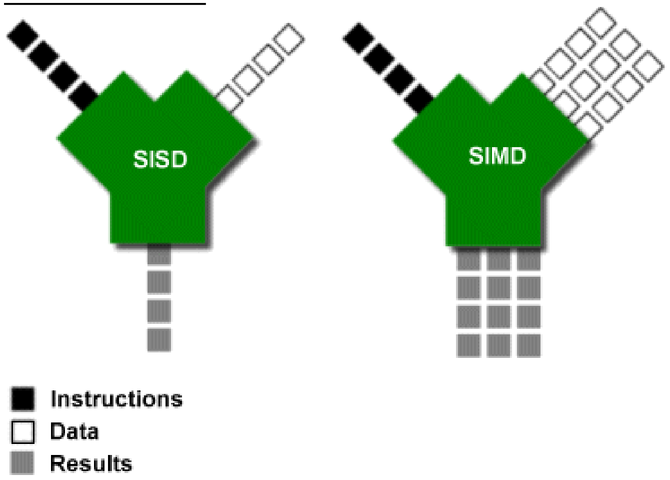
VFP is implemented based on VFP registers such as , MVFR0 {A32}, MVFR0_EL1{A64}
(Details skipped in this document but available in ARM Ref Manual)

ARM VFP Instruction Example

Syntax	Definition of Definition	Coding Example
VABS	Floating-point absolute value	VABS{*cond*}.F32 *Sd*, *Sm* d = destination, m = operand s → single word, if d → double word (64b) F32 → FP with 32 bits (if double word →F64)
VADD	Floating-point add	VADD{*cond*}.F32 {*Sd*}, *Sn*, *Sm* ➔ Value @ d = value @ n + value @ m all with 32b FP
VSUB	Floating-point subtract	VSUB{*cond*}.F32 {*Sd*}, *Sn*, *Sm*
VCMP	Floating-point compare	VCMP{*cond*}.F32 *Sd*, *Sm* Compare values at d and m having 32bit m can be replaced with immediate #A, A = integer?
VCVT	Convert between integer to FP	VCVT{*cond*}.*typeQd*, *Qm* {, #*fbits*} Type: i.e. S32.F32 → 32b FP to 32b integer
VDIV	Floating-point divide	VDIV{*cond*}.F32 {*Sd*}, *Sn*, *Sm* ➔ Value @ d = value @ n / value @ m, all with 32b FP
VMUL	Floating-point multiply	VMUL{*cond*}.F32 {*Sd,*} *Sn*, *Sm* ➔ With word (32b) value @ d = value @ n * value @ m
VMLA	Floating-point multiply accumulate	VMLA{cond}.datatype {Qd}, Qn, Qm ➔ With q-word (128b) value @ d = value @ n * value @ m Data type: I8, I16, I32 or F32 → for VFP must be F32
VNEG	Floating-point negate	VNEG{cond}.F64 Dd, Dm ➔ Swapping sign bit of value @ m and load its value to d
VMLS	Floating-point multiply subtract	VMLS{cond}.F32 Sd, Sn, Sm ➔ New value @d = value @n * value @m – current value @d

ARM SIMD Instructions - Overview

SISD vs SIMD

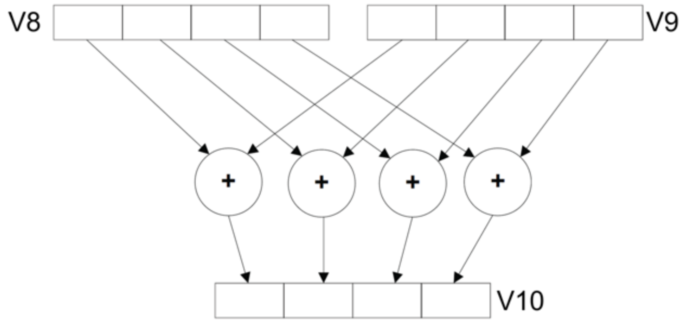


SISD Instruction Example

```
ADD w0, w0, w5
ADD w1, w1, w6
ADD w2, w2, w7
ADD w3, w3, w8
```

SIMD Instruction Example

```
ADD V10.4S, V8.4S, V9.4S
```



SISD vs SIMD

SISD:

- Single operand (max 2) + destination register per instruction line
- Need multiple instructions to generate chains of parallel data ➔ longer latency

SIMD:

- Multiple operands in a single instructions
- Capable of processing array of data at once
  - Operand size: 8/16/32/64 bits (*64b only for AArch64)

Efficient use of register space

- i.e. 128 bit space available vs operand has only 16 bits

SISD Instruction Example (Left)

- Need 4 single instructions to get 4 output series (w0, w1, w2, w3)
- Need 4 x CPI latency + possible additional instructions to integrate them

SIMD Instruction Example (Left)

- Only single instruction needed, instead of 4, to update in register at v10 containing 4 arrays of data
  - v10 corresponds to integral of w0-w3 in SID case
- SIMD instruction is such to add each chunk of operands from both sides
- Definitely shorter latency till getting expected output (SIMD CPI would be longer than SISD but definitely faster than having multiple)

ARM SIMD Instructions – Use Case Example

Flexible video transcoding

Enhanced captured video

Gaming, advanced user interface

Machine and deep learning

Computer vision AR/VR

Speech recognition, advanced audio processing

ARM SIMD

- SIMD: ARMv6~ (8/16/32/*64 bit) *AArch64 only
- NEON: ARMv7 A/*R ~. ~128 bits *Cortex-R53
- Helium* ARMv8.1-M

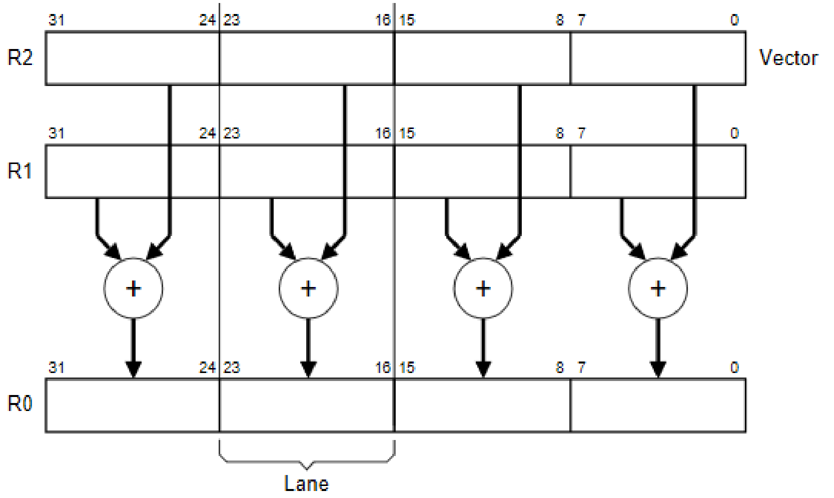
ARM SIMD function - Example

- Math Function: vector and matrix
- Signal Processing: FFT, FIR, IIR
- Image Processing: image resize/rotate

Replacement and offload of GPU/DSP engines (If high performance not required)

ARM SIMD vs NEON

ARM SIMD Example (4 way 8bit integer add operarion)



ARM SIMD

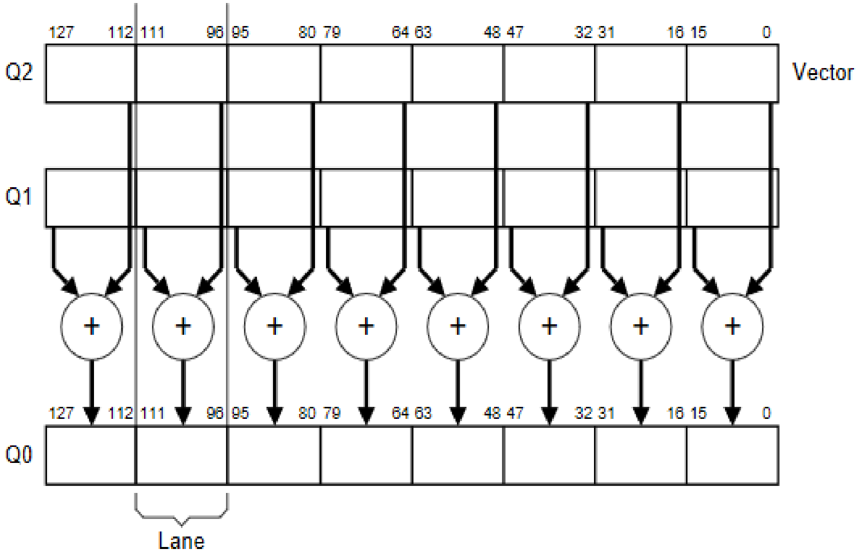
ARMv6~

Packing 8 or 16 bit into 32 bit GPR

Left example;

- *UADD8 R0, R1, R2*
- UADD8 → 8 bit unsigned, thus 32/8 = 4 ways packed into GPRs, R1 and R2
- Result placed into R0

NEON Example (8 way 16 bit integer add operarion)



NEON

ARMv7 ~, as optional SIMD extension to ARMv7-A/R

Data Types:

- Integer: 8bit (i.e., I.8), 16 bit, 32 bit, 64 bit
- FP: 32bit (i.e. F.32)

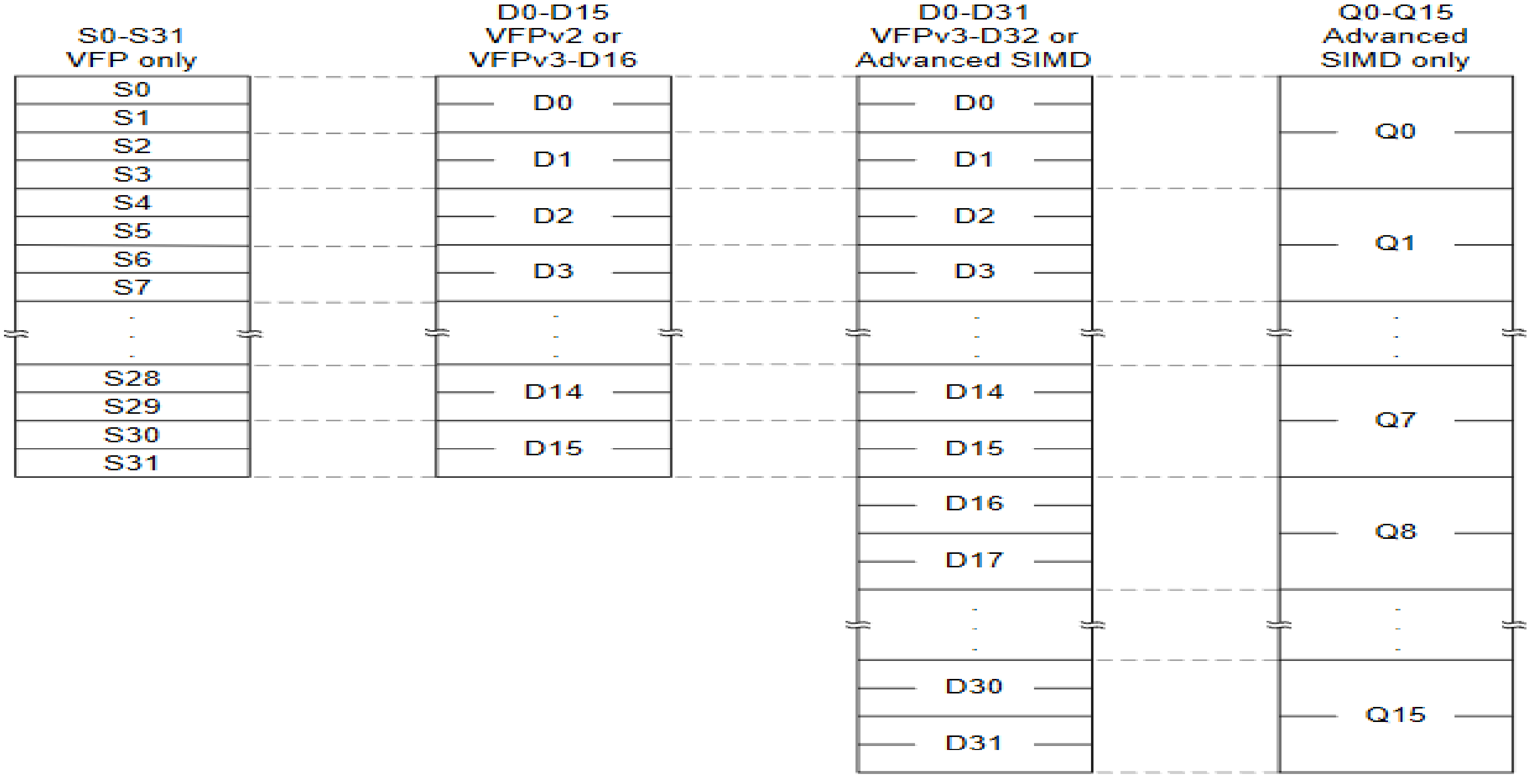
Storing in registers, 64b D (doubleword) vector register or 128b Q (quadword) vector register

Left Example;

- *VADD.I16 Q0, Q1, Q2*
- VADD.I16 → 16 bit integer, thus 128/16 = 8 ways packed into GPRs, Q1 and Q2
- Result placed into Q0

Note: If 64 bit register → D instead of Q

VFP & NEON Registers



32 x 32 bit registers (S0 – S31): VFP only

32 x 64 bit registers (D0-D16): Can be shared between VFP and NEON

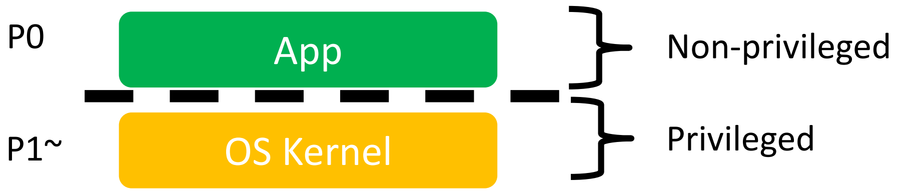
- VFP may use only D0 – D15
- D16-D31: exclusive for Advanced SIMD (NEON) and VFPv3

16 x 128 bit registers (Q0 – Q15): NEON only

Use of registers are up to selection of instructions (not transparent to App SW)

Virtualization/Hypervisor with ARM - Overview

With no virtual machine/Single OS



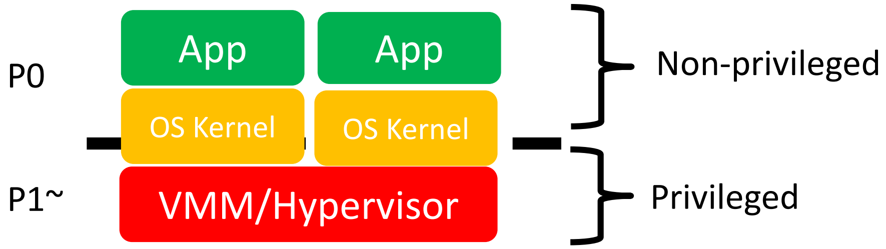
P0 = Non-privileged level

- Limited access to resource/memory
- Users has limited access

P1 = Privileged level

- Wider/full access to resource/memory
- Higher capability & performance

With virtual machine/Gues OSs without VT



VMM/Hypervisor occupies privileged level

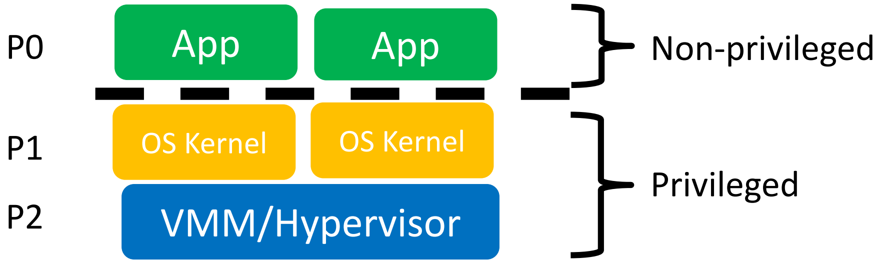
- Handling switch between guest OSs
- Resource/memory allocation to guest OSs

OS kernel pushed out to non-privileged level

Issue:

Decreased performance of OS/kernel and applications running on top due to limited access to resource/memory

With virtual machine/Gues OSs with VT (ARM Solution)



Mechanism to handle virtual machines (guest OSs) with minimum performance impact

Solution:

Provide another level in privileged region

Split of privileged region

- OS/kernel (P1): Full access to resource/memory
- HV (P2): Focuses on instructions to implement hypervisor (other privileged tasks given to P1)

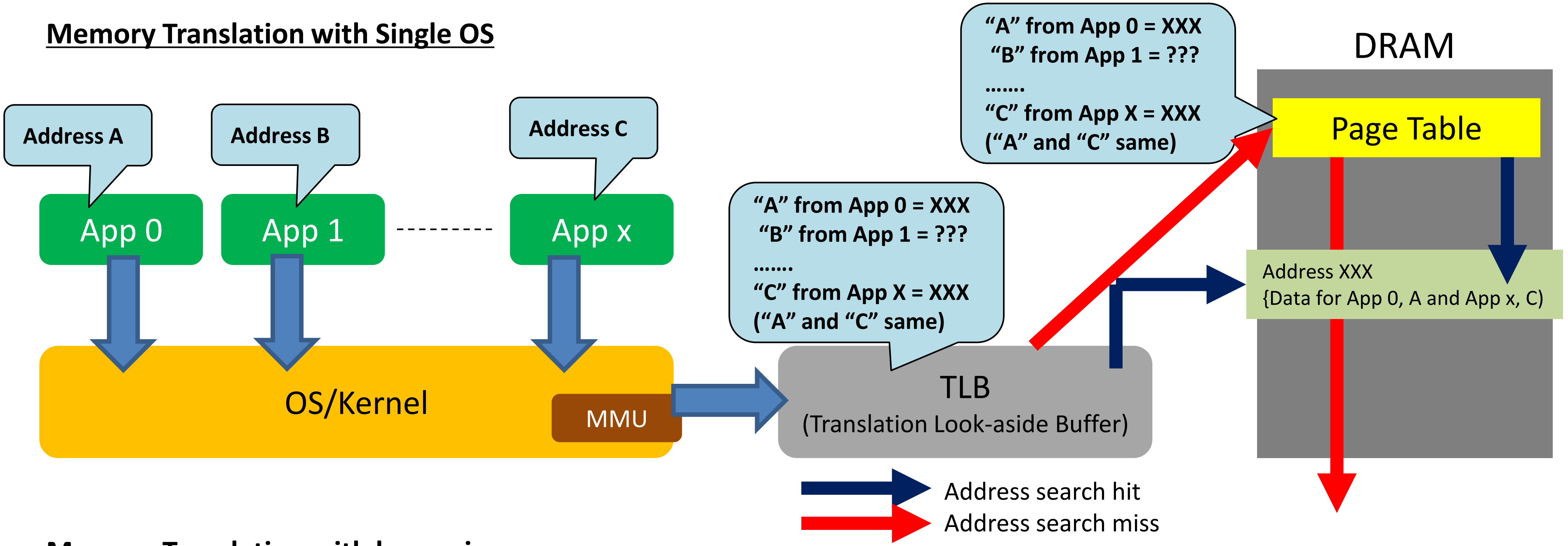
P2 @ higher priority

When needed, switch hv mode in CPSR [4:0]

→ When finished, return to P1

Virtualization Extension – MMU Overview

Memory Translation with Single OS



Address A

Address B

Address C

App 0

App 1

App x

OS/Kernel

MMU

“A” from App 0 = XXX
“B” from App 1 = ???
…….
“C” from App X = XXX
(“A” and “C” same)

TLB
(Translation Look-aside Buffer)

“A” from App 0 = XXX
“B” from App 1 = ???
…….
“C” from App X = XXX
(“A” and “C” same)

DRAM

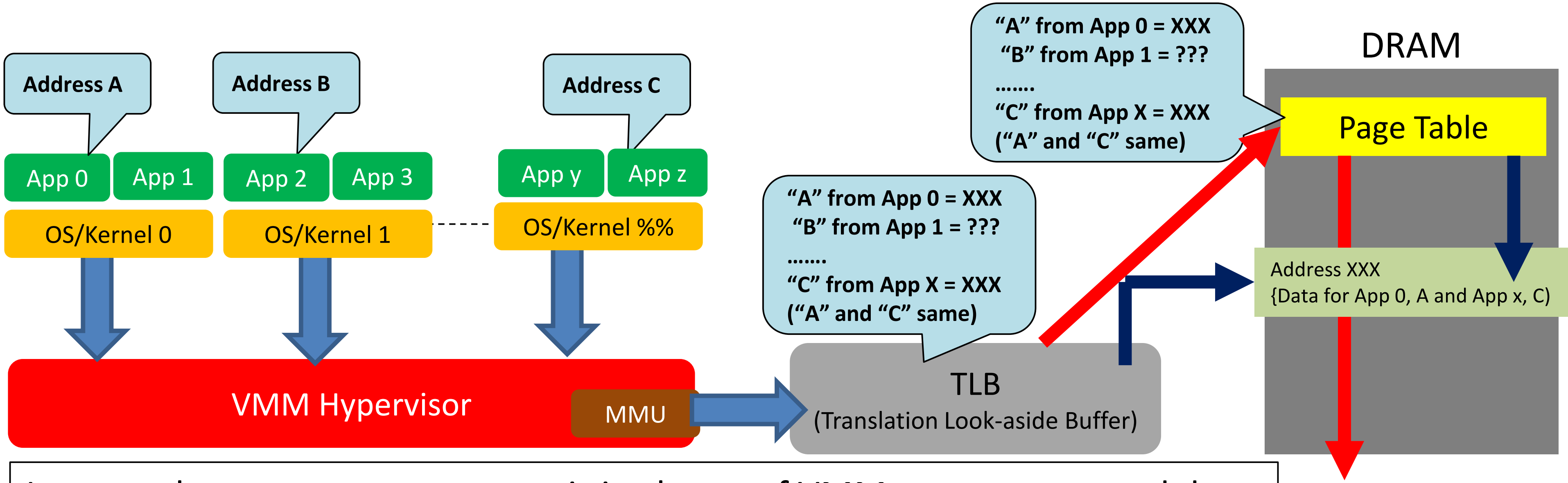
Page Table

Address XXX
{Data for App 0, A and App x, C)

Address search hit

Address search miss

Memory Translation with hypervisor



Address A

Address B

Address C

App 0

App 1

App 2

App 3

App y

App z

OS/Kernel 0

OS/Kernel 1

OS/Kernel %%

VMM Hypervisor

MMU

“A” from App 0 = XXX
“B” from App 1 = ???
…….
“C” from App X = XXX
(“A” and “C” same)

TLB
(Translation Look-aside Buffer)

“A” from App 0 = XXX
“B” from App 1 = ???
…….
“C” from App X = XXX
(“A” and “C” same)

DRAM

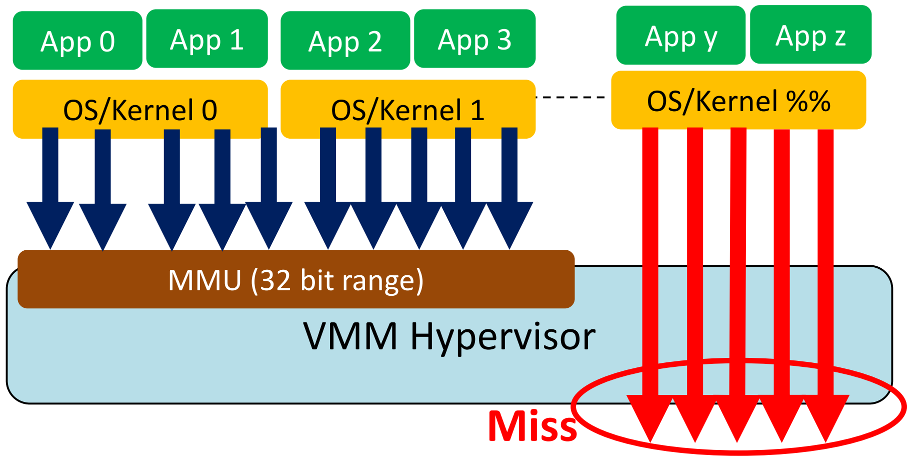
Page Table

Address XXX
{Data for App 0, A and App x, C)

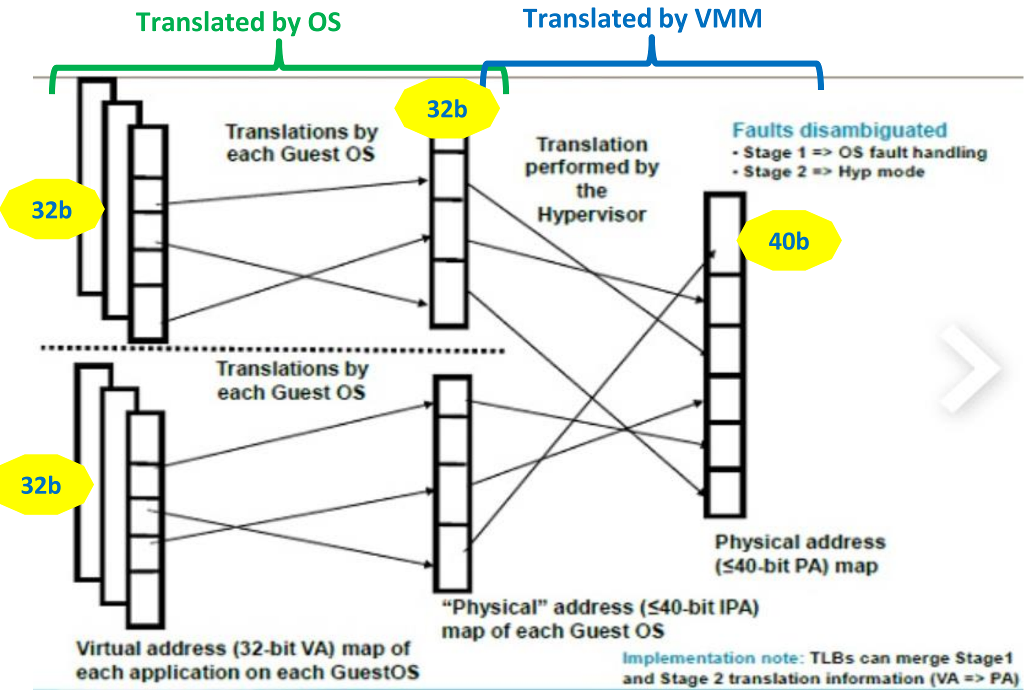
In general, memory management is in charge of VMM to access wanted data

Virtualization Extension – MMU with LPAE

Memory Translation with Hypervisor - General



Memory Translation with Large Physical Address Extension (LPAE)


Translated by OS
Translated by VMM
32b
Translations by each Guest OS
32b
Translation performed by the Hypervisor
Faults disambiguated
- Stage 1 => OS fault handling
- Stage 2 => Hyp mode
40b
32b
Translations by each Guest OS
Physical address (≤40-bit PA) map
“Physical” address (≤40-bit IPA) map of each Guest OS
Virtual address (32-bit VA) map of each application on each GuestOS
Implementation note: TLBs can merge Stage1 and Stage 2 translation information (VA => PA)

Issue with general MMU

Many memory access from Apps on multiple OS expected

- App only assumes 32 bits on OS and not aware of other guest OS and hypervisor resource
- Virtual address issues by different App or guest OS may be mapped to same physical address
- OS not handling translation
- MMU has capacity limitation

➔ *Higher risk of “Miss”*

- Larger latency → Decreased performance
- Some of “missed” address could have been actually the same as physical address as other “hit”

Solution (LPAE): 2 Stage address translation

Stage 1:

Translation from App to OS kernel

Likewise to non-hypervisor environment

Translated to address that is to be shown to VMM

If not hit, already miss → Not paged

Stage 2:

Translated from OS level address

Handled by VMM/Hypervisor

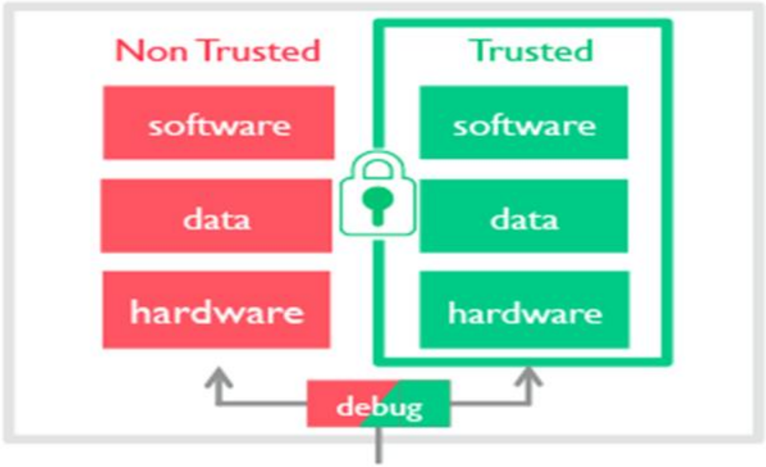
40 bits address space instead of 32 bits

➔ 256 times wider ➔ 256 times less chance of miss

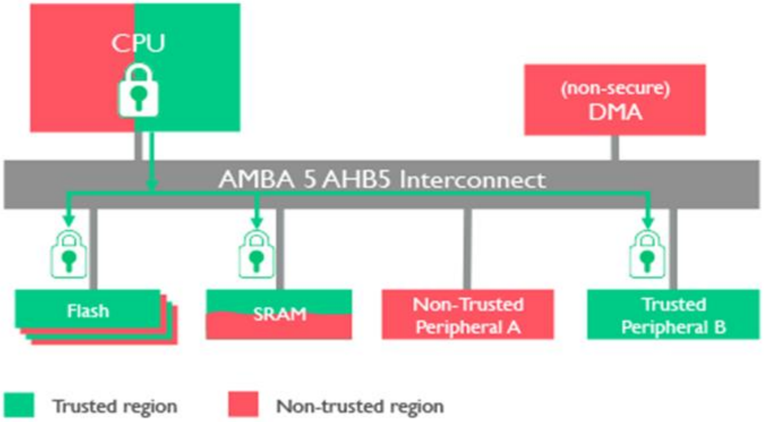
➔ 256 guest OS can be accommodated

ARM – Trusted Zone (TZ) & Trusted Execution Environment (TEE) Overview

Separation of normal world (not trusted) & Secure world (trusted)



Separation of HW resource between trusted and non-trusted



Background:

- Devices are connected to internet → more threat of security breach
- Some Apps/SW are not trusted (vulnerable, malicious etc)
- Need enclosed region for protection to guarantee secure environment

Trusted Execution Environment:

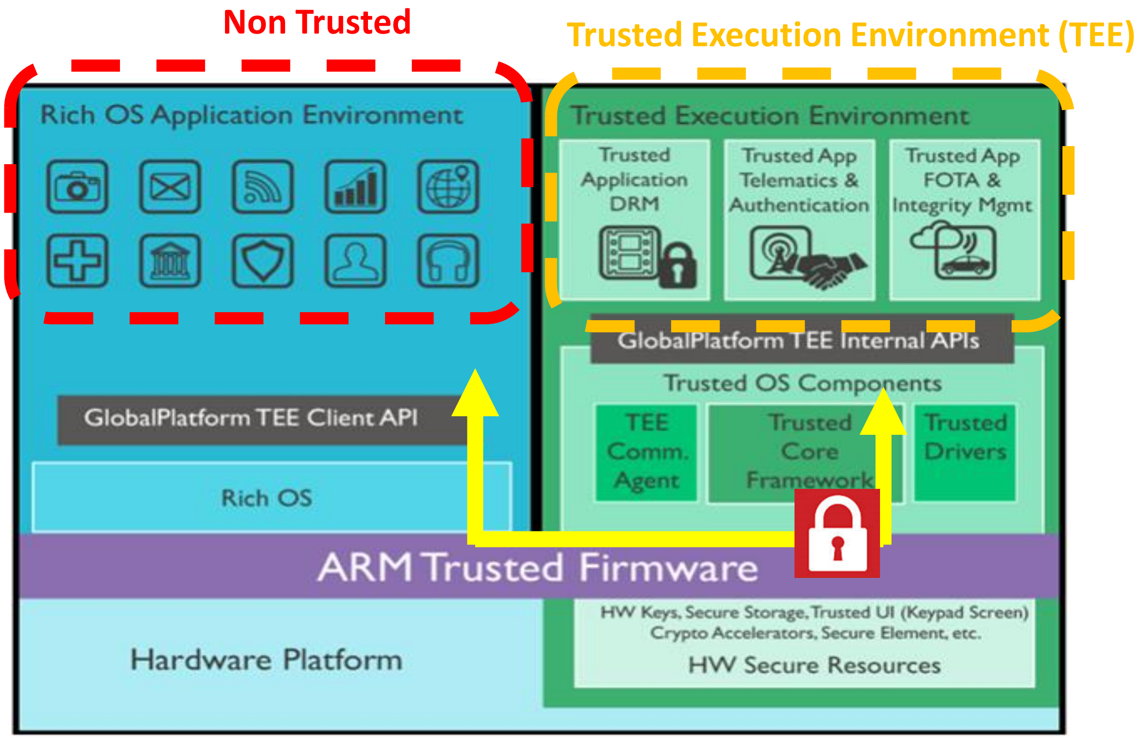
- First defined by Open Mobile Terminal Platform (OMTP)
- Platform defining to isolate between secure and non-secure region

Trusted Zone:

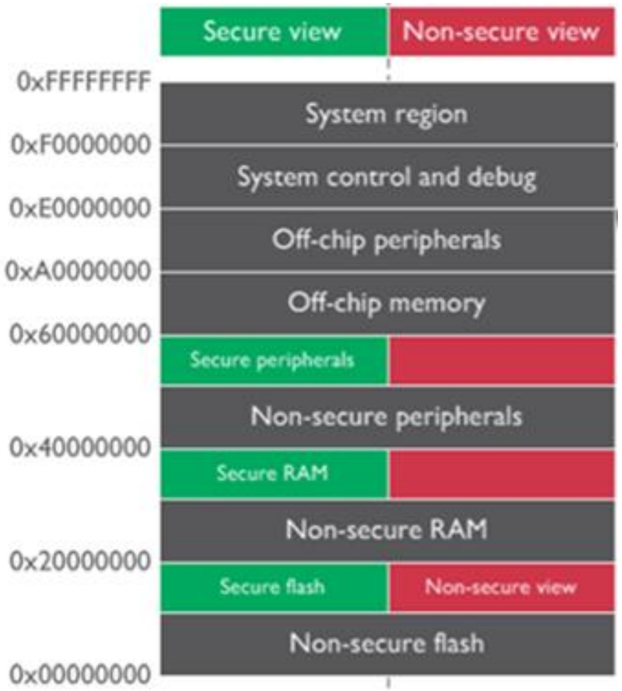
- ARM technology to implement TEE
- Protection not only CPU and software but also other HW resource (i.e. memory, IO)

ARM - TZ

TZ on ARM


Non Trusted
Trusted Execution Environment (TEE)
Rich OS Application Environment
Trusted Execution Environment
Trusted Application DRM
Trusted App Telematics & Authentication
Trusted App FOTA & Integrity Mgmt
GlobalPlatform TEE Internal APIs
Trusted OS Components
TEE Comm. Agent
Trusted Core Framework
Trusted Drivers
GlobalPlatform TEE Client API
Rich OS
ARM Trusted Firmware
HW Keys, Secure Storage, Trusted UI (Keypad Screen) Crypto Accelerators, Secure Element, etc.
HW Secure Resources
Hardware Platform

Example of memory map with TZ



Non Trusted (Normal World/Non-Secure/Rich Execution Environment)

- No guarantee of security
- Limited access to CPU/HW resource
- If security process is needed, need authentication of reaching to Trusted zone

Trusted (Secured/TEE)

- Only trusted/authorized apps are admitted (i.e. crypto, key
- Full resource access admitted

Memory mapping with TZ

- If approached from REE, region reserved for TEE are blocked (paging does not map)

ARM TZ - Implementation

Fig 1: TZ Transaction Overview

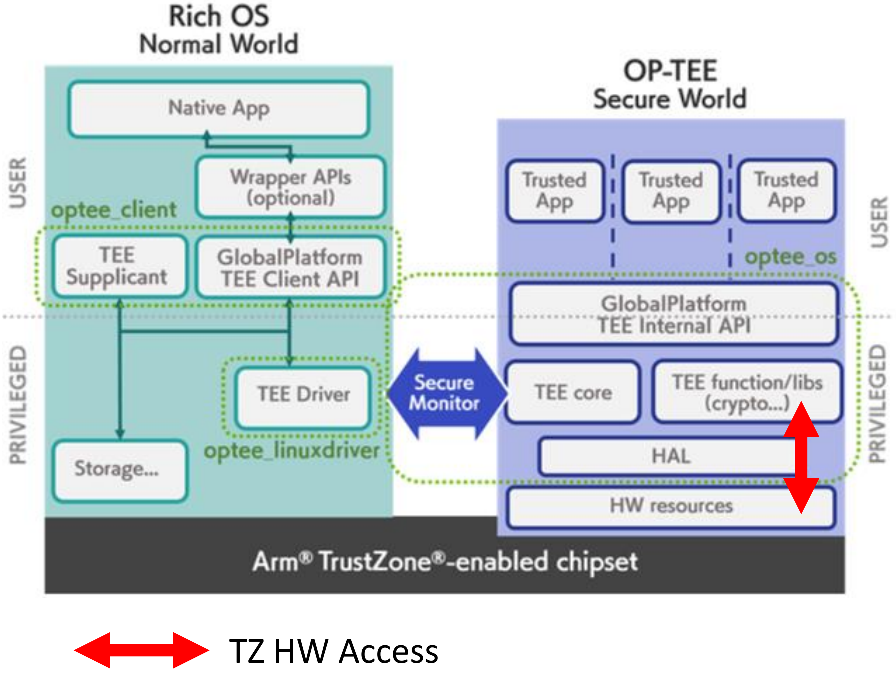


Fig 2: API to approach from REE App to TZ

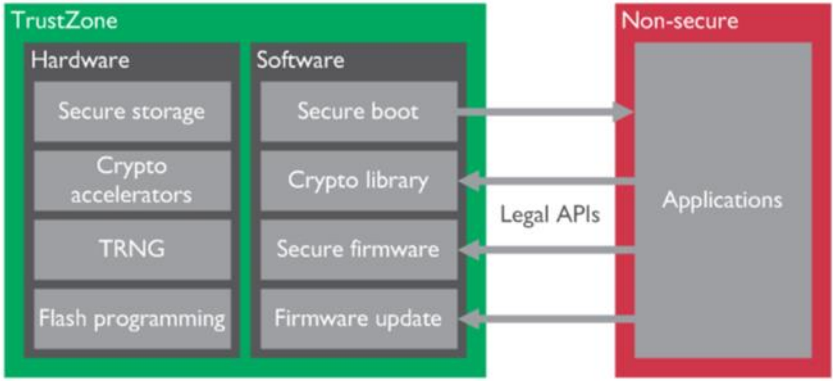
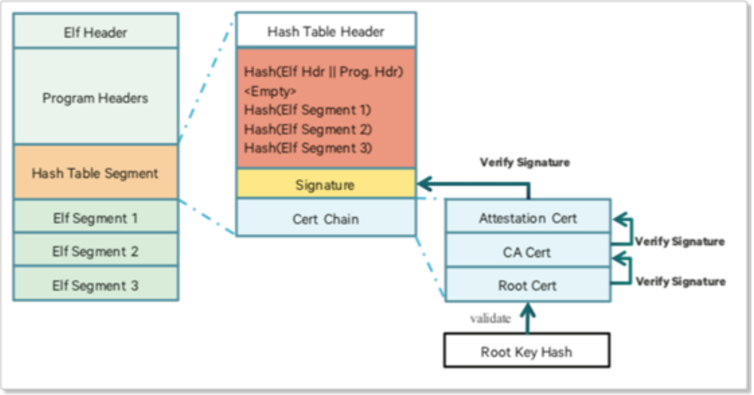


Fig 3: Example of secure monitor (certification)



Transaction between REE and TEE done with secure monitor (PL3, monitor mode)

- REE needs to have TEE driver (PL1) and TEE core at TZ interface to REE

Figure 2: REE App hits API that will activate TEE driver

Figure 3: To be admitted by TEE at secure monitor, several steps of certification steps take place

If TZ HW resource is needed (i.e. encryption), TEE API hits through library → HAL to TZ HW resource

Use Case Example: For IPSec

REE App approach to TZ with TEE API/Driver to request to encrypt payload → REE App authenticated with secure monitor → TEE approaches to HW crypto engine at TZ to encrypt → TEE returns encrypted payload to REE → RSS App transmits data

ARM – Thumb Instructions - Overview

Motivations

- Older version of ARM had less than 32 bit
  - For example, Super-H
  - Want to carry over software know-how from previous version ARM to newer version of ARM
- Want to save cost and power consumption
  - Code size can be reduced with 30% on average comparing with just using ARM-32 bit
  - Reduced code size → memory size reduction → cost down
  - Simpler code → Simpler architecture → reduction of power consumption
- Sacrifice of performance is lower priority than cost
  - Estimated to be degraded with 20%

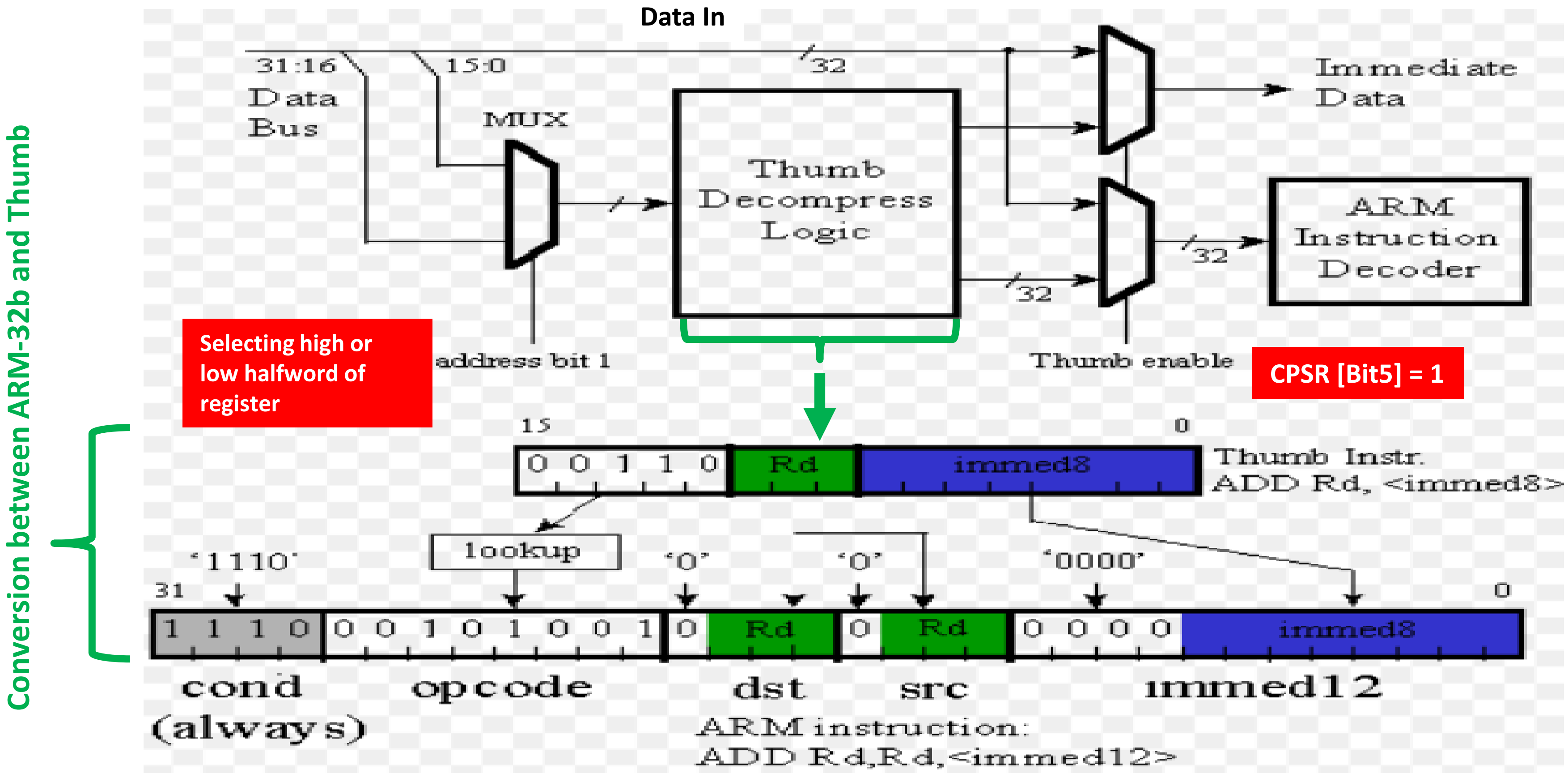
Available from ARMv4T (First one = ARM7TDMI) *{Intel ® Xscale also had it}*

ARM family with "T" supports "thumb" feature

To do "thumb" ➔ Bit 5 of CPSR must be set

- Switch between ARM-32b and Thumb triggered with branch and exchange (BX)

ARM – Thumb Instructions



Selecting high or low halfword of register

CPSR [Bit5] = 1

Conversion between ARM-32b and Thumb

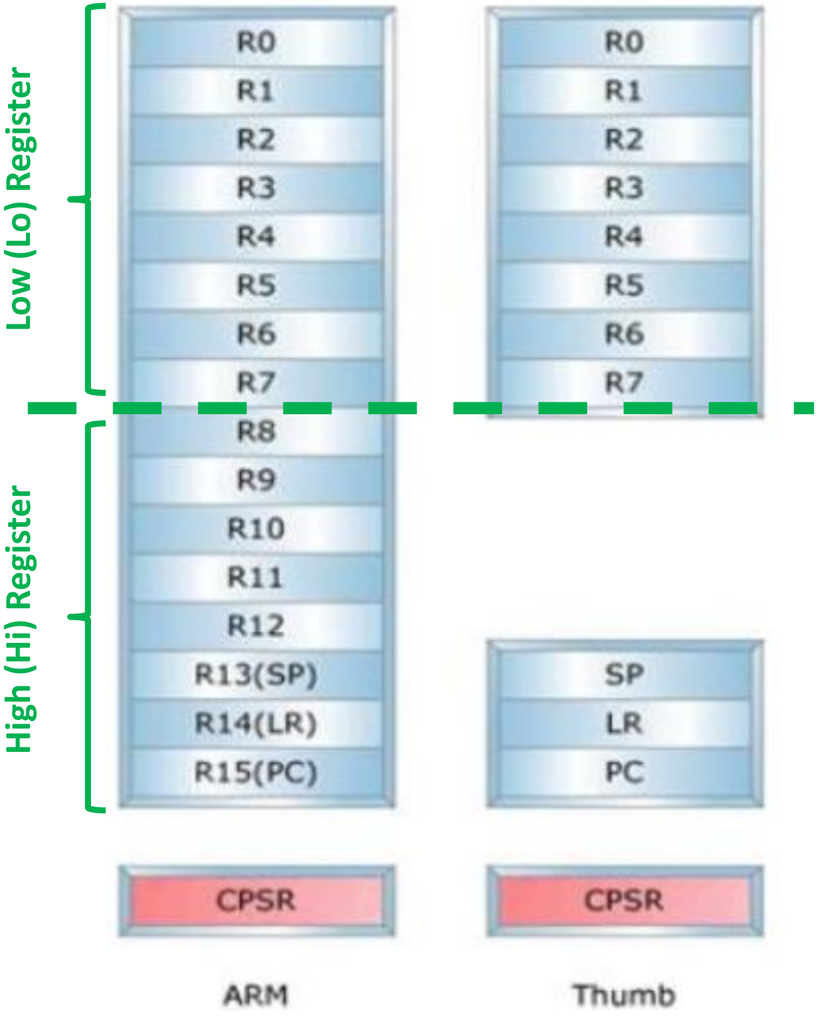
Above Diagram shows

- Architecture of processing ARM and Thumb
- Example of converting between ARM32 and Thumb {ADD Rd Rd #8bit}

Limitations with Thumb compared to ARM-32bits

- 16bit
- Only 2 operands in a single instruction
- Only sub-set of ARM-32 bit can be translated. For other instructions, must go with ARM-32bit

ARM – Thumb - Register Use



With Thumb mode:

- Only R0 – R7 can be used
- Possible to stuff 2 datagrams in a single register line, Rx, rather than just stuffing with 0 in upper 16 bits

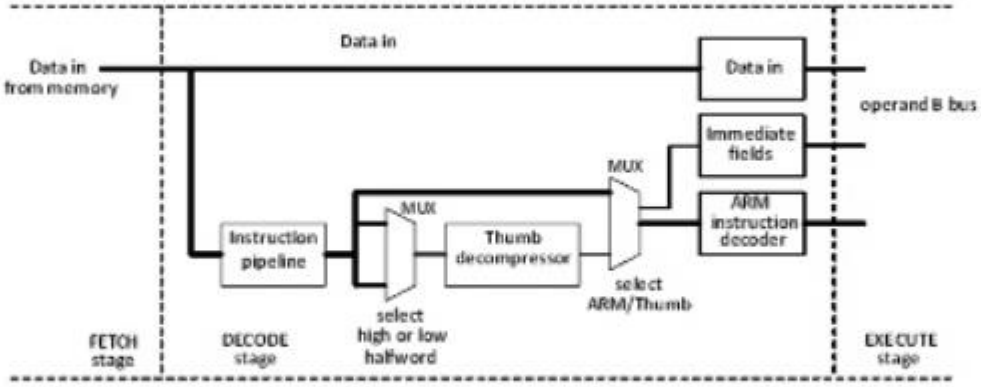
Use of SP for Thumb

- Unique stack mnemonics (**PUSH, POP**)
  - Assumes the existence of a stack pointer, **R13**
  - Equivalent to load and store instructions in ARM state

CPSR:

Thumb status bit [bit 5] is 1

ARM – Thumb - Benchmark

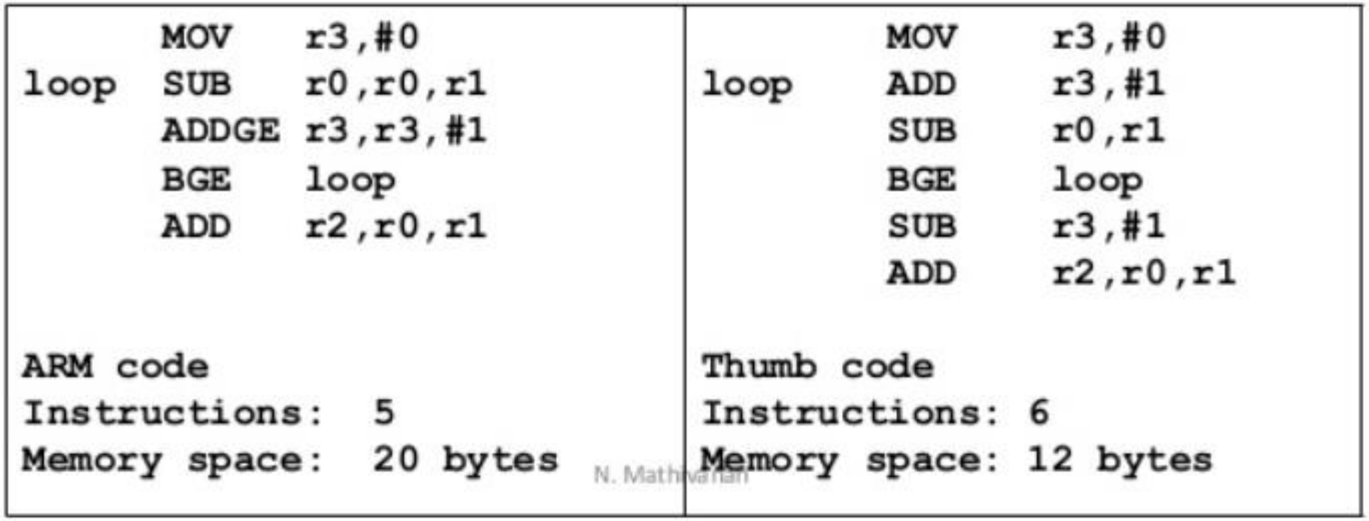


Instruction for benchmark:

Let r1, r2 have value and divisor.

Program produces quotient and remainder in r3 and r2

ARM-32 bit case **Thumb case**



```
        MOV    r3,#0
loop    SUB    r0,r0,r1
        ADDGE  r3,r3,#1
        BGE    loop
        ADD    r2,r0,r1


ARM code
Instructions:    5
Memory space:    20 bytes
```

```
        MOV      r3,#0
loop    ADD      r3,#1
        SUB      r0,r1
        BGE      loop
        SUB      r3,#1
        ADD      r2,r0,r1

Thumb code
Instructions: 6
Memory space: 12 bytes
```

"Thumb" Saves 40% of memory space ➔ Contributing to reduction of cost and power consumption (memory)

With sacrifice of

**20% of number of instructions

(**Note: Exact performance depends on CPI of each instruction)

Thumb2 and ThumbEE

Thumb-2: Mixed (32- and 16-bit) length instruction set

Available from ARMv6T2

With 32 bit mixing, can cover both instructions of Thumb and ARM-32b

- Except thumb2 has no conditional code for instruction
- Compensating with “IT (if-then)” instruction
  - `IT{x{y{z}}} {cond}` ← like “switch” in C

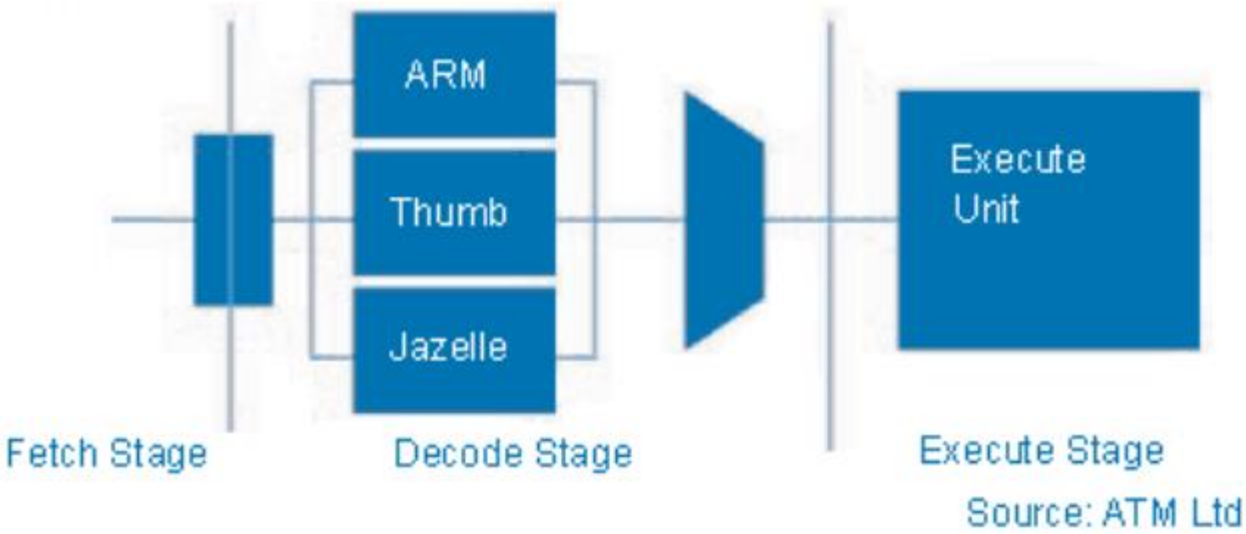
Access to full 16 registers

Thumb Execution Environment (ThumbEE)

Available from ARMv

based on Thumb, with some changes and additions to make it a better target for dynamically generated code, that is, code compiled on the device either shortly before or during execution.

ARM - Jazelle



Source: ATM Ltd

Technology to let Java bytecode to be executed directly on native ARM HW (Getting into state along side ARM and Thumb)

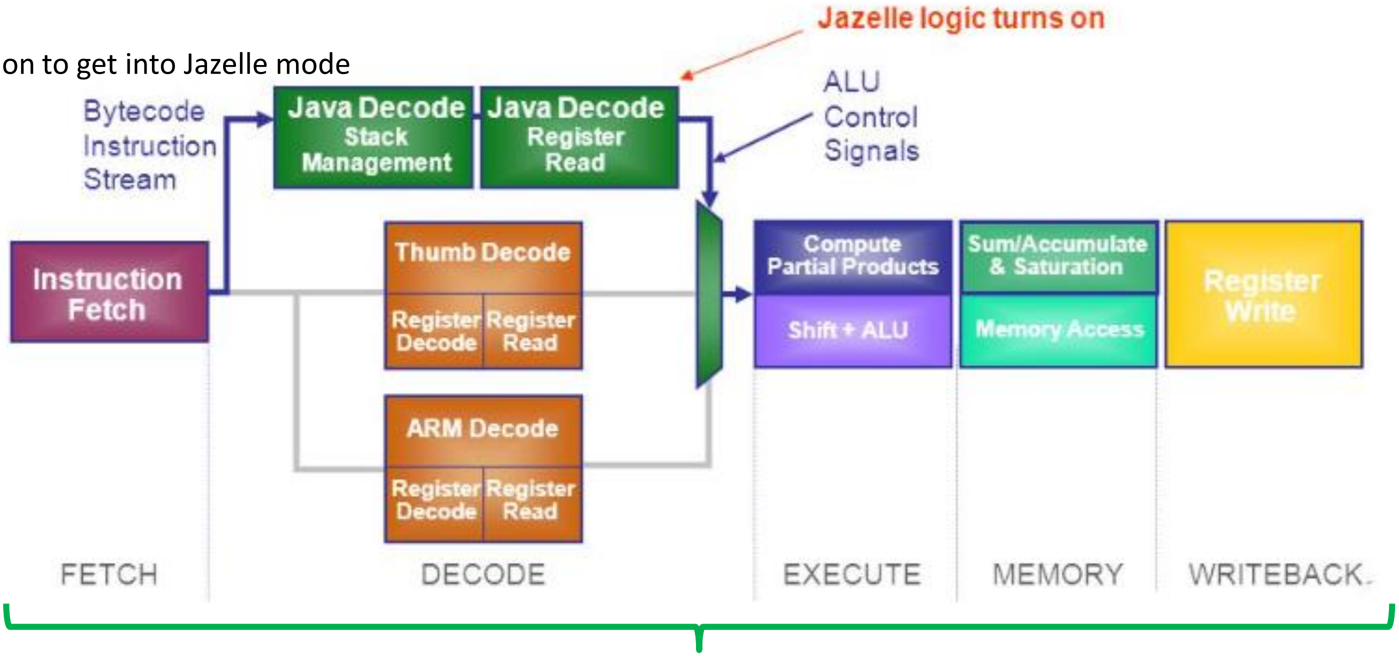
ARM version with “J” has Jazelle
(i.e. ARM9EJ-S and ARM7EJ-S , etc)

Available from ARMv5 (except ARMv7-M)

When Jazelle implemented, J-bit in CPSR (bit 24) must be set.

ARM – Jazelle Cont.

Use “BXJ” instruction to get into Jazelle mode



6 stages pipelines

Motivations for Jazelle

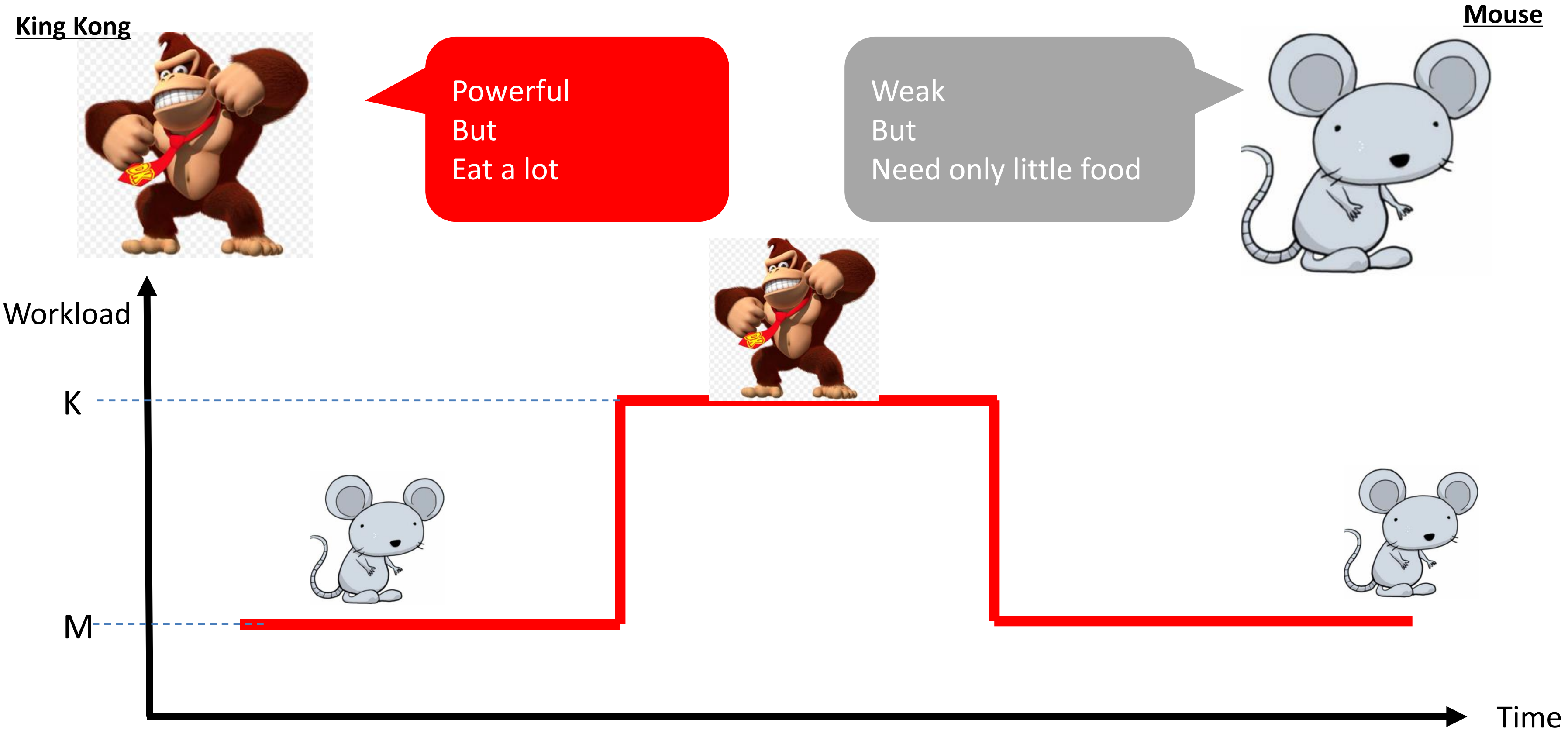
- 80% or more Java App instructions are so simple that they can be done on HW native code (8 bit instructions instead of 32bit~)

Benefit of Jazelle:

- 10 times performance increase for Java App than using only Java Virtual Machine (JVM)
  - Only 6 stages pipelines vs 11 stages or more (Cortex-A family)
- Reduced code size → Reduction of memory size possible
  - 8 bit instruction size vs 32 bit ~ if executed on ARM mode

JVM can let 80% or more Java app code to be done on Jazelle native HW

big.LITTLE Technology - Overview



Powerful
But
Eat a lot

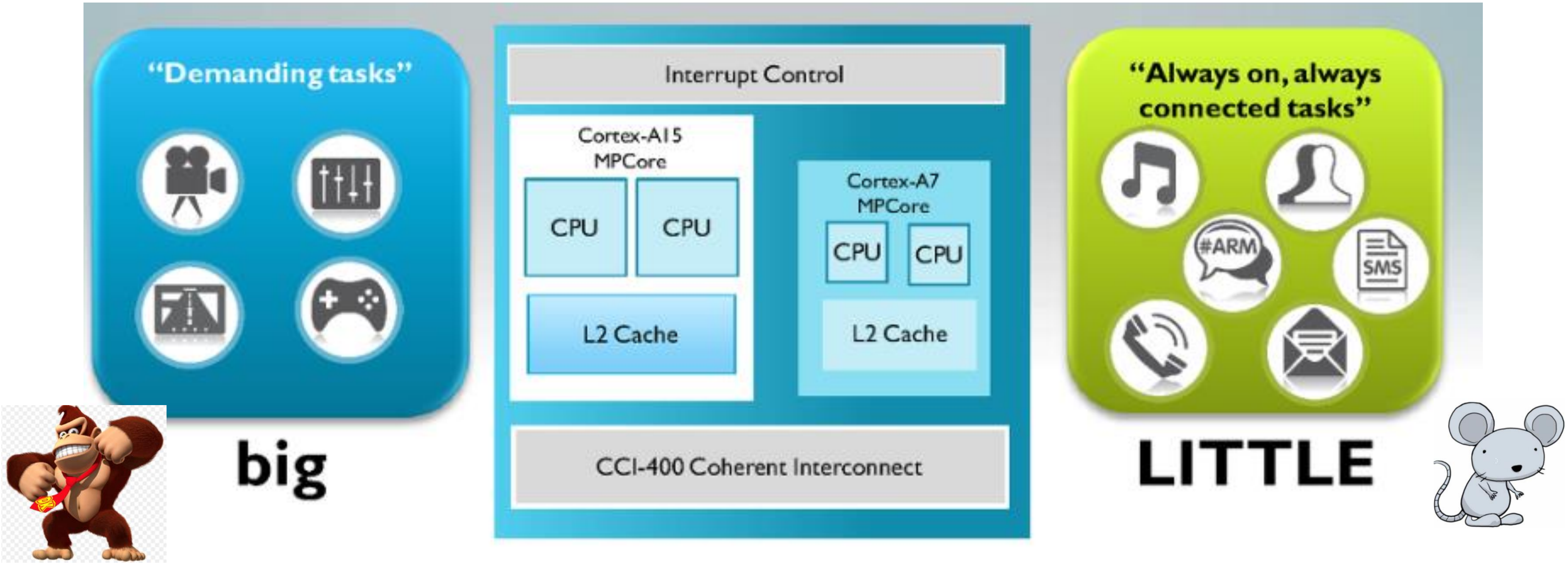
Weak
But
Need only little food

Amount of workload is dynamic over the time

- If using only King Kong → Job is done but consumes too much food
- If using only Mouse → Little food consumed but job may not be done

Switching between these 2 are needed for efficiency optimization

big.LITTLE Technology

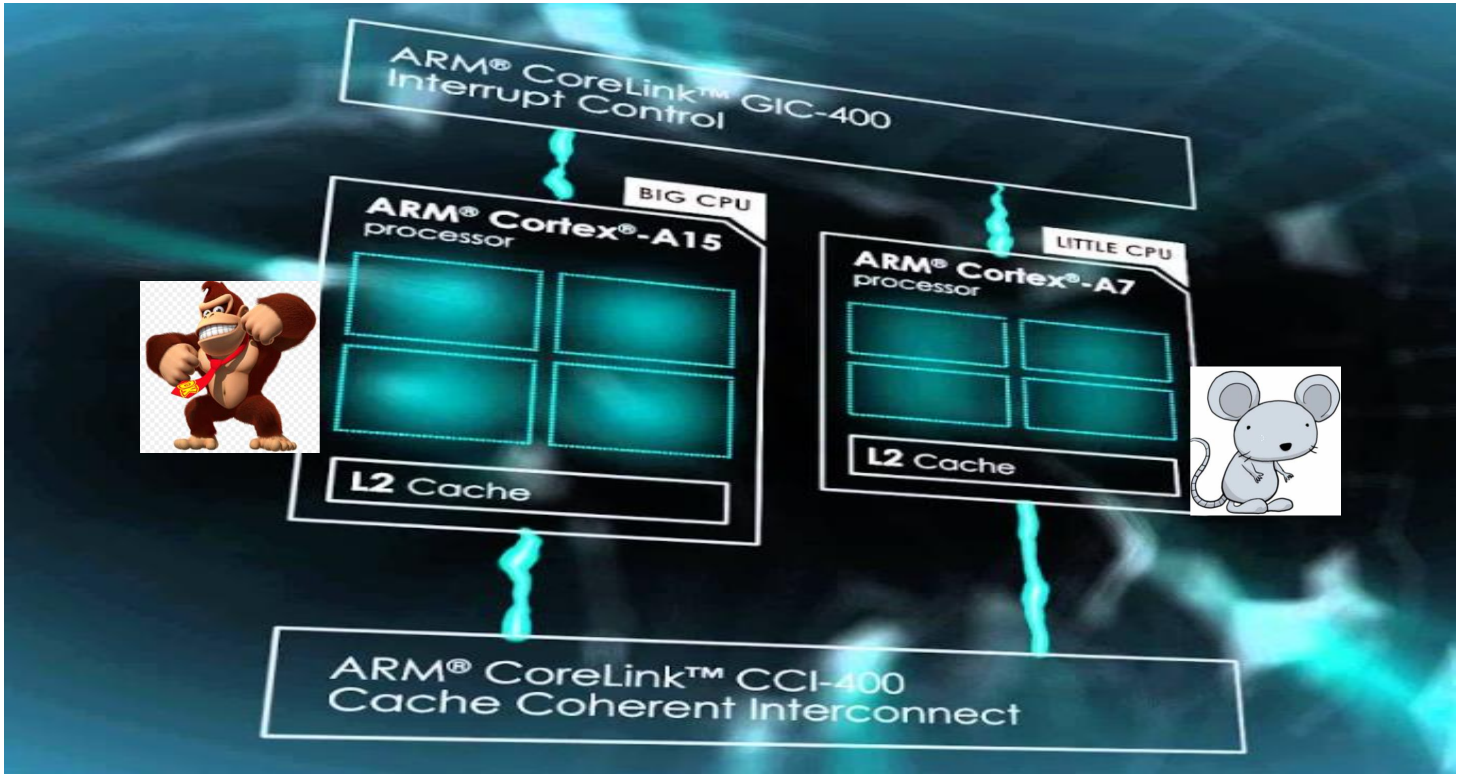


Using right processor at appropriate time

Seamless software handover using interconnect

Significant energy saving with typical workload

big.LITTLE Technology – Mechanism & Limitation



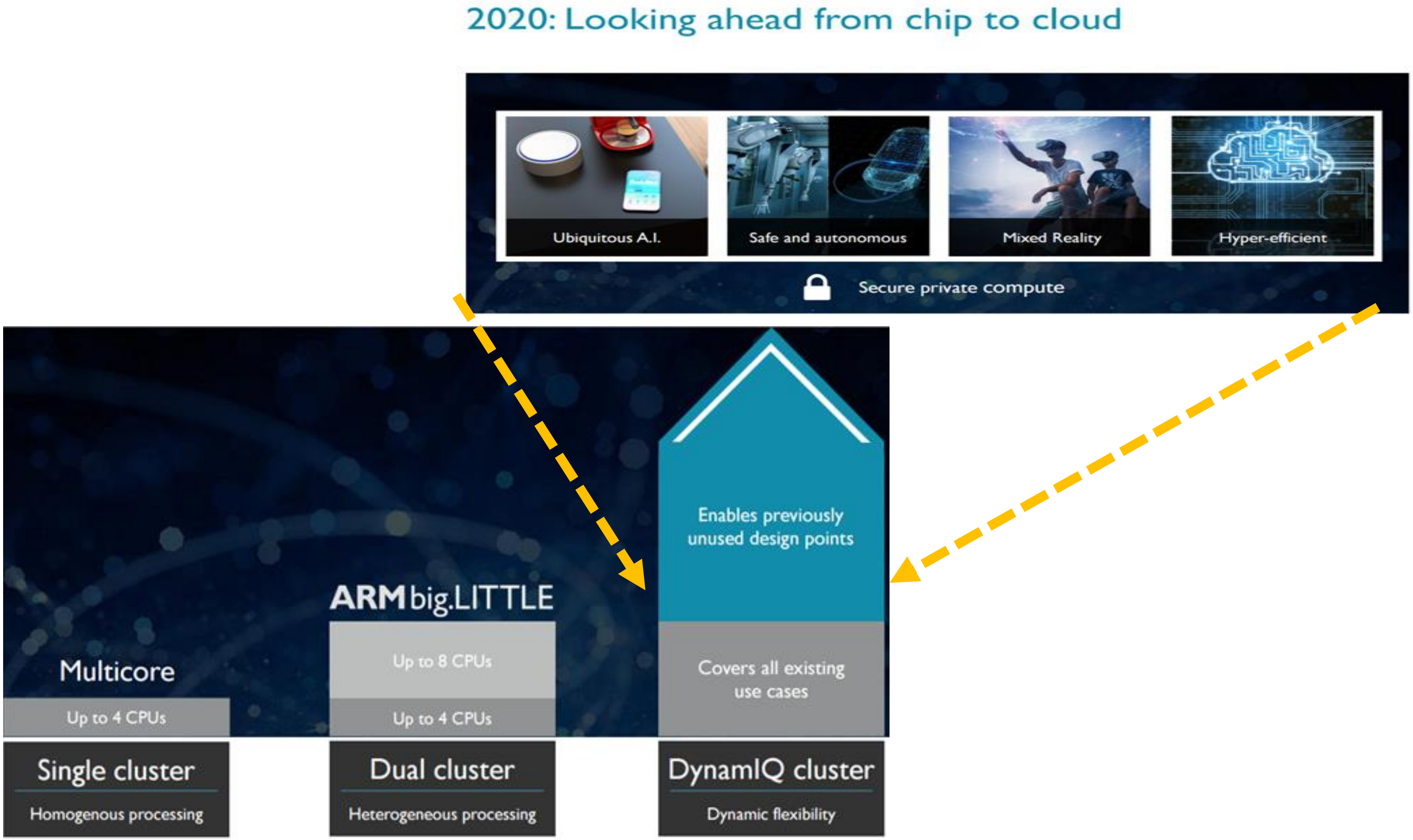
Mechanism:

- OS handles handover between Big and Little cluster
- Synchronized and triggered by GIC (interrupt) and CCI (cache coferency)

Limitation:

- If one cluster is active the other must be in sleep
- Can never activate both clusters
- All cores in active cluster must be awake

big.Little Evolution → DynamIQ - Overview



2020: Looking ahead from chip to cloud

Ubiquitous A.I.

Safe and autonomous

Mixed Reality

Hyper-efficient

Secure private compute

Enables previously unused design points

ARM big.LITTLE

Multicore

Up to 4 CPUs

Up to 8 CPUs

Up to 4 CPUs

Covers all existing use cases

Single cluster

Homogenous processing

Dual cluster

Heterogeneous processing

DynamIQ cluster

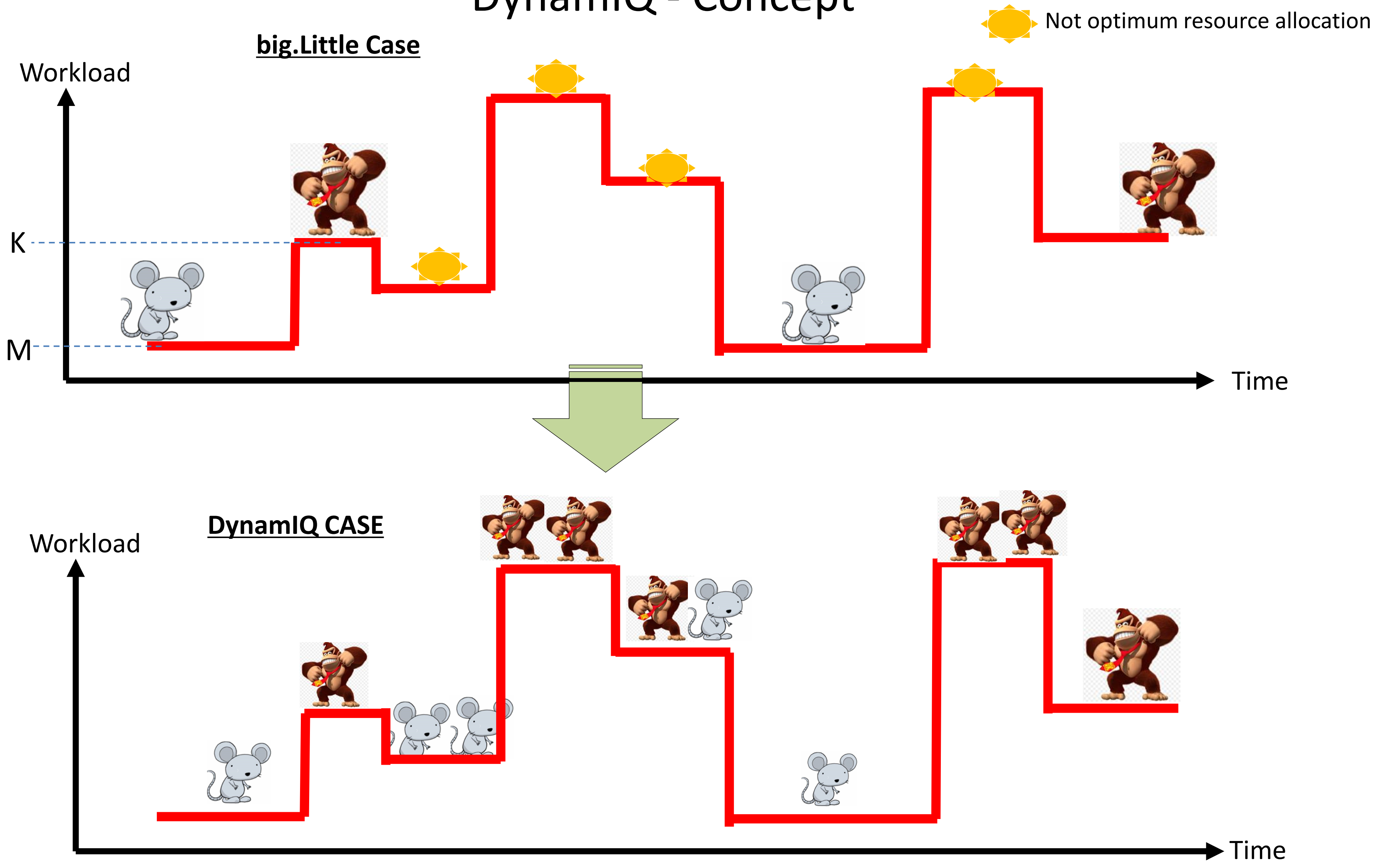
Dynamic flexibility

More intelligent and dynamic computer power needed

Workload of future device will increase but not steady all times

ARM is aiming to cloud business

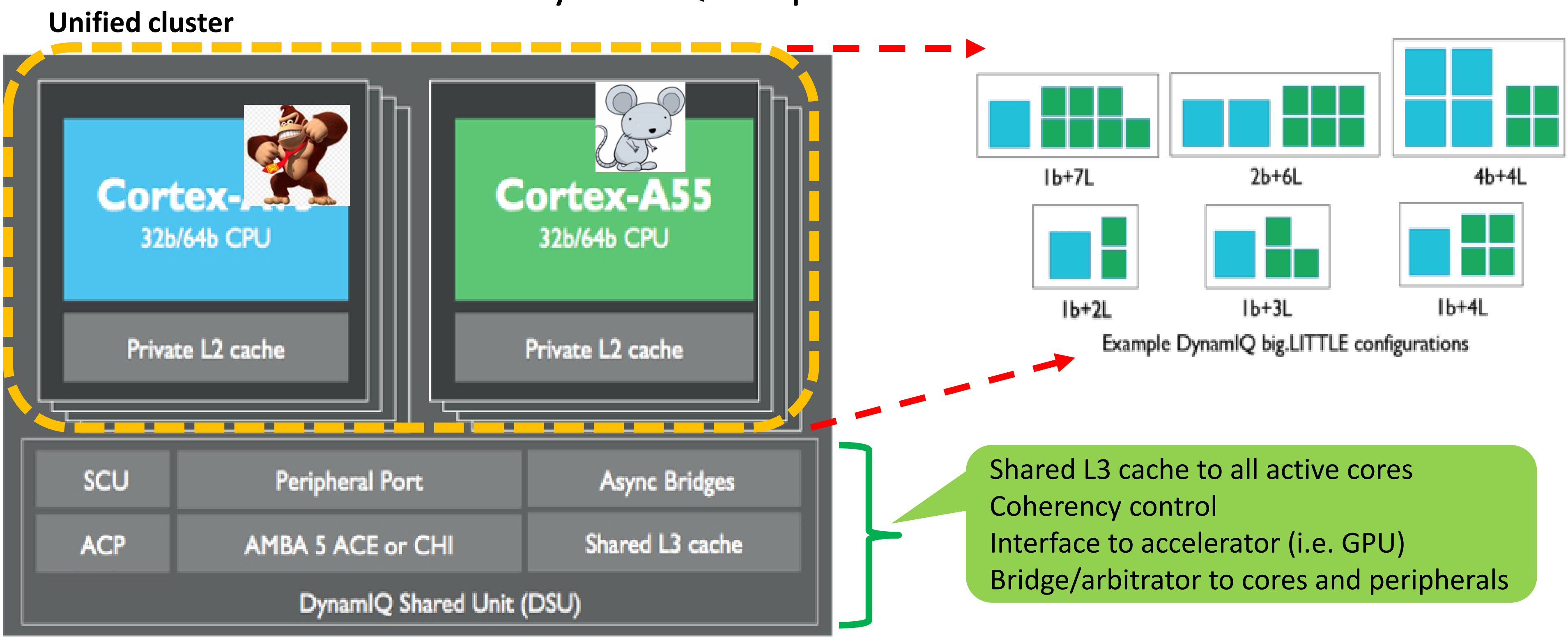
DynamIQ - Concept



Future Needs Trend:

- **Greater processing power**
- **More flexibility of performance/power level tailoring**

DynamIQ - Implementation



Shared L3 cache to all active cores
Coherency control
Interface to accelerator (i.e. GPU)
Bridge/arbitrator to cores and peripherals

Enhanced scalability of performance levels

Availability of both higher and lower performance than big.Little

Enhanced energy efficiency with more performance levels availability

Each core has own L2 cache vs L2$ shared by cores in the cluster

Core assignments with DSU (DynamIQ Shared Unit) controlling whole unified cluster

Contents

ARM Overview

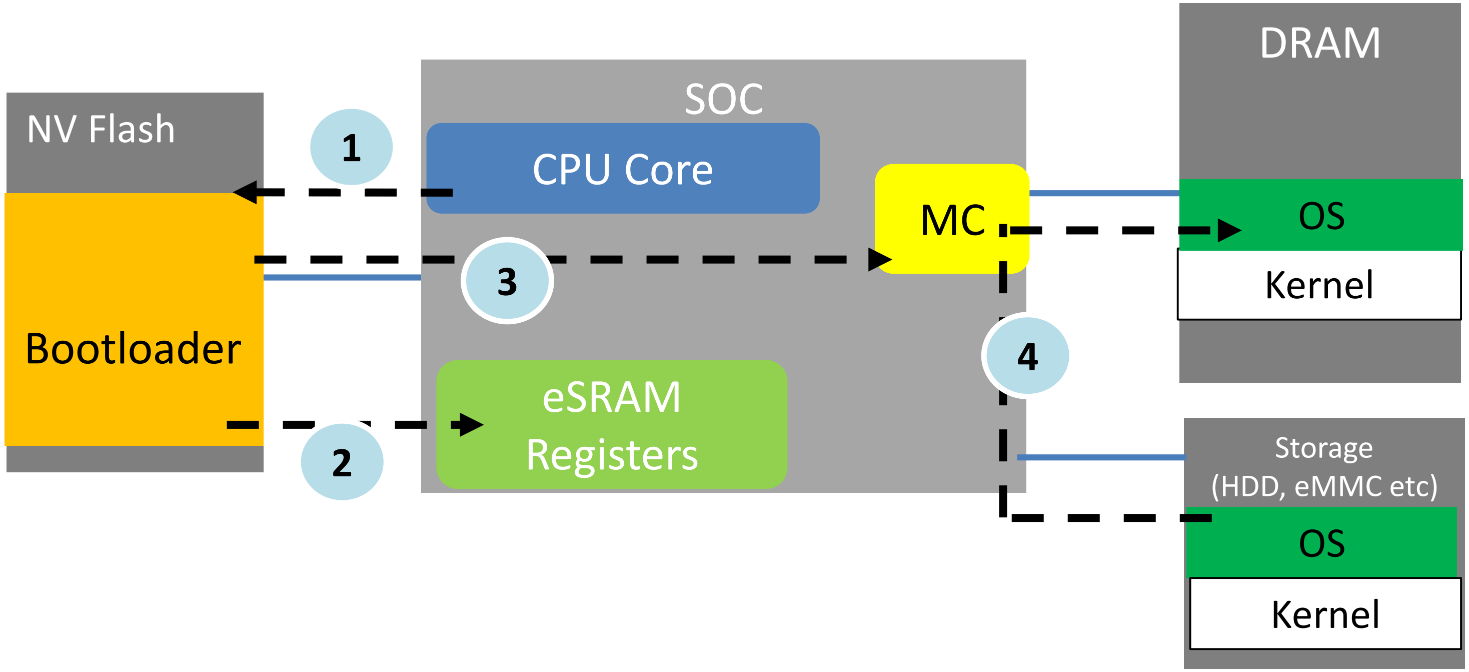
ARM Architecture Basic

ARM Key technologies

ARM Misc

- Bootloader System
- Device Tree
- Master Boot Record
- Tightly Coupled Memory
- MCU vs MPU

ARM Bootloader - Overview



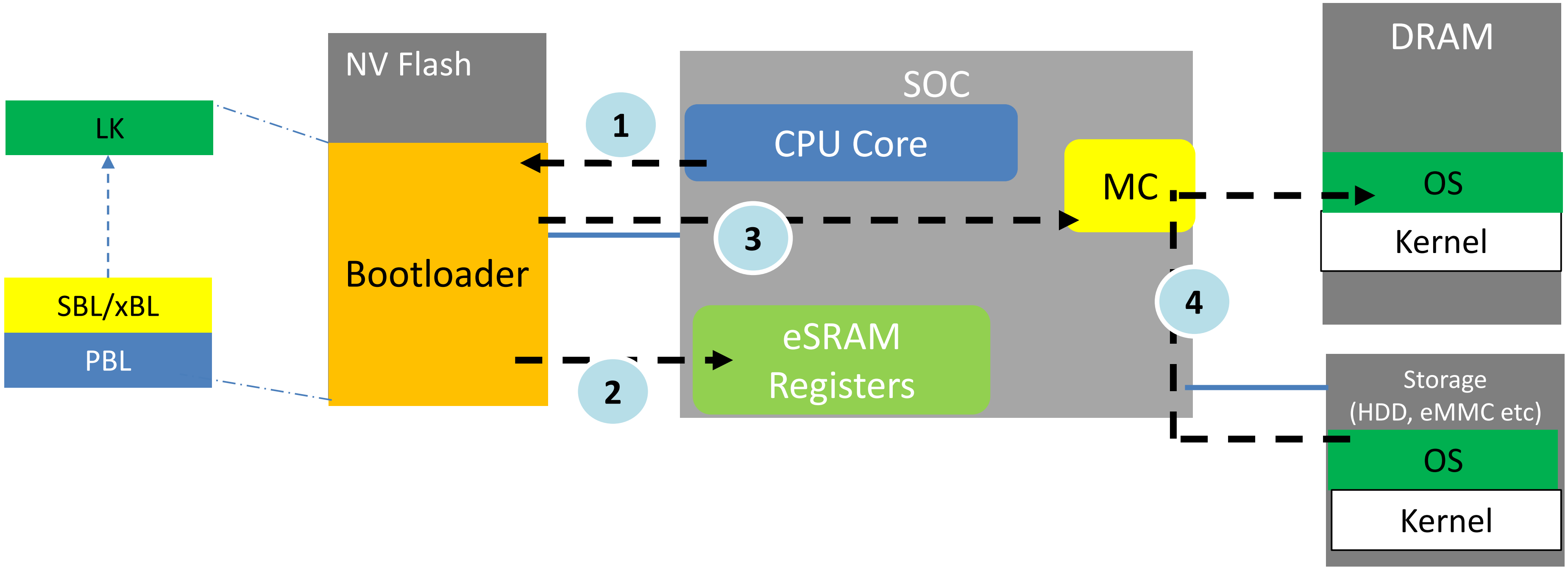
Bootloader (BL) = Program to start OS after power-on event

- Stored at the first address to be pointed after power-on in NV flash
- When all program executed ➔ handed to OS

1. After power-on, access to first address space where start line of BL is found
2. Loading file/code in storage space inside SOC (registers, u-code etc)
3. Initializing external RAM (DRAM) ← i.e. MRC
4. Loading kernel image and OS to RAM

BL completion and transition to OS

ARM Bootloader – Overview (PBL/SBL/xBL/LK)



If SOC is simple (i.e. MCU) → Single chunk of BL is enough

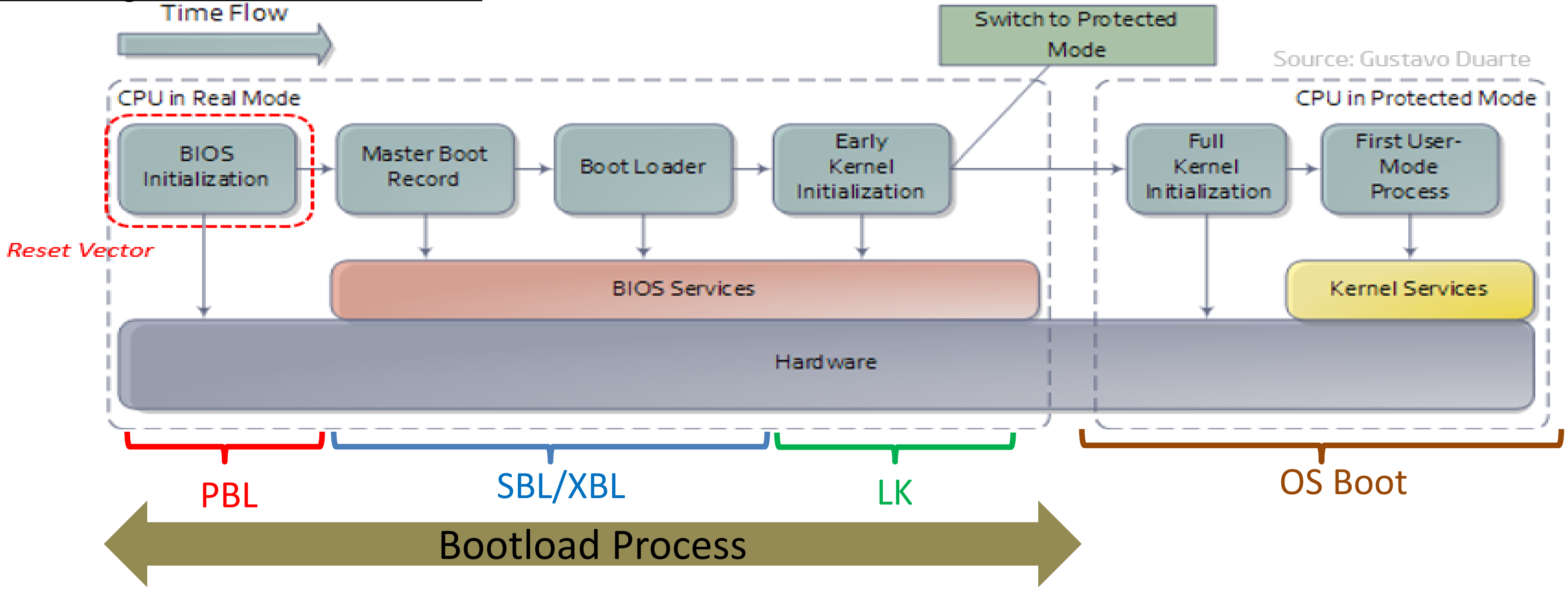
If SOC is complex (i.e.MPU/CPU) → BL consists of several blocks

Bootloader program is so simple that it can access only small of data

- If whole BL process is complex enough → Split into several stage
- BL does not proceed to next stage BL unless current stage is complete

ARM Bootloader – PBL/SBL(XBL)/LK

Note: Taking Intel Architecture Boot case



Bootload (BL) process runs with single task mode (Real Mode)

After Little Kernel (LK), ready to start multi-processing (protected mode)

- Kernel does task/resource management

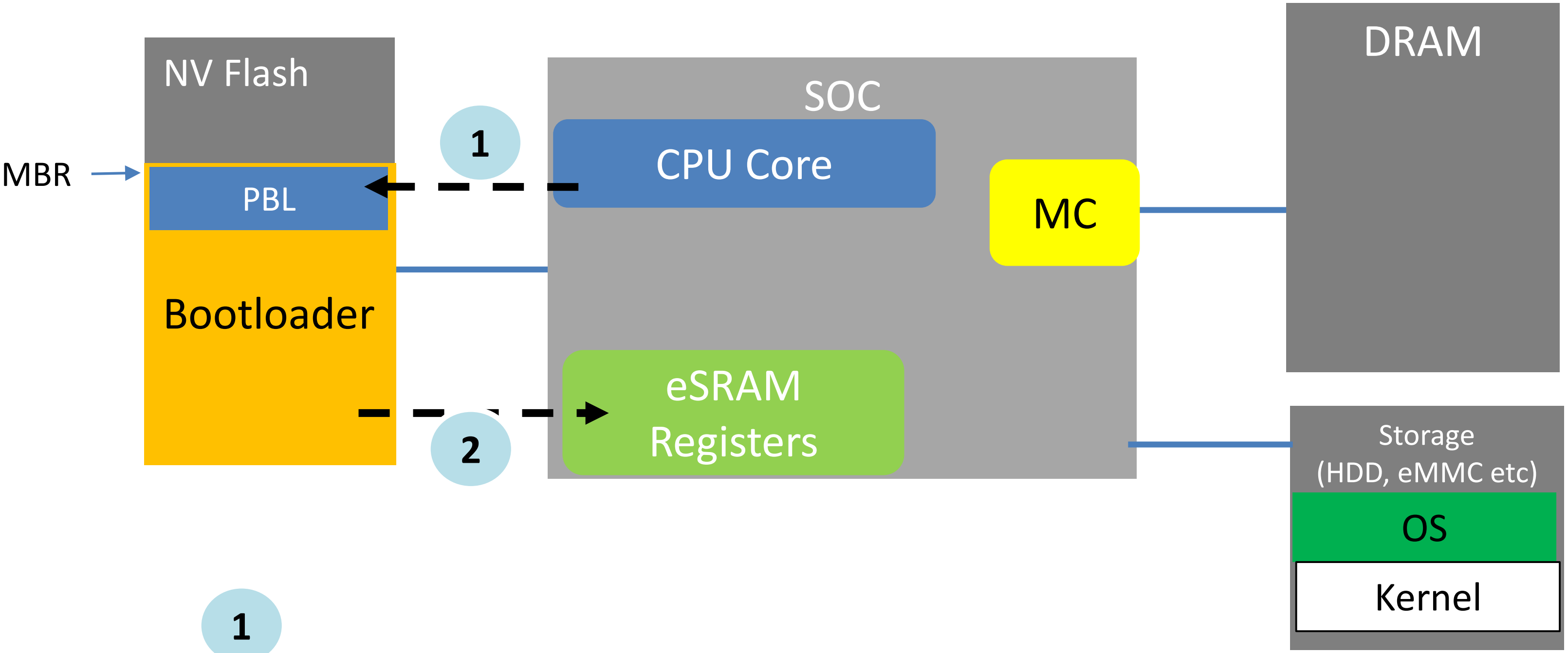
Whole BL split into stages

PBL: Initialization within SOC/CPU

SBL/XBL: Initializing peripherals & Loading selected bootstrap

LK: Loading minimum set of kernel needed to run SOC operation

ARM Bootloader – Primary Boot Loader (PBL)



1

After power-on, CPU core access to PBL in BL (known as Master Boot Record, MBR)

2

PBL loads data (&u-code) to eRAM and registers within SOC

Equivalent to firmware

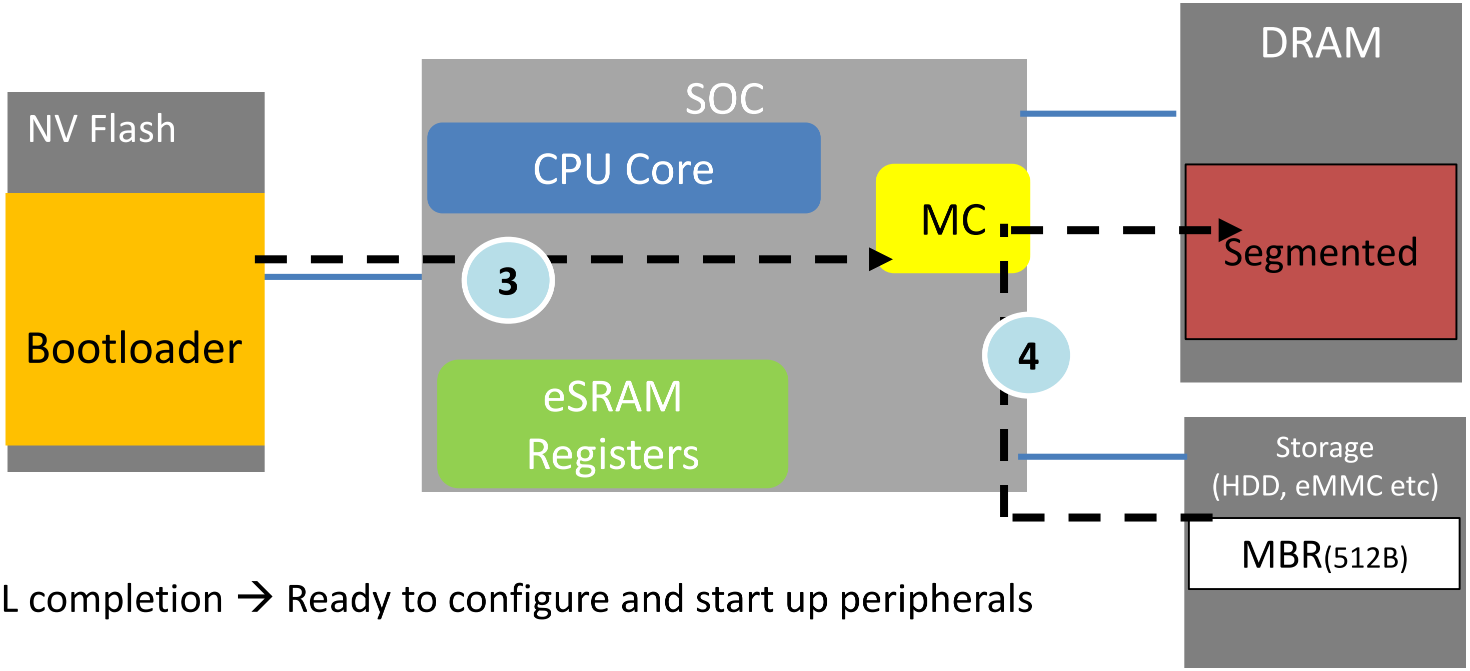
Without PBL completion, SOC is not initialized and IOs not active

(Not able to boot OS, initialize peripherals including memory/storage etc)

At completion of PBL, functions in SOC ramped & IOs available

Ready to start access to peripheral to continue booting whole system

ARM Bootloader – Secondary/Extensible BL (SBL/XBL)



After PBL completion → Ready to configure and start up peripherals

DDR initialization → Allocation of DDR regions (Data, MMIO, etc)

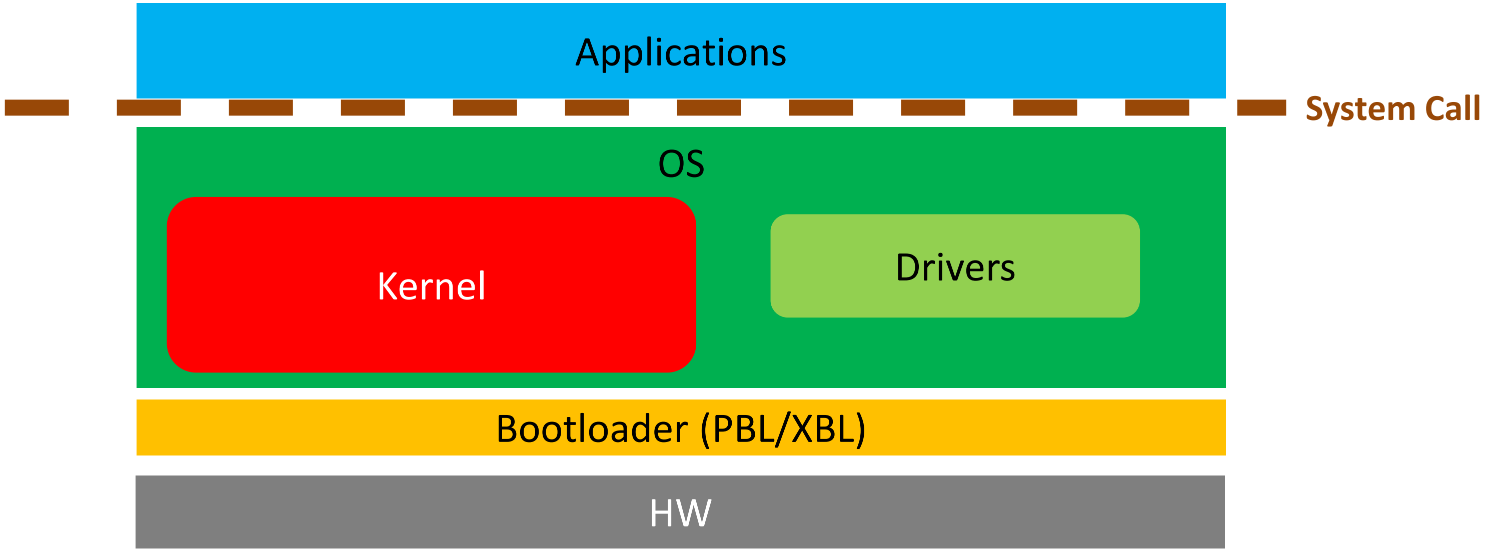
Bootstrap begins (i.e. LILO for Linux, GRUB for Unix, BOOTMGR for Win Vista~, etc)

- Start with referring to Master Boot Record (MBR) stored in storage (HDD, SDD, eMMC etc)
- With Reference to MBR, it loads flagged partition to load
  - Each partition has information of file type information (NFTS, exFAT, FAT32 or signs of empty)
- Loading flagged partition data into DDR memory

Then ready to start loading OS kernel

Equivalent to UEFI
(Unified Extensible Firmware Interface)
SBL/XBL = link between HW/FW and OS

ARM Bootloader – Kernel Overview



System call: Gate for applications to start approaching to OS/SOC resource

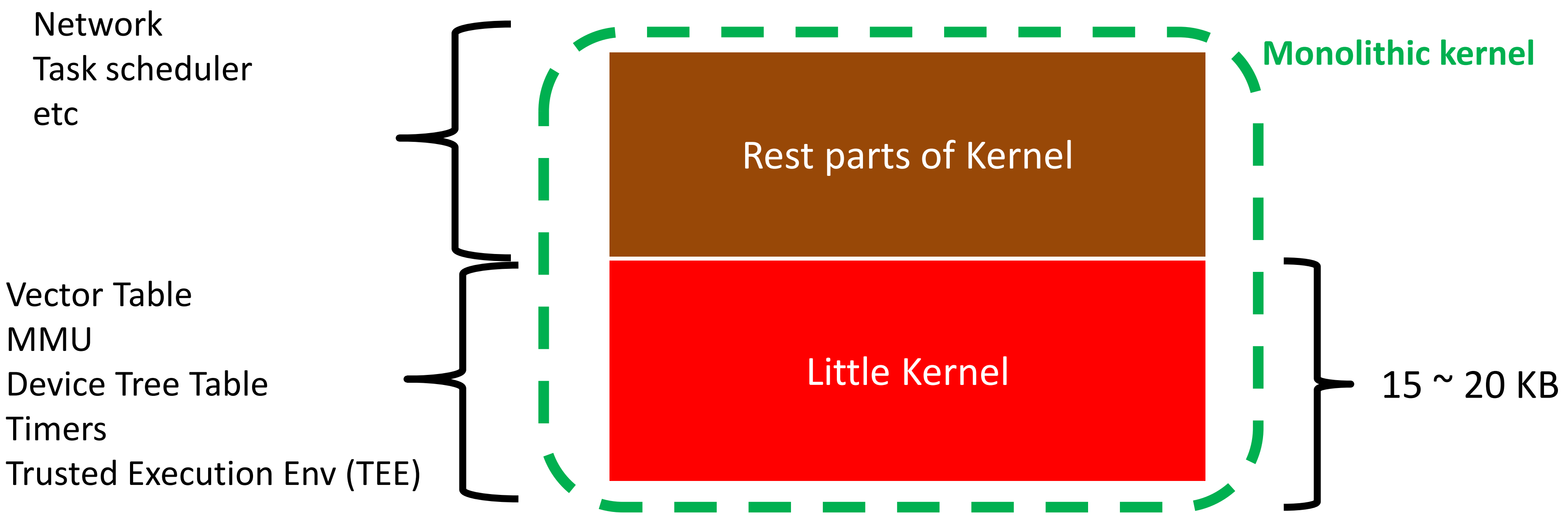
- Applications approach to resource of OS
- SOC vendors give library code (API) to initiate the access to it

Kernel: Core of OS to bridge between App and HW to handle HW resource

Main kernel blocks:

- Process/Task Mgmt; Tasks prioritization/scheduling, Task state handling
- Memory Mgmt; Paging (physical <-> logical address), memory region allocation
- Device Mgmt: Linking applications and target HW (IO, memory, etc)
  - Using drivers

ARM Bootloader – Little Kernel (LK)



Network
Task scheduler
etc

Vector Table
MMU
Device Tree Table
Timers
Trusted Execution Env (TEE)

Monolithic kernel

Rest parts of Kernel

Little Kernel

15 ~ 20 KB

Minimum subset of kernel;

- Always included regardless of OS
- Single task mode (Real Mode) also use

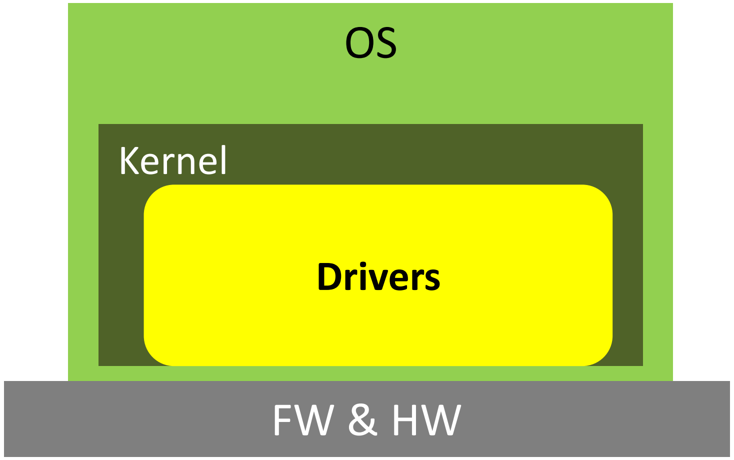
Loaded separately before main kernel and included as part of BL

After LK completion, SOC is ready to do SMP (Protected Mode)

Available in GitHub with source code

Device Tree - Overview

Without DT



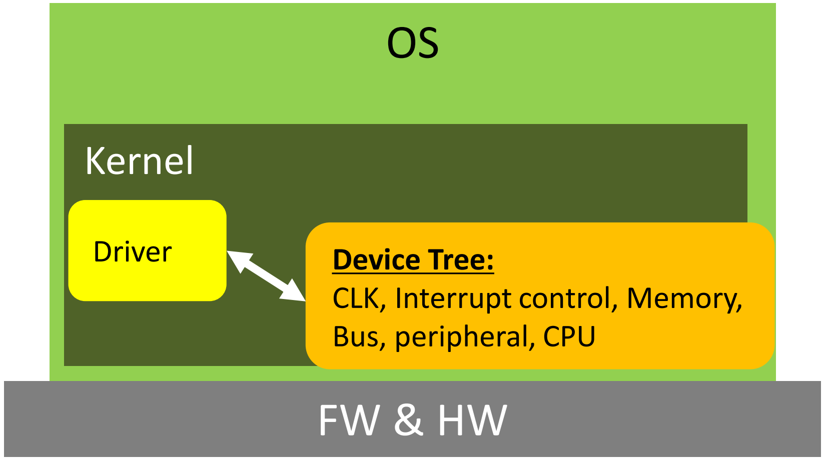
OS

Kernel

Drivers

FW & HW

With DT



OS

Kernel

Driver

Device Tree:
CLK, Interrupt control, Memory, Bus, peripheral, CPU

FW & HW

Device Tree: ARM Linux property with data structure info/repository

- Device drivers refer to DT

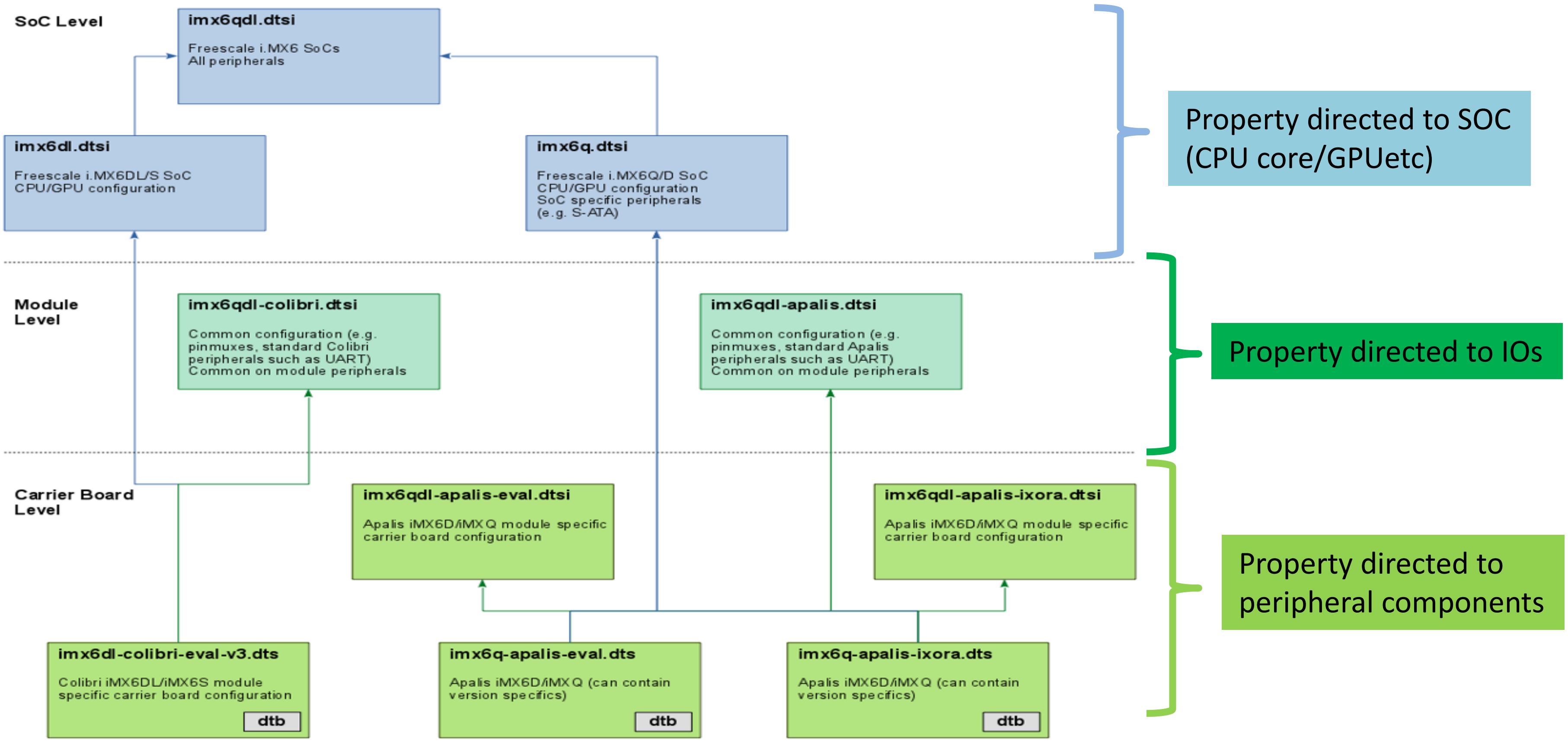
Issue without DT

- Large portion of kernel source owned by drivers & including unused ones
- No standard driver script format for ARM ➔ Custom effort ➔ No reusability

DT Benefit

- DT created for platform specific➔ Efficient kernel usage
- Reusability to other ARM platform running Linux

Device Tree - Structure



SoC Level

imx6qdl.dtsi
Freescale i.MX6 SoCs
All peripherals

imx6dl.dtsi
Freescale i.MX6DL/S SoC
CPU/GPU configuration

imx6q.dtsi
Freescale i.MX6Q/D SoC
CPU/GPU configuration
SoC specific peripherals
(e.g. S-ATA)

Property directed to SOC (CPU core/GPUetc)

Module Level

imx6qdl-colibri.dtsi
Common configuration (e.g. pinmuxes, standard Colibri peripherals such as UART)
Common on module peripherals

imx6qdl-apalis.dtsi
Common configuration (e.g. pinmuxes, standard Apalis peripherals such as UART)
Common on module peripherals

Property directed to IOs

Carrier Board Level

imx6qdl-apalis-eval.dtsi
Apalis iMX6D/iMX Q module specific carrier board configuration

imx6qdl-apalis-ixora.dtsi
Apalis iMX6D/iMX Q module specific carrier board configuration

imx6dl-colibri-eval-v3.dts
Colibri iMX6DL/iMX6S module specific carrier board configuration
dtb

imx6q-apalis-eval.dts
Apalis iMX6D/iMX Q (can contain version specifics)
dtb

imx6q-apalis-ixora.dts
Apalis iMX6D/iMX Q (can contain version specifics)
dtb

Property directed to peripheral components

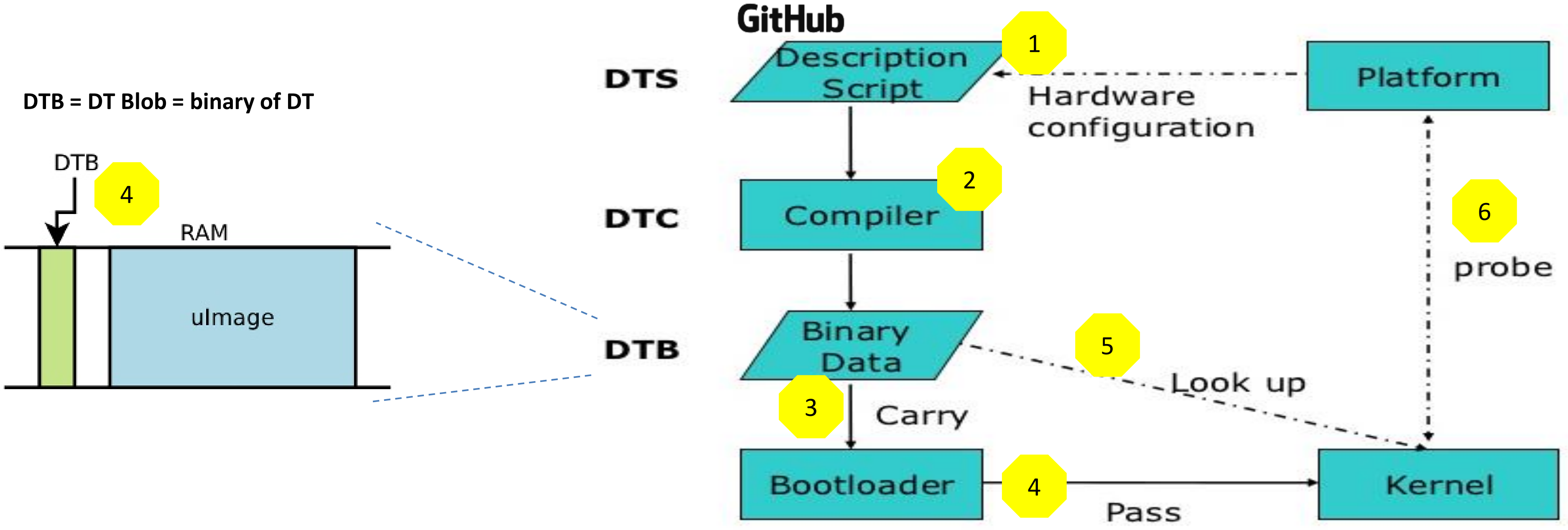
DT is hierarchical structure & there must be correlation

When DT is developed, tree structure is planned

Each DT files (dtb) located under “/proc/device-tree” in Linux directory

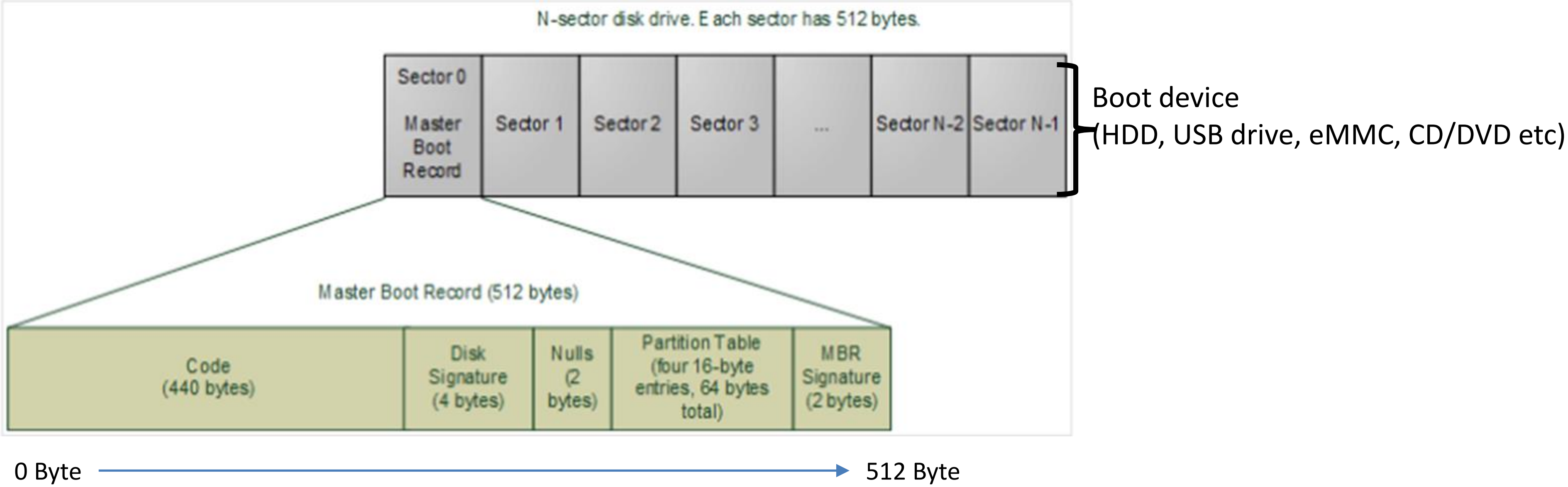
- Contents checked with “hexdump” command

Device Tree – Building into Linux



1. After data structure is planned, develop DT source code (DTS)
   - Base source code available from GitHub (linux/arch/arm/boot/dts)
   - dts = Board specific & dtsi = SOC specific
2. Through compiler (DTC) ➔ generate binary (DTB)
3. Generated DTB built into bootloader/boot image
4. When system boots, DTB is loaded to kernel
   - DTB loaded to specific memory region.
   - Base address & offset address for each DTB file is known to kernel
5. While running, driver in kernel refers to DTB for HW specific properties
6. Generated DTB can be still debugged with dump command (last slide)

Master Boot Record (MBR) - Overview


N-sector disk drive. Each sector has 512 bytes.
Sector 0 Master Boot Record
Sector 1
Sector 2
Sector 3
...
Sector N-2
Sector N-1
Boot device (HDD, USB drive, eMMC, CD/DVD etc)
Master Boot Record (512 bytes)
Code (440 bytes)
Disk Signature (4 bytes)
Nulls (2 bytes)
Partition Table (four 16-byte entries, 64 bytes total)
MBR Signature (2 bytes)
0 Byte
512 Byte

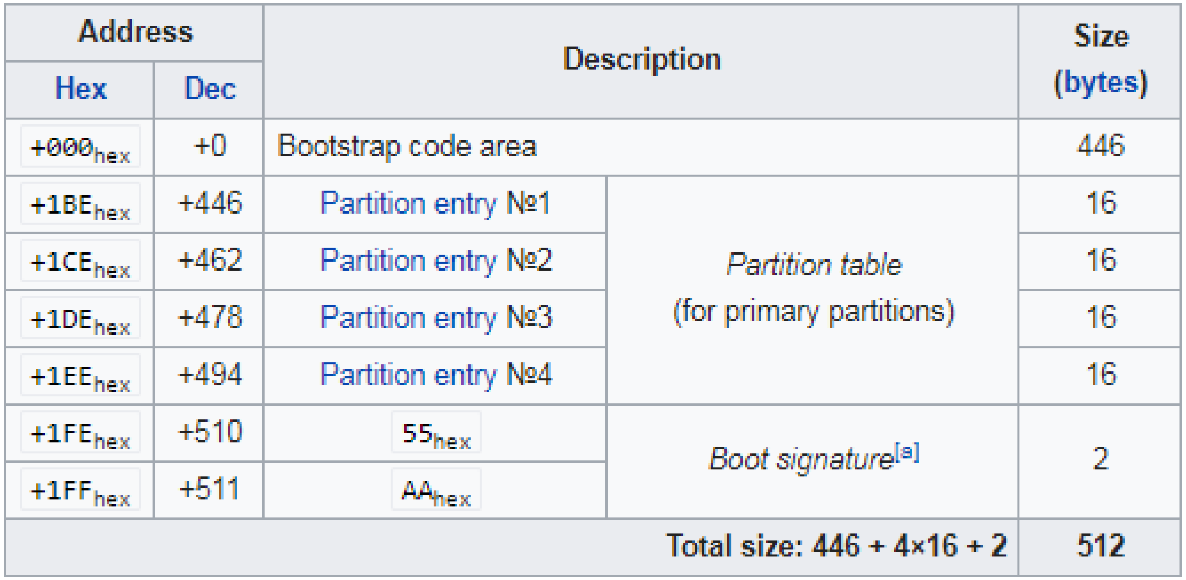
Every storage device has MBR regardless of:

- Boot priority
- Number of partitions in that storage device (even only 1. Max 4 partitions)

MBR main points:

- Hand over to next stage of bootloader
- Tells the location of target OS in its storage device to be booted
- Load its target booted OS into DDR

MBR (Master Boot Record) - Breakdown



Address		Description		Size (bytes)
Hex	Dec			
+000$_{\text{hex}}$	+0	Bootstrap code area		446
+1BE$_{\text{hex}}$	+446	Partition entry №1	*Partition table* (for primary partitions)	16
+1CE$_{\text{hex}}$	+462	Partition entry №2		16
+1DE$_{\text{hex}}$	+478	Partition entry №3		16
+1EE$_{\text{hex}}$	+494	Partition entry №4		16
+1FE$_{\text{hex}}$	+510	55$_{\text{hex}}$	*Boot signature*[a]	2
+1FF$_{\text{hex}}$	+511	AA$_{\text{hex}}$		
			Total size: 446 + 4×16 + 2	512

Bootstrap code: 446 Byte. Kicking off procedure to look up file from disk and load

- Contents: Assembly code to start BL (440B) + disk signature (4B) + RSVD (2B)

Partition Tables: Total 64 Byte. Each with 16 Byte. Containing information which partition to boot (Boot flag)

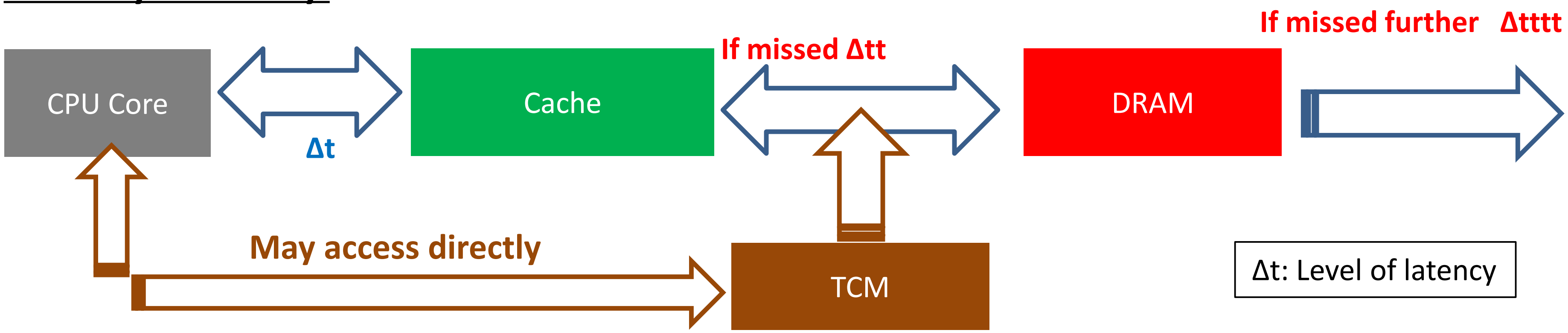
- Boot flag (1B): Status of physical drive. **Bit 7 needs to be set to be active and bootable**
- CHS partition address (3B): Partition start address with CHS (cyclic head sector)
- Partition Type (1B): i.e. FAT16 (0x04), NTFS, exFAT (x07), FAT32 with CHS (0x0B), etc
- CHS partition end address (3B)
- LBA partition start address (4B): With Logical block Address mapped from CHS
- LBA partition end address (4B)

Boot signature: 2 Byte. Validating bootability of this storage

- if 0x55 0xAA → Valid and proceed to boot from this device
- Else → not bootable storage and move to next priority level boot storage
- If no storage has 0x55 0xAA → system cannot boot

Tightly Coupled Memory (TCM)

Memory Hierarchy



Cache

- + First search place and makes little latency when target data is hit
- - Effective only when data is used frequently (or replace by other immediately)
- - Unsuitable for real time control which data is often exceptional (not repetitive)

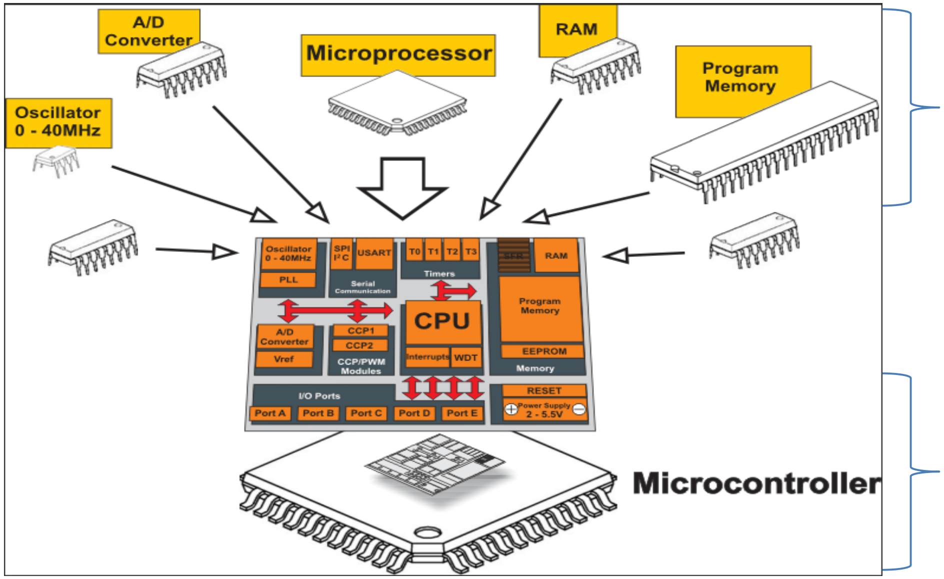
DRAM (or lower hierarchy)

- Too large latency to get data → Not suitable for real time processing
- Critical neck: Always start with referring to page table for address translation

TCM:

- Dedicated region of memory for real time process (interrupt, stack pt, vector, scratch pad etc)
- Using dedicated region of DRAM (physical address only) or (usually) dedicated embedded SRAM
- Can be user defined (size, what data to put) vs cache cannot be user defined
- Usually small size (4kB-256kB) but enough for RT but not for application which prefers larger (cache)
  - ➢ Trade off between size (SRAM size) and cost

SOC – MPU vs MCU



MPU (Equivalent to CPU)

- Mainly ALU, cache, registers and IOs
- Need peripheral devices to complete whole system

MCU

- Integration of MPU and peripherals in a package
- Feeding power to its unit makes whole system work

	MCU	MPU
Bit	8-32	32, 64
Operating Frequency	Few MHz ~ few*100MHz	Few*100MHz~Few GHz
Architecture	Mainly Pipelines	Superscaler
Memory	Mainly embedded (i.e TCM)	Mainly external (i.e DDR)
Instructions	Arithmetic, Logic	Arithmetic, Logic + DSP (SIMD), FP
Process Technology	Big (Peripheral, specially analogue cannot progress much)	Small
Price	Low	High
Target Usage	Small size, Low power, control	Higher performance applications (more complex than control)