ARM Processor Architecture

Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

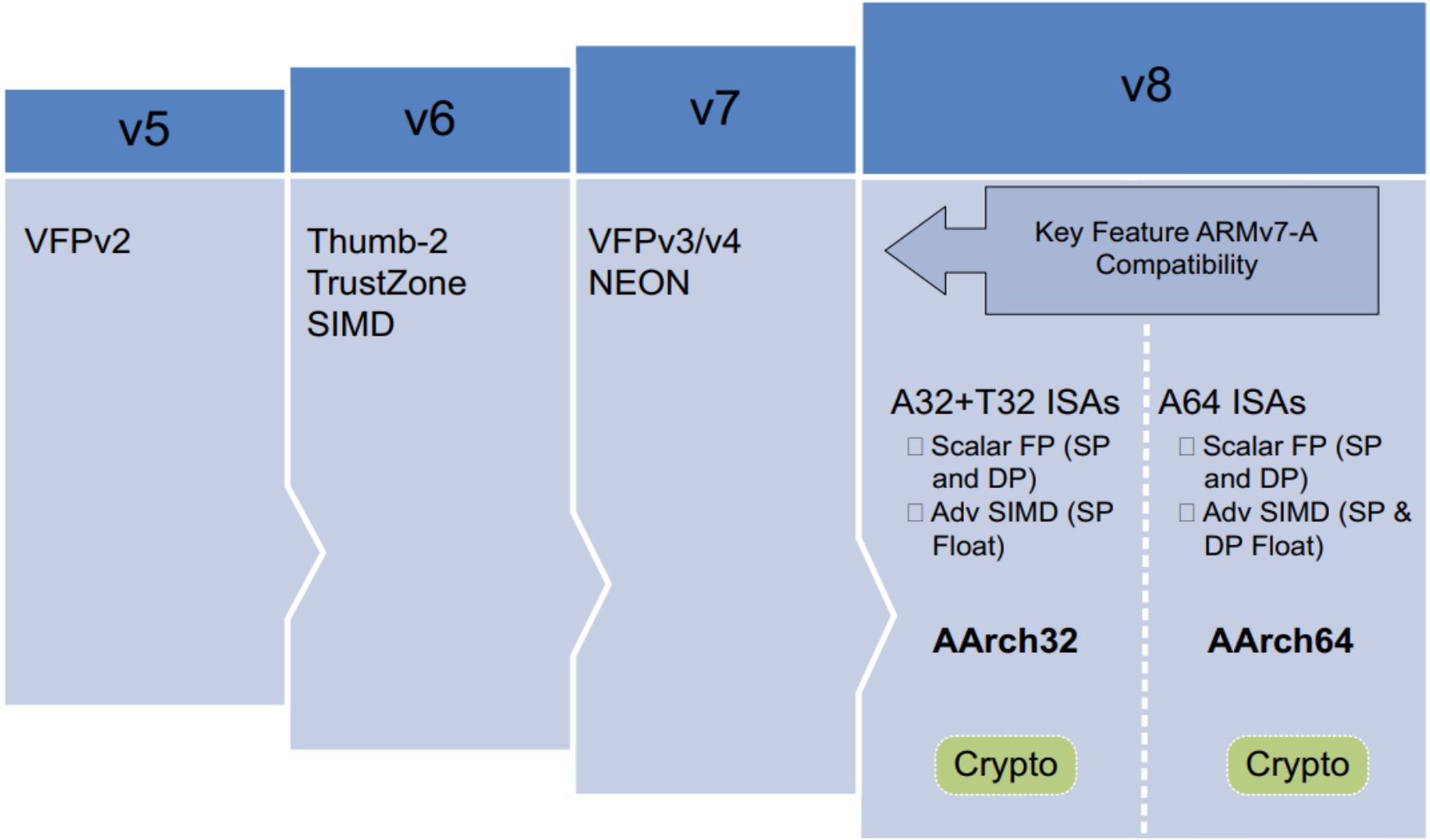
ARM Processor Architecture: Outline

- ARM Processor Architecture
- ARMv8 Exception and Privilege Levels
- ARMv8 Security States
- ARMv8 Execution States
- ARMv8 AArch64 Registers
- ARMv8 AArch64 Instruction Set Architecture
  - Data Transfer: Load and Store
  - Data Processing: Arithmetic, Logical, and Shift
  - Branch: Conditional and Unconditional

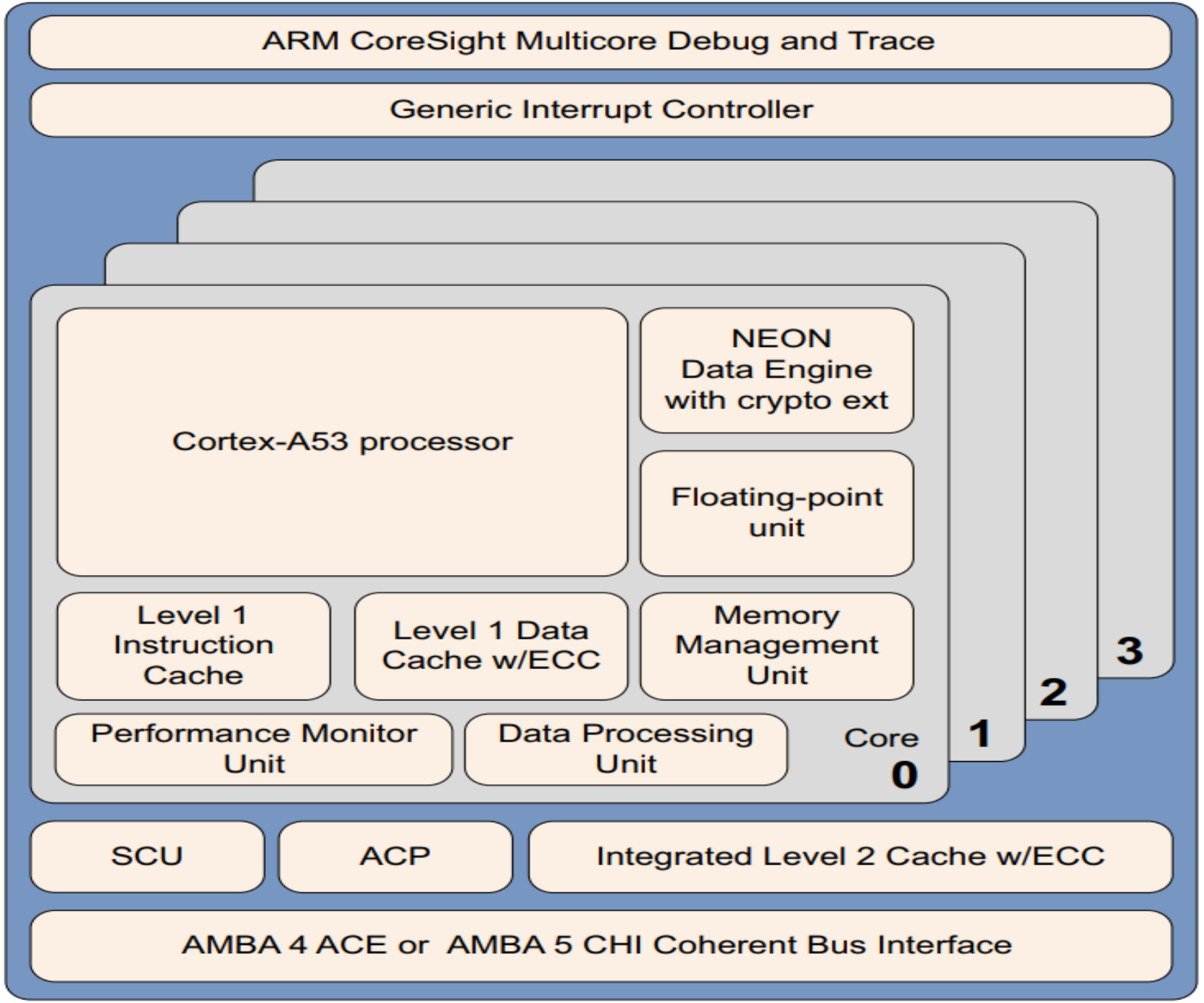
ARM Processor Architecture

- ARM, primarily a RISC system with the following attributes:
  - Moderate array of uniform registers
  - A load/store model of data processing in which operations only perform on operands in registers and not directly in memory
  - A uniform fixed-length instruction of 32 bits
  - Separate arithmetic logic unit (ALU), Multiply, and Shifter units
  - A small number of addressing modes with all load/store addresses determined from registers and instruction fields
  - Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops

Development of the ARMv8 architecture



ARMv8 (Cortex-A53) Processor

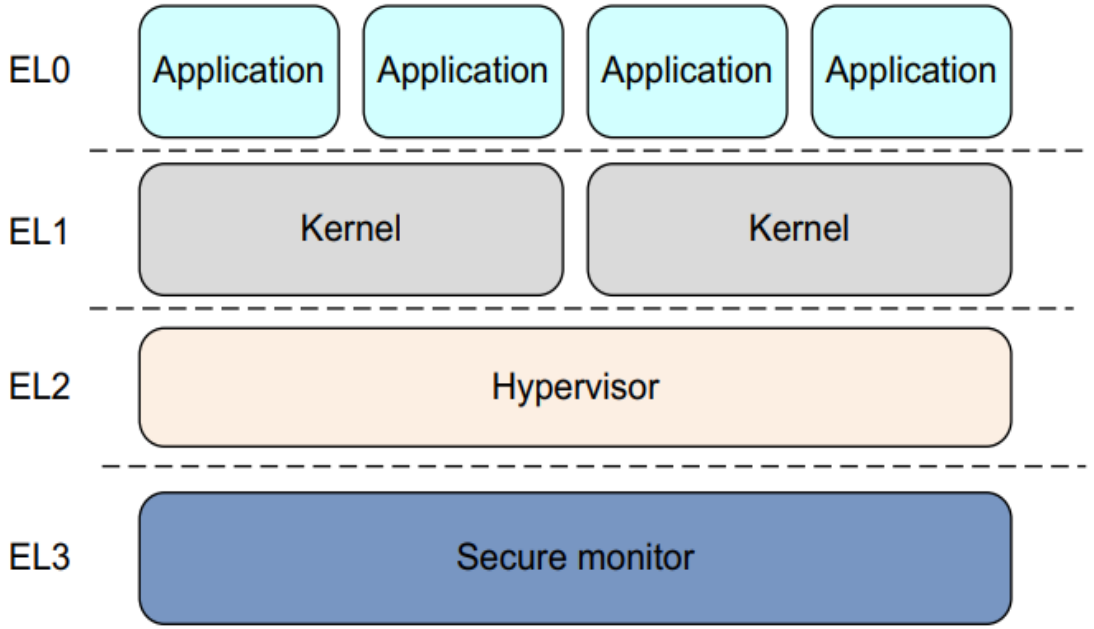


ARMv8 Exception and Privilege Levels

- Execution occurs in **4 Exception Levels**: **EL0**, **EL1**, **EL2**, and **EL3**.
- Execution in exception level **ELn** usually corresponds to privilege level **PLn**.
- The **higher the number** in **ELn/PLn** the **higher the privilege**.
- Multiple exception/privilege levels support the concept of hierarchical protection domains.

ARMv8 Exception Levels

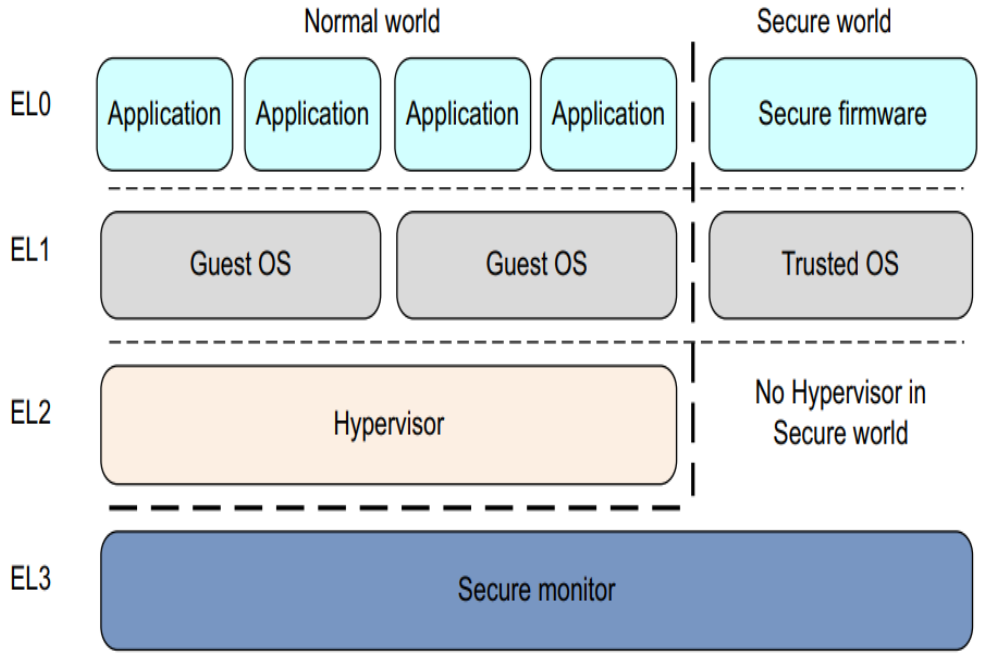
- Typical example of what software runs at each exception level:
  - EL0 Normal user applications, not privileged.
  - EL1 Operating system kernel, privileged.
  - EL2 Hypervisor, more privileged.
  - EL3 Low-level firmware, including the Secure Monitor, highly privileged



ARMv8 Security States

- Provides two **security states**: non secured state (**normal world**) and secured state (**secured world**)

- Enables **Guest OS** in **normal world** runs in parallel with a **Trusted OS** in **secured world** on the same hardware.
- Enables a **Hypervisor** running in **normal world** to host multiple **Guest OS** on the same hardware.
- **Secure Monitor** acts as a **gateway** between **normal** and **secured** world.



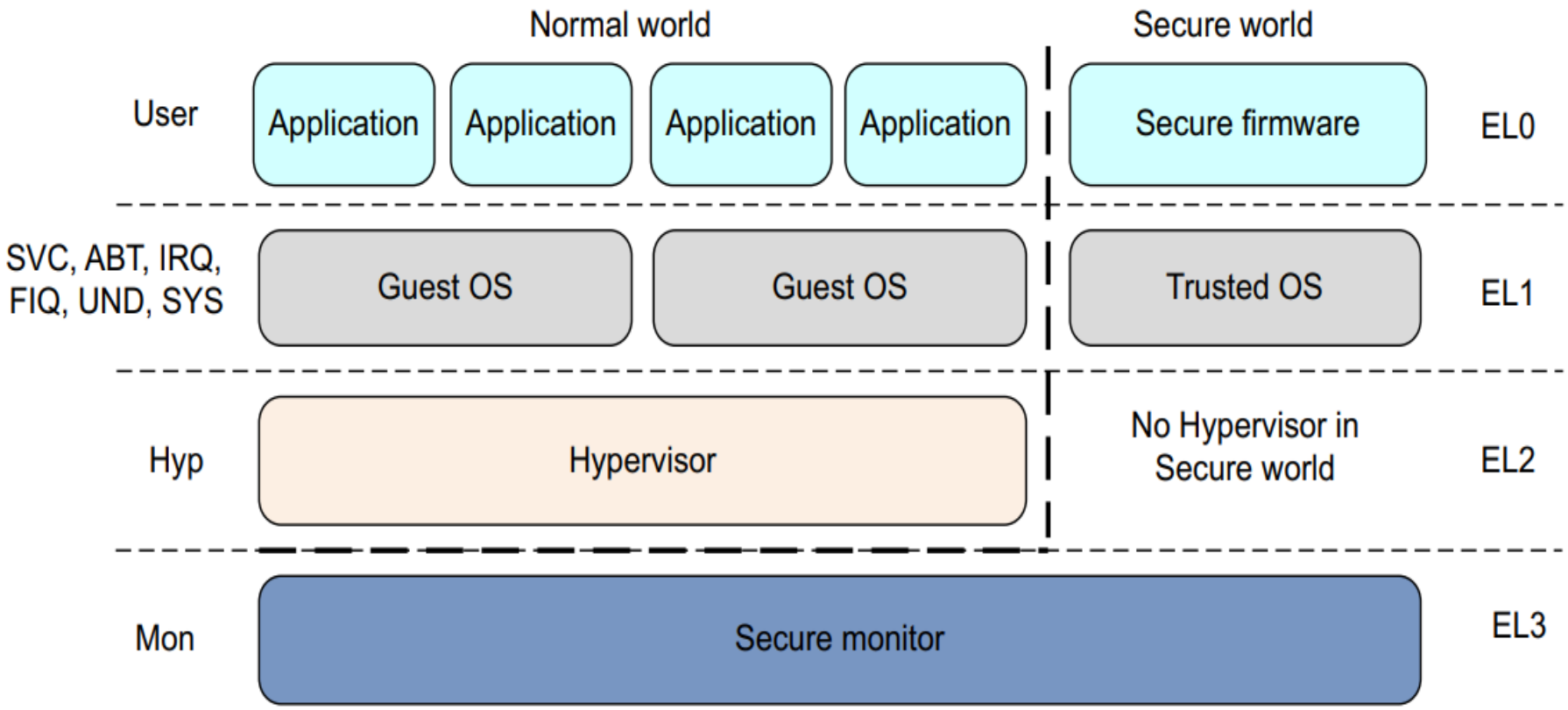
ARMv8 Processor Execution States

- An ARMv8 processor can be **configured** as executing in one of two **execution states**: **AArch32** (32 bits) and **AArch64** (64 bits).

- **AArch32**: **32-bit instructions**, **data**, and **address space**. Uses the **instruction set**, the **register set**, and the **processor modes** like **ARMv7**.

- **AArch64**: **32-bit instructions** but **64-bit data** and **address space**. Uses **enhanced instruction set**, a **larger register set** with **wider registers**, and the **processor modes** that are different from **ARMv7**.

AArch64 Exception Modes

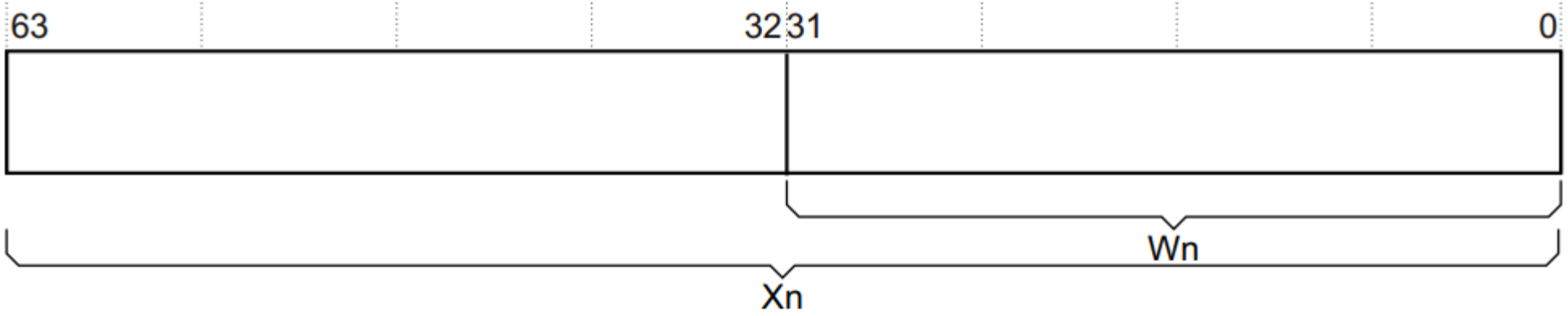


AArch64 Exception Modes

Mode	Function	Security state	Privilege level
User (USR)	Unprivileged mode in which most applications run	Both	PL0
FIQ	Entered on an FIQ interrupt exception	Both	PL1
IRQ	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	Entered when the SMC instruction (Secure Monitor Call) is executed or when the processor takes an exception which is configured for secure handling. Provided to support switching between Secure and Non-secure states.	Secure only	PL3
Abort (ABT)	Entered on a memory access exception	Both	PL1
Undef (UND)	Entered when an undefined instruction is executed	Both	PL1
System (SYS)	Privileged mode, sharing the register view with User mode	Both	PL1
Hyp (HYP)	Entered by the Hypervisor Call and Hyp Trap exceptions.	Non-secure only	PL2

AArch64 Register Set

- Provides **31 double word** (**64 bit**) general purpose registers, referred as **x0-x30**, accessible at all times and in all Exception Levels.
- Each **double word register** (x0-x30) can also be accessed as a **word register** (**w0-w30**).



AArch64 General Purpose Register Set

Name	Register number	Usage	Preserved on call?
X0–X7	0–7	Arguments/Results	no
X8	8	Indirect result location register	no
X9–X15	9–15	Temporaries	no
X16 (IP0)	16	May be used by linker as a scratch register; other times used as temporary register	no
X17 (IP1)	17	May be used by linker as a scratch register; other times used as temporary register	no
X18	18	Platform register for platform independent code; otherwise a temporary register	no
X19–X27	19–27	Saved	yes
X28 (SP)	28	Stack Pointer	yes
X29 (FP)	29	Frame Pointer	yes
X30 (LR)	30	Link Register (return address)	yes
XZR	31	The constant value 0	n.a.

AArch64 General Purpose Register Set

X0 – X7: Arguments or Results Registers
Arguments to a function are passed through x0 – x7 registers.
Results from a function are returned through x0 – x7 registers.

X8: Indirect Result Location Register
Results from a function is usually returned from a function to its caller through x0 to x7 registers if the function is returning directly to the caller.
There are some situations where a function does not call another function directly and the called function also does not return to the caller directly. For example, a user mode function calls a kernel function indirectly using a syscall and the kernel function does not return to the user mode function. User mode function passes a syscall number using x8 register to a trap handler. Trap handler calls an appropriate kernel function based on the syscall number. The kernel function returns the result to the user function using x8 register.

AArch64 General Purpose Register Set

X9 - X15: Temporary Registers
Functions are free to use these registers as temporary or scratch registers. Functions do not need to save and retrieve the current value of these registers before use and before return respectively.

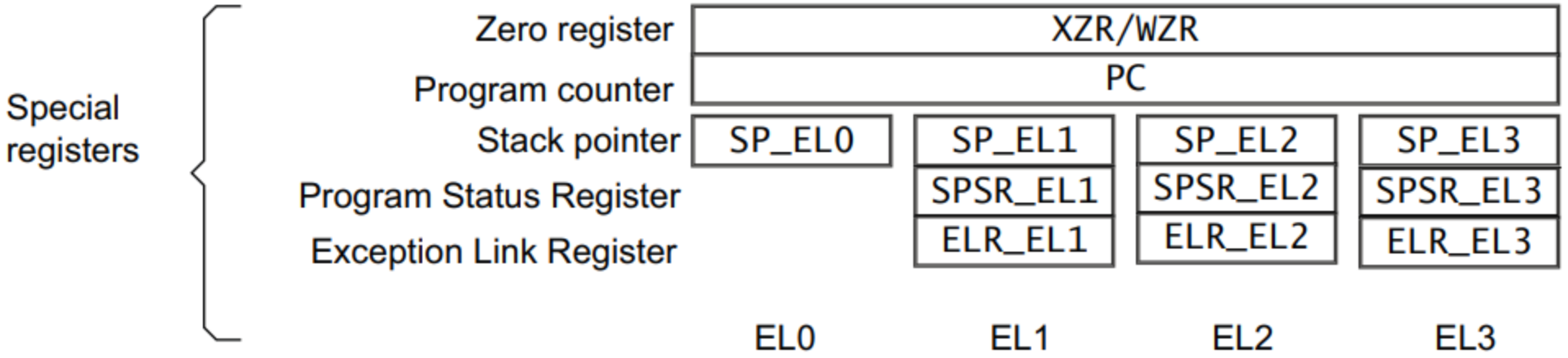
X16 and X17: Intra-Procedure Call Scratch Registers
Linker often inserts codes before and after function codes in order to facilitate a function call, which are called prologue and epilogue respectively.
Registers x16 and x17 can be used by the linker as the scratch registers in both prologue and epilogue codes.
Regular function codes should avoid using both x16 and x17.
If a regular function uses x16 and x17 as the temporary registers, it should be aware of the fact that the value it writes on these registers might not be seen by its called functions since the linker might have used them as the scratch registers and modifies their values in prologue or epilogue codes.

AArch64 General Purpose Register Set

X18: Platform Register
Register x18 is reserved for individual platform or OS specific codes in order to use it in platform specific way. Platform independent codes must avoid using it.
If a platform chooses not to use x18 as its platform specific special register, platform independent codes can use x18 register as a temporary register.

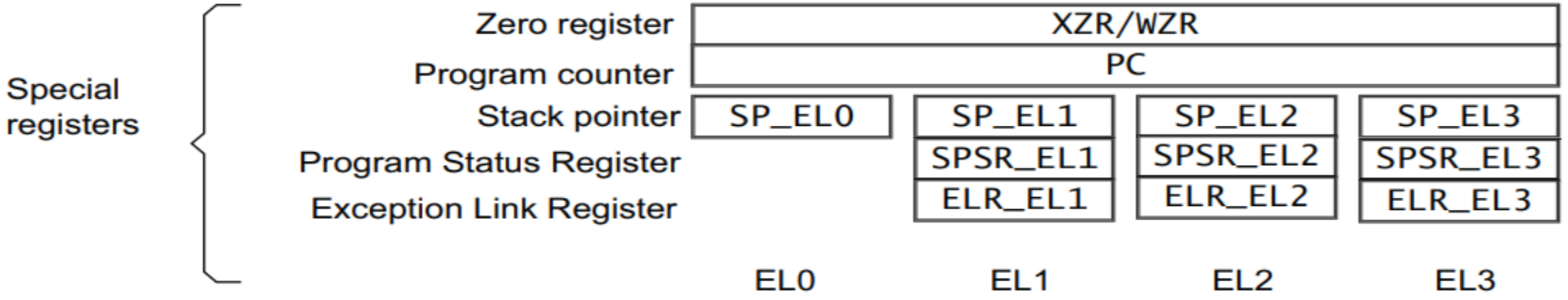
X19 – X27: Saved Register
A function must save the current value of these registers before using them and restore the saved value before returning. It guarantees that the values in these registers are preserved across multiple function calls.

AArch64 Special Registers: XZR and PC



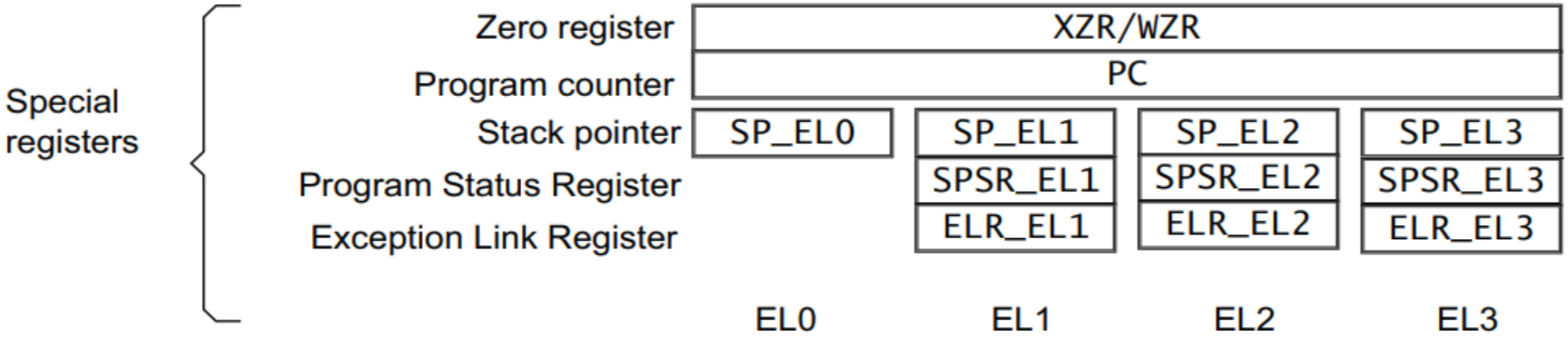
- There is **no actual register** numbered x31. **Reading** from **xzr** in code reads **64 bit zeros** and **writing** into **xzr** does not do anything.
- **PC** is not a general purpose register and is **not accessible** in code.

AArch64 Special Registers: Stack Pointer



- There are dedicated **stack pointers** (**SP_EL0**, **SP_EL1**, **SP_EL2**, and **SP_EL3**) for each exception level.
- **SP** refers to the **current stack pointer**.
- Program running in exception levels **EL1**, **EL2**, and **EL3** can use either the dedicated stack pointer or **EL0** stack pointer.
  - Suffix **t** (SP_ELn*t*) in the name indicates SP_EL0 has bee selected.
  - Suffix **h** (SP_ELn*h*) in the name indicates SP_Eln has been selected.

AArch64 Special Registers: Linked Register



- There are dedicated **linked register** (**ELR_EL1**, **ELR_EL2**, and **ELR_EL3**) for exception levels **EL1**, **EL2**, and **EL3**.
- Before taking into an exception level **return address** is saved into that level's **ELR**.
- The saved return address is used from the level's ELR at the return from the exception.

AArch64 PSTATE Fields

Name	Description
N	Negative Condition flag
Z	Zero Condition flag
C	Carry Condition flag
V	Overflow Condition flag
SS	Software Step bit
IL	IL Execution state bit
D	Debug Mask bit
A	SError mask bit
I	IRQ mask bit
F	FIQ mask bit
nRW	Execution Mode (0=64-bit, 1=32-bit)
EL(2)	Exception Level (00, 01, 10, 11)
SP	SP Selector (0= SP_EL0, 1= SP_ELn)

AArch64 PSTATE Fields

- There is no dedicated register like **CPSR** in **AArch32** to hold the **processor state** or **PSTATE Fields** in **AArch64,** instead they are **independently accessible** in **groups**.

- The **PSTATE.{N**, **Z**, **C**, **V}** fields can be accessed at **EL0**. All other PSTATE fields can be executed at **EL1** or higher and are UNDEFINED at EL0.

AArch64 PSTATE Fields

- **PSTATE** fields are accessed using **special-purpose registers**,

NZCV	Holds the condition flags
DAIF	Specifies the current interrupt mask bits.
CurrentEL	Holds the current Exception level.
SPSel	At EL1 or higher, this selects between the SP for the current Exception level (SP_Eln) and SP_EL0

- **Read** directly using **Move** to **Register** from **System** (**MRS)** instruction.

  MRS <Xt>, NZCV

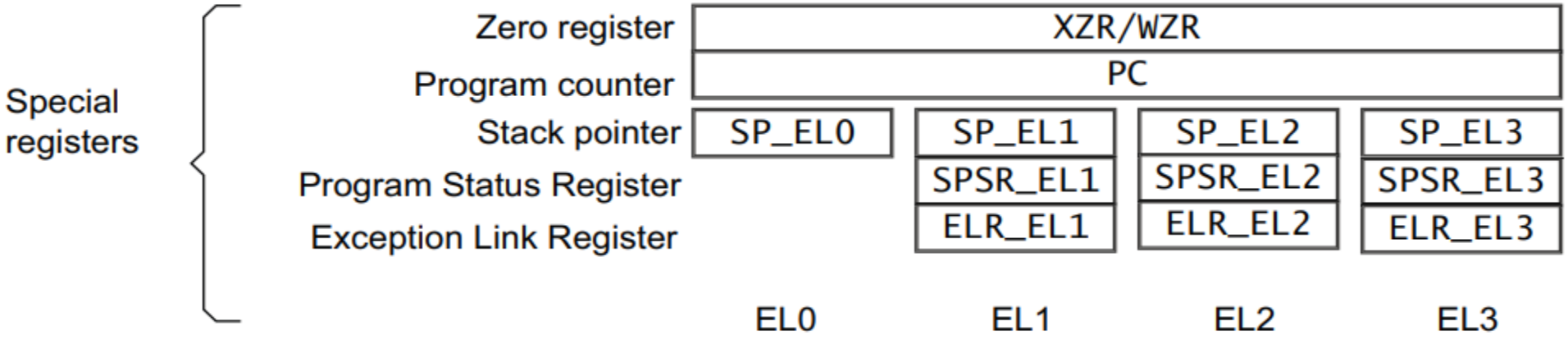
- **Written** directly using **Move** to **System** from **Register** (**MSR**) instruction.

  MSR DAIF, <Xt>

AArch64 Condition Codes on NZCV Condition Flags

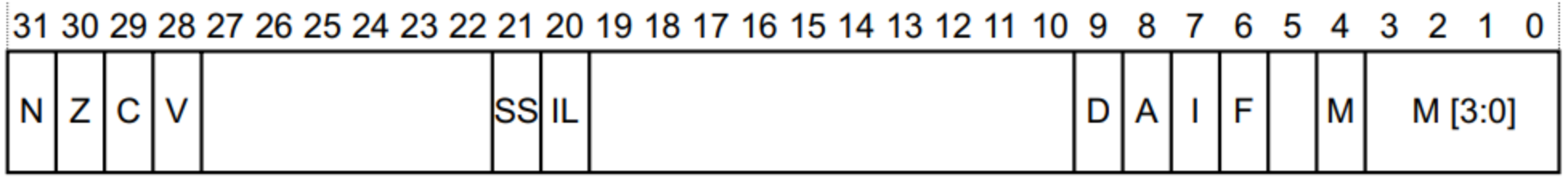
Code	Symbol	Condition Tested	Comment
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS/HS	C = 1	Carry set/unsigned higher or same
0011	CC/LO	C = 0	Carry clear/unsigned lower
0100	MI	N = 1	Minus/negative
0101	PL	N = 0	Plus/positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 AND Z = 0	Unsigned higher
1001	LS	C = 0 OR Z = 1	Unsigned lower or same
1010	GE	N = V [(N = 1 AND V = 1) OR (N = 0 AND V = 0)]	Signed greater than or equal
1011	LT	N ≠ V [(N = 1 AND V = 0) OR (N = 0 AND V = 1)]	Signed less than
1100	GT	(Z = 0) AND (N = V)	Signed greater than
1101	LE	(Z = 1) OR (N ≠ V)	Signed less than or equal
1110	AL	—	Always (unconditional)
1111	—	—	This instruction can only be executed unconditionally

AArch64 Special Registers: SPSR



- There are dedicated **saved program status register** (**SPSR_EL1**, **SPSR_EL2**, and **SPSR_EL3**) for exception levels **EL1**, **EL2**, and **EL3**.
- Before taking into an exception level **current processor state (PSTATE)** is saved into that level's **SPSR**.
- The saved PSTATE is restored from the level's SPSR at the return from the exception.

AArch64 Special Registers: SPSR



N Negative result (N flag).

Z Zero result (Z) flag.

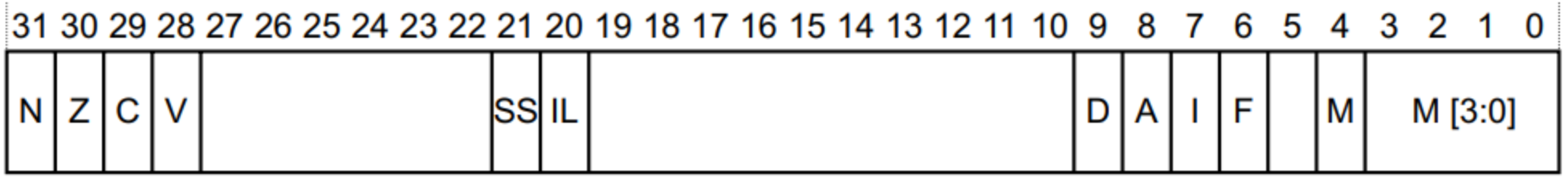
C Carry out (C flag).

V Overflow (V flag).

SS Software Step. Indicates whether software step was enabled when an exception was taken.

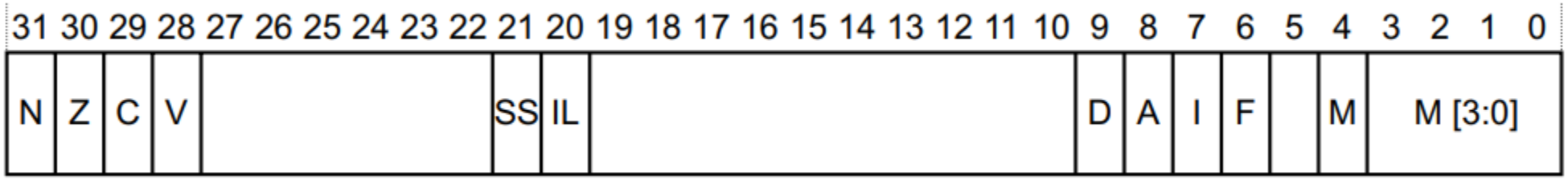
IL Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.

AArch64 Special Registers: SPSR



D	Process state Debug mask. Indicates whether debug exceptions from watchpoint, breakpoint, and software step debug events that are targeted at the Exception level the exception occurred in were masked or not.
A	SError (System Error) mask bit.
I	IRQ mask bit.
F	FIQ mask bit.
M[4]	Execution state that the exception was taken from. A value of 0 indicates AArch64.
M[3:0]	Mode or Exception level that an exception was taken from.

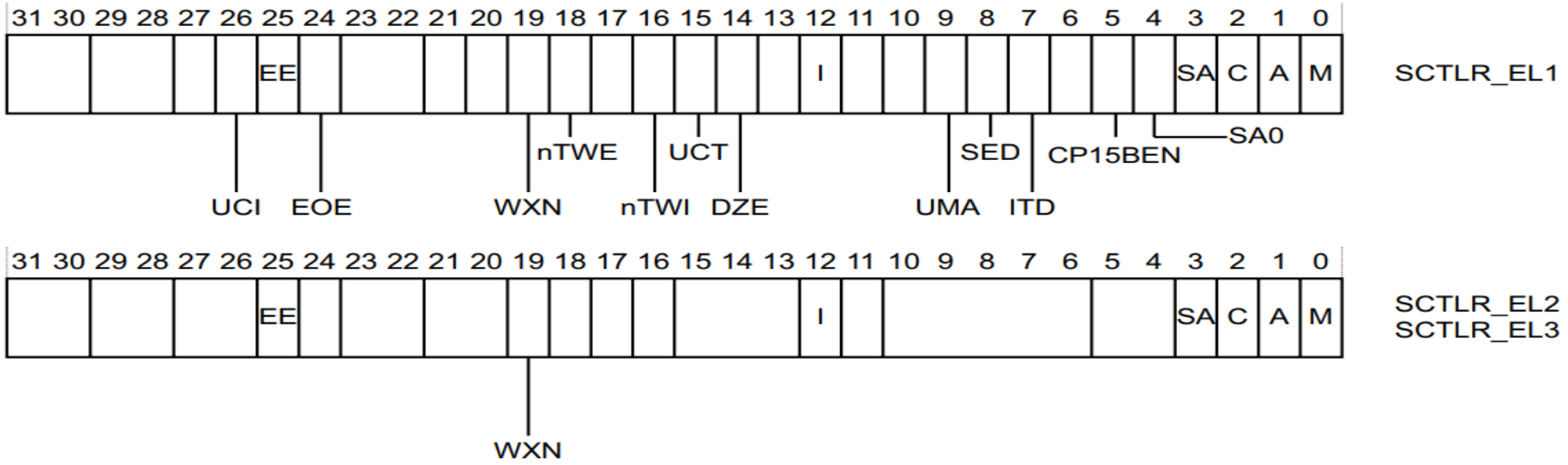
AArch64 Special Registers: SPSR



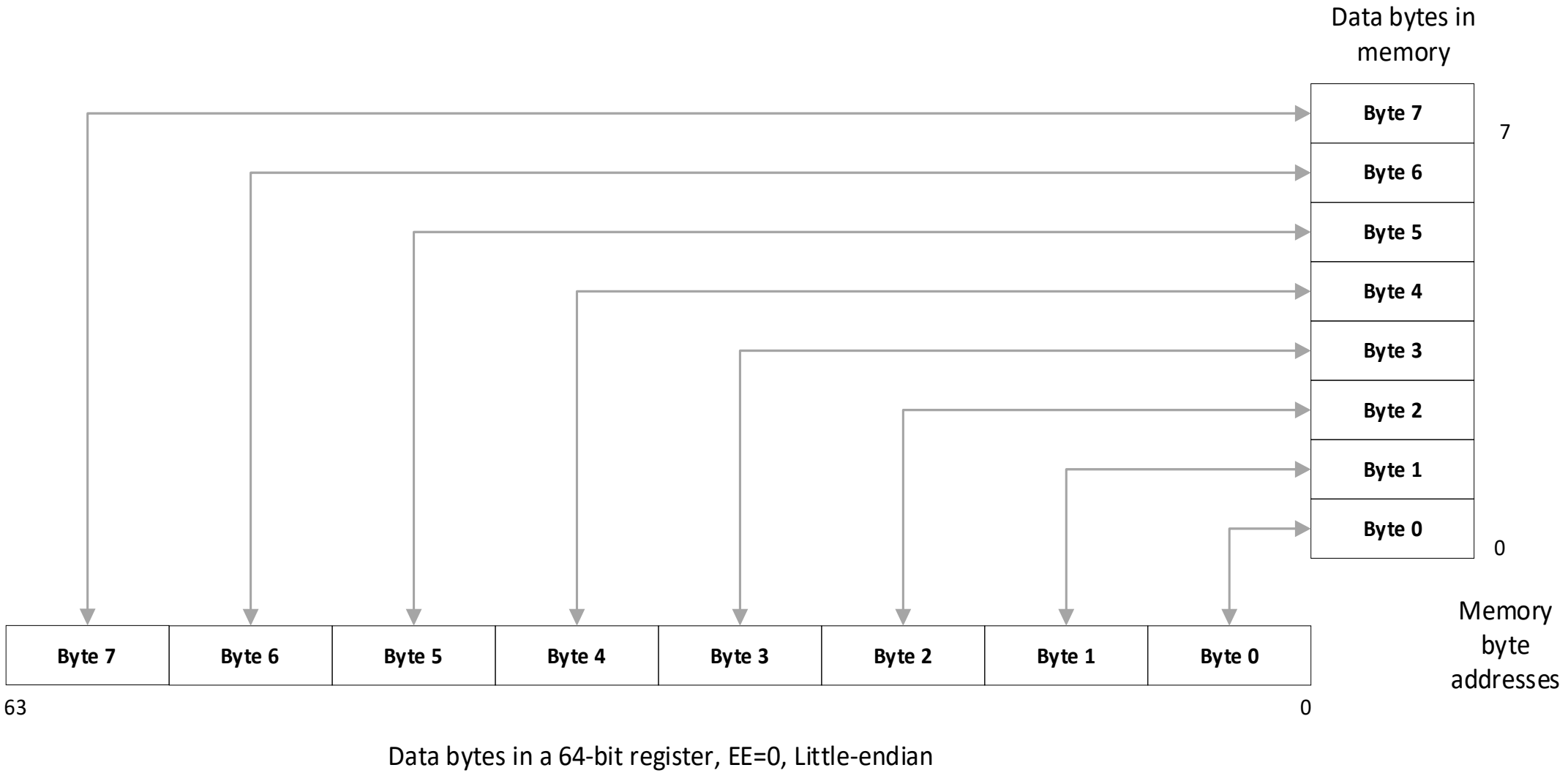
M[3:0]	Meaning
0b0000	EL0.
0b0100	EL1 with SP_EL0 (ELt).
0b0101	EL1 with SP_EL1 (EL1h).
0b1000	EL2 with SP_EL0 (EL2t).
0b1001	EL2 with SP_EL2 (EL2h).

AArch64 System Control Register

- The **System Control Register** (**SCTLR**) is a register that **controls standard memory**, **system facilities** and provides status information for functions that are implemented in the core.
- **SCTLR** is **read** and **written** using **MRS** instructions.
- **SCTLR** can be configured with **EE** = **0** for **little-endian** and **EE** = **1** for big-endian.

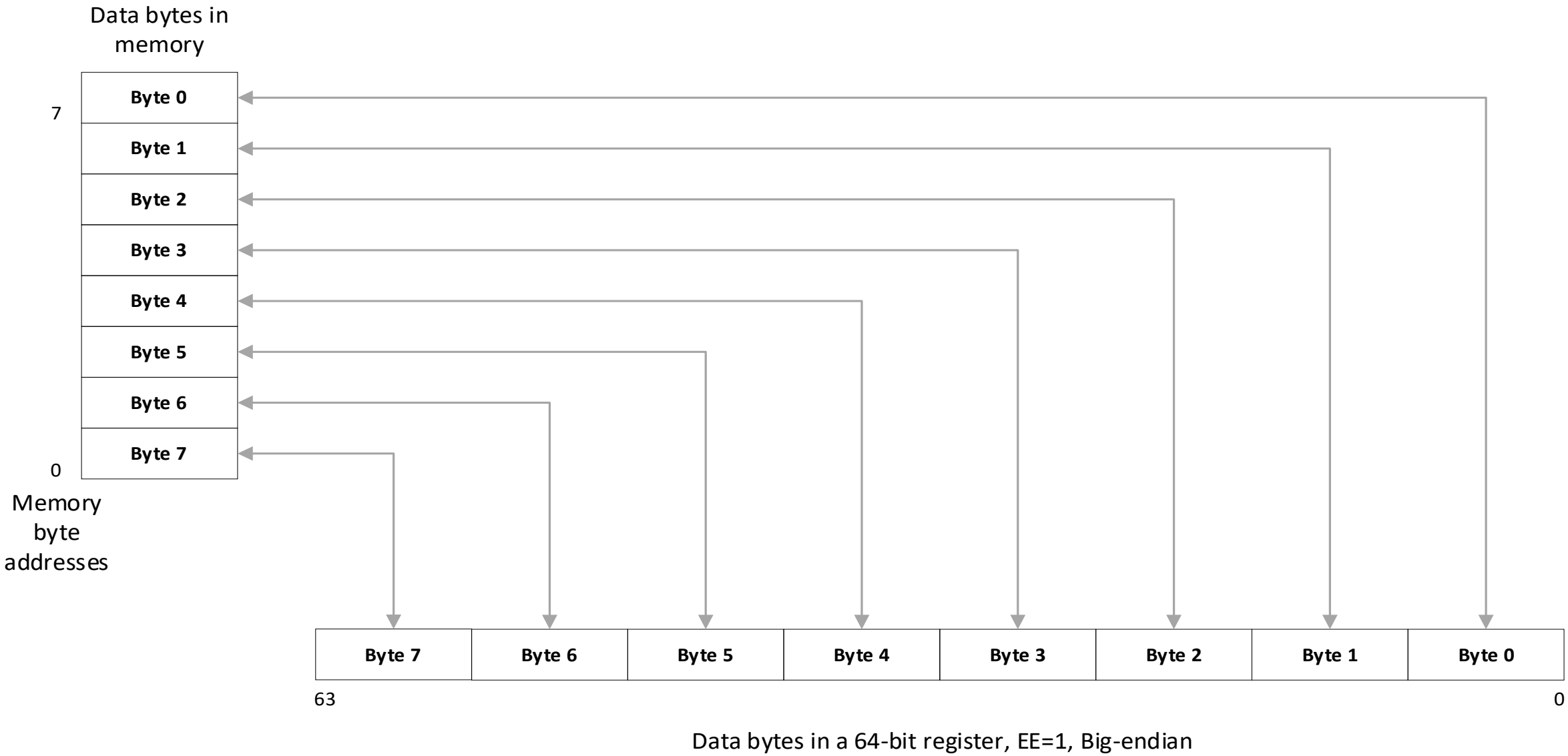


AArch64 Little-Endian



Data bytes in a 64-bit register, EE=0, Little-endian

AArch64 Big-Endian



Data bytes in a 64-bit register, EE=1, Big-endian

ARM Processor 64-bit Architecture Word Register Set

- ARM64 assembly language ***overloads*** instruction mnemonics and distinguishes between the different forms of an instruction based on the **operand register names**.
- For example, the ADD instructions below all have different opcodes, but the programmer only has to remember one mnemonic and the assembler automatically chooses the correct opcode based on the operands.

```
ADD W0, W1, W2          // add 32-bit register
ADD X0, X1, X2          // add 64-bit register
ADD X0, X1, #42         // add 64-bit immediate
```

ARM8 A64 Data Transfer Instructions

Load and Store Instructions		
LDP	rt, rt2, [addr]	$rt2{:}rt = [addr]_{2N}$
LDPSW	Xt, Xt2, [addr]	$Xt = [addr]^{\pm}_{32}$; $Xt2 = [addr+4]^{\pm}_{32}$
LD{U}R	rt, [addr]	$rt = [addr]_{N}$
LD{U}R{B,H}	Wt, [addr]	$Wt = [addr]^{\emptyset}_{N}$
LD{U}RS{B,H}	rt, [addr]	$rt = [addr]^{\pm}_{N}$
LD{U}RSW	Xt, [addr]	$Xt = [addr]^{\pm}_{32}$
PRFM	prfop, addr	Prefetch(addr, prfop)
STP	rt, rt2, [addr]	$[addr]_{2N} = rt2{:}rt$
ST{U}R	rt, [addr]	$[addr]_{N} = rt$
ST{U}R{B,H}	Wt, [addr]	$[addr]_{N} = Wt_{N0}$

Addressing Modes (addr)		
xxP,LDPSW	[Xn{, #i_{7+s}}]	$addr = Xn + i^{\pm}_{6+s:s}{:}0_{s}$
xxP,LDPSW	[Xn], #i_{7+s}	$addr{=}Xn$; $Xn{+}{=}i^{\pm}_{6+s:s}{:}0_{s}$
xxP,LDPSW	[Xn, #i_{7+s}]!	$Xn{+}{=}i^{\pm}_{6+s:s}{:}0_{s}$; $addr{=}Xn$
xxR*,PRFM	[Xn{, #i_{12+s}}]	$addr = Xn + i^{\emptyset}_{11+s:s}{:}0_{s}$
xxR*	[Xn], #i_{9}	$addr = Xn$; $Xn \mathrel{+}= i^{\pm}$
xxR*	[Xn, #i_{9}]!	$Xn \mathrel{+}= i^{\pm}$; $addr = Xn$
xxR*,PRFM	[Xn,Xm{, LSL #0|s}]	$addr = Xn + Xm \ll s$
xxR*,PRFM	[Xn,Wm,{S,U}XTW{ #0|s}]	$addr = Xn + Wm^{?} \ll s$
xxR*,PRFM	[Xn,Xm,SXTX{ #0|s}]	$addr = Xn + Xm^{\pm} \ll s$
xxUR*,PRFM	[Xn{, #i_{9}}]	$addr = Xn \mathrel{+}= i^{\pm}$
LDR{SW},PRFM	$\pm rel_{21}$	$addr = PC + rel^{\pm}_{20:2}{:}0_{2}$

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Load

```
ldr x7, [x3, x2]        //Load x7 with a double word (8 bytes) from the address [x3 + x2],
                        //the address [x3 +x2] must be multiple of 8.

ldr x7, [x3, #8]        //Load x7 with a double word (8 bytes) from the address [x3 + 8],
                        //the address [x3 + #8] must be multiple of 8.

ldr x7, [x3, x2]!       //Load x7 with a double word (8 bytes) from the address [x3 + x2],
                        //then store the address in x3, pre-indexed,
                        //the address [x2, x2] must be multiple of 8.

ldr x9, [x2, #8]!       //Load x9 with a double word (8 bytes) from the address [x2 + 8],
                        //then store the address in x2, pre-indexed,
                        //the address [x2, #8] must be multiple of 8.

ldr x7, [x3], #8        //Load x7 with a double word (8 bytes) from the address in [x3],
                        //then increment x3 by 8, post-indexed,
                        //the address [x3] must be multiple of 8.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Load

```
ldr w7, [x3, x2]        //Load x7 with a word (4 bytes) from the address [x3 + x2],
                        //the address [x3 +x2] must be multiple of 4.

ldr w7, [x3, #4]        //Load x7 with a word (4 bytes) from the address [x3 + 4],
                        //the address [x3 + #4] must be multiple of 4.

ldr w7, [x3, x2]!       //Load x7 with a word (4 bytes) from the address [x3 + x2],
                        //then store the address in x3, pre-indexed,
                        //the address [x2, x2] must be multiple of 4.

ldr w9, [x2, #4]!       //Load x9 with a word (4 bytes) from the address [x2 + 4],
                        //then store the address in x2, pre-indexed,
                        //the address [x2, #4] must be multiple of 4.

ldr w7, [x3], #4        //Load x7 with a word (4 bytes) from the address in [x3],
                        //then increment x3 by 4, post-indexed,
                        //the address [x3] must be multiple of 4.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Half-word and Byte Load

```
ldrh x9, [x2, #2]!      //Load x9 with the half-word (2 bytes) from the address [x2 + 2] and zero extend x9,
                        //then store the address in x2, pre-indexed,
                        //the address [x2 + 2] must be multiple of 2.

ldrsh x5, [x2]          //Load x5 with the half-word (2 bytes) from the address [x2] and sign extend x5.
                        //the address [x2 + 2] must be multiple of 2.

ldrb x9, [x2, #1]!      //Load x9 with 1 byte from the address [x2 + 1] and zero extend x9,
                        //then store the address in x2, pre-indexed,
                        //the address [x2, #1] must be multiple of 1.

ldrsb x5, [x2]          //Load x5 with 1 byte from the address [x2] and sign extend x5,
                        //the address [x2] must be multiple of 1.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Store

```
str x7, [x3, x2]        //Store x7 (double word) at the address [x3 + x2]
                        //the address must be multiple of 8.

str x7, [x3, #8]        //Store x7 (double word) at the address [x3 + 8]
                        //the address must be multiple of 8.

str x7, [x3, x2]!       //Store x7 (double word) at the address [x3 + x2], then
                        //store the address in x3, pre-indexed, the address must be multiple of 8.

str x9, [x2, #8]!       //Store x9 (double word) at the address [x2 + 8], then store
                        //the address in x2, pre-indexed, the address must be multiple of 8.

str x7, [x3], #8        //Store x7 (double word) at the address [x3] then
                        //increment x3 by 8, post-indexed, the address must be multiple of 8.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Store

```
str w7, [x3, w2]        //Store w7 (word) at the address [x3 + w2)], the address must be multiple of 4.

str w7, [x3, #4]        //Store w7 (word) at the address [x3 + 4], the address must be multiple of 4.

str w7, [x3, w2]!       //Store w7 (word) at the address [x3 + w2], then
                        //store the address in x3, pre-indexed, the address must be multiple of 4.

str w9, [x2, #4]!       //Store w9 (word) at the address [x2 + 4], then store
                        //the address in x2, pre-indexed, the address must be multiple of 4.

str w7, [x3], #4        //Store w7 (word) at the address [x3] then
                        //increment x3 by 4, post-indexed, the address must be multiple of 4.
```

ARM8 A64 Data Transfer Instructions: 64-bit Scaled Half-word and Byte Load

```
strh x9, [x2, #2]!      //Store the low half-word from x9 to the address [x2 + 2] and then store the address
                        //in x2, pre-indexed, the address must be multiple of 2.

strb x9, [x2, #1]!      //Store the low byte from x9 to the address [x2 + 1] and then store the address in x2,
                        //pre-indexed, the address must be multiple of 1.

strh w9, [x2, #2]!      //Store the low half-word from w9 to the address [x2 + 2] and then store the address
                        //in x2, pre-indexed, the address must be multiple of 2.

strb w9, [x2, #1]!      //Store the low byte from w9 to the address [x2 + 1] and then store the address in x2,
                        //pre-indexed, the address must be multiple of 1.
```

ARM8 A64 Data Transfer Instructions: Unscaled and Scaled

- Scaled
  - **LDR** Wt/Xt, [Xn|SP{, **#pimm**}]
  - **STR** Wt/Xt, [Xn|SP{, **#pimm**}]
  - **pimm** value range:
    - 32-bit: **0 ~ 16380**, and **pimm % 4 == 0** (is a multiple of 4)
    - 64-bit: **0 ~ 32760**, and **pimm % 8 == 0** (is a multiple of 8)
    - NEED to be a multiple of data access size, e.g., 4 or 8
- Unscaled
  - **LDUR** Wt/Xt, [Xn|SP{, #**simm**}]
  - **STUR** Wt/Xt, [Xn|SP{, #**simm**}]
  - **simm** value range: **-256 ~ 255**
    - NO need to be a multiple of data access size.

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Load

```
ldur x7, [x3, x2]       //Load x7 with double word from the address [x3 + x2]

ldur x7, [x3, #2]       //Load x7 with double word from the address [x3 + 2]

ldur x7, [x3, x2]!      //Load x7 with double word from the address [x3 + x2], then
                        //store the address in x3, pre-indexed

ldur x9, [x2, #2]!      //Load x9 with double word from the address [x2 + 2], then
                        //store the address in x2, pre-indexed

ldur x7, [x3], #2       //Load x7 with double word from the address [x3] then
                        //increment x3 by 2, post-indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Half-word and Byte Load

```
ldurh x9, [x2, #2]!     //Load x9 with the half-word from the address [x2 + 2] and
                        //zero extend x9, then store the address in x2, pre-indexed

ldursh x5, [x2]         //Load x5 with the half-word from the address [x2] and sign
                        //extend x5.

ldurb x9, [x2, #1]!     //Load x9 with the byte from the address [x2 + 1] and zero
                        //extend x9, then store the address in x2, pre-indexed

ldursb x5, [x2]         //Load x5 with the byte from the address [x2] and sign
                        //extend x5.
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Store

```
stur x7, [x3, x2]        //Store x7 (double word) at the address [x3 + x2]

stur x7, [x3, #2]        //Store x7 (double word) at the address [x3 + 2]

stur x7, [x3, x2]!       //Store x7 (double word) at the address [x3 + x2], then
                         //store the address in x3, pre-indexed

stur x9, [x2, #2]!       //Store x9 (double word) at the address [x2 + 2], then store
                         //the address in x2, pre-indexed

stur x7, [x3], #2        //Store x7 (double word) at the address [x3] then
                         //increment x3 by 2, post indexed
```

ARM8 A64 Data Transfer Instructions: 64-bit Unscaled Half-word and Byte Store

```
sturh x9, [x2, #2]!     //Store half-word of x9 at the address [x2 + 2], then store
                        //the address in x2, pre-indexed

sturb x9, [x2, #1]!     //Store byte of x9 at the address [x2 + 1], then store
                        //the address in x2, pre-indexed

strh x9, [x2, #2]!      //Store half-word of x9 at the address [x2 + 2], then store
                        //the address in x2, pre-indexed

strb x9, [x2, #1]!      //Store byte of x9 at the address [x2 + 1], then store
                        //the address in x2, pre-indexed
```

ARM8 A64 Data Transfer Instructions: Literal Pool Load

```
ldr x9, =label
        //loads memory address referred by label to X9
```

```
                .data
number:         .word 100

                .text
                ldr x2, =number         //Loads the address of number into x2
                ldr x2, [x2]            //Loads the value of number into x2.
```

ARM8 A64 Arithmetic Instructions

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	

Operand 2 (op2)		
all	*rm*	rm
all	rm, LSL #i_6	rm ≪ i
all	rm, LSR #i_6	rm ≫ i
all	rm, ASR #i_6	rm $\bar{\gg}$ i
logical	rm, ROR #i_6	rm ⋙ i
arithmetic	Wm, {S,U}XTB{ #i_3}	$Wm^{?}_{B0}$ ≪ i
arithmetic	Wm, {S,U}XTH{ #i_3}	$Wm^{?}_{H0}$ ≪ i
arithmetic	Wm, {S,U}XTW{ #i_3}	$Wm^{?}$ ≪ i
arithmetic	Xm, {S,U}XTX{ #i_3}	$Xm^{?}$ ≪ i
arithmetic	#i_{12}	$i^{\emptyset}$
arithmetic	#i_{24}	$i^{\emptyset}_{23:12}:0_{12}$
AND,EOR,ORR,TST	#mask	mask

Keys	
N	Operand bit size (8, 16, 32 or 64)
s	Operand log byte size (0=byte,1=hword,2=word,3=dword)
rd, rn, rm, rt	General register of either size (Wn or Xn)
prfop	P{LD,LI,ST}L{1..3}{KEEP,STRM}
{,sh}	Optional halfword left shift (LSL #{16,32,48})
$val^{\pm}$, $val^{\emptyset}$, $val^{?}$	Value is sign/zero extended (? depends on instruction)
$\bar{\times}$ $\bar{\div}$ $\bar{\gg}$ $\bar{>}$ $\bar{<}$	Operation is signed

ARM8 A64 Arithmetic Logic Instructions

```
add x0, x1, x2          //x0=x1+x2
adds x0, x1, x2         //x0=x1+x2, and set pstate flags
adc x0, x1, x2          //x0=x1+x2+carry
adcs x0, x1, x2         //x0=x1+x2+carry, and set pstate flags

sub x0, x1, x2          //x0=x1-x2
subs x0, x1, x2         //x0=x1-x2, and set pstate flags
sbc x0, x1, x2          //x0=x1-x2-1+carry
sbcs x0, x1, x2         //x0=x1-x2-1+carry, and set pstate flags

cmp x0, #imm            //compare x0 with #imm
cmp x0, x1              //compare x0 with x1
```

ARM8 A64 Arithmetic Logic Instructions

```
mul w0, w1, w2      //w0=w1*w2          32-bit = 32-bit x 32-bit, lower 32 bits of result
mul x0, x1, x2      //x0=x1*x2          64-bit = 64-bit x 64-bit, lower 64 bits of result

smulh x0, x1, x2    //x0=x1*x2, treats source operands as signed, upper 64 bits of result
umulh x0, x1, x2    //x0=x1*x2, treats source operands as unsigned, upper 64 bits of result

smull x0, w1, w2    //x0=w1*w2, treats source operands as signed, 64-bit = 32-bit x 32-bit
umull x0, w1, w2    //x0=w1*w2, treats source operands as unsigned, 64-bit = 32-bit x 32-bit

sdiv w0, w1, w2     //w0=w1÷w2, treats source operadns as signed
sdiv x0, x1, x2     //x0=x1÷x2, treats source operadns as signed
udiv w0, w1, w2     //w0=w1÷w2, treats source oprands as unsigned
udiv x0, x1, x2     //x0=x1÷x2, treats source oprands as unsigned
```

ARM8 A64 Logical Instructions

Logical and Move Instructions			
AND{S}	rd, rn, op2	rd = rn & op2	
ASR	rd, rn, rm	rd = rn $\gg$ rm	
ASR	rd, rn, #i_6	rd = rn $\bar{\gg}$ i	
BIC{S}	rd, rn, op2	rd = rn & $\sim$op2	
EON	rd, rn, op2	rd = rn $\oplus$ $\sim$op2	
EOR	rd, rn, op2	rd = rn $\oplus$ op2	
LSL	rd, rn, rm	rd = rn $\ll$ rm	
LSL	rd, rn, #i_6	rd = rn $\ll$ i	
LSR	rd, rn, rm	rd = rn $\gg$ rm	
LSR	rd, rn, #i_6	rd = rn $\gg$ 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 = $\sim$op2	
ORN	rd, rn, op2	rd = rn | $\sim$op2	
ORR	rd, rn, op2	rd = rn | op2	
ROR	rd, rn, #i_6	rd = rn $\ggg$ i	
ROR	rd, rn, rm	rd = rn $\ggg$ rm	
TST	rn, op2	rn & op2	

Operand 2 (op2)		
all	*rm*	rm
all	rm, LSL #i_6	rm $\ll$ i
all	rm, LSR #i_6	rm $\gg$ i
all	rm, ASR #i_6	rm $\bar{\gg}$ i
logical	rm, ROR #i_6	rm $\ggg$ i
arithmetic	Wm, {S,U}XTB{ #i_3}	$Wm^{?}_{B0} \ll$ i
arithmetic	Wm, {S,U}XTH{ #i_3}	$Wm^{?}_{H0} \ll$ i
arithmetic	Wm, {S,U}XTW{ #i_3}	$Wm^{?} \ll$ i
arithmetic	Xm, {S,U}XTX{ #i_3}	$Xm^{?} \ll$ i
arithmetic	#i_{12}	$i^{\emptyset}$
arithmetic	#i_{24}	$i^{\emptyset}_{23:12}:0_{12}$
AND,EOR,ORR,TST	#mask	mask

Keys	
N	Operand bit size (8, 16, 32 or 64)
s	Operand log byte size (0=byte,1=hword,2=word,3=dword)
rd, rn, rm, rt	General register of either size (Wn or Xn)
prfop	P{LD,LI,ST}L{1..3}{KEEP,STRM}
{,sh}	Optional halfword left shift (LSL #{16,32,48})
$val^{\pm}$, $val^{\emptyset}$, $val^{?}$	Value is sign/zero extended (? depends on instruction)
$\bar{\times}$ $\bar{\div}$ $\bar{\gg}$ $\bar{>}$ $\bar{<}$	Operation is signed

ARM8 A64 Logical Instructions

```
and x0, x1, #bimm64        //x0 = x1 & #bimm64
and w0, w1, #bimm32        //w0 = w1 & #bimm32
and w0, w1, w2             //w0 = w1 & w2
ands x0, x1, x2            //x0 = x1 & x2, and set flags (clears C and V)

orr x0, x1, #bimm64        //x0 = x1 | #bimm64
orr x0, x1, x2             //x0 = x1 | x2

eor x0, x1, #bimm64        //x0 = x1 ⊕ #bimm64
eor x0, x1, x2             //x0 = x1 ⊕ x2
```

ARM8 A64 Logical Instructions

```
asr x0, x1, #uimm        //x0 = arithmetic shift right x1 #uimm bits
lsl x0, x1, #uimm        //x0 = logical shift left x1 #uimm bits
lsr x0, x1, #uimm        //x0 = logical shift right x1 #uimm bits
ror x0, x1, #uimm        //x0 = rotate right x1 #uimm bits

asr x0, x1, x2       //x0 = arithmetic shift right x1 (x2 & 0x3f) bits
lsl x0, x1, x2       //x0 = logical shift left x1 (x2 & 0x3f) bits
lsr x0, x1, x2       //x0 = logical shift right x1 (x2 & 0x3f) bits
ror x0, x1, x2       //x0 = rotate right x1 (x2 & 0x3f) bits
```

ARM8 A64 Branch Instructions

Branch Instructions		
B	rel_{28}	$\text{PC} = \text{PC} + \text{rel}^{\pm}_{27:2}{:}0_2$
Bcc	rel_{21}	$\text{if(cc) PC} = \text{PC} + \text{rel}^{\pm}_{20:2}{:}0_2$
BL	rel_{28}	$\text{X30} = \text{PC} + 4;\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{27:2}{:}0_2$
BLR	Xn	$\text{X30} = \text{PC} + 4;\ \text{PC} = \text{Xn}$
BR	Xn	$\text{PC} = \text{Xn}$
CBNZ	rn, rel_{21}	$\text{if(rn} \neq 0)\ \text{PC} \mathrel{+}= \text{rel}^{\emptyset}_{21:2}{:}0_2$
CBZ	rn, rel_{21}	$\text{if(rn} = 0)\ \text{PC} \mathrel{+}= \text{rel}^{\emptyset}_{21:2}{:}0_2$
RET	{Xn}	$\text{PC} = \text{Xn}$
TBNZ	rn, #i, rel_{16}	$\text{if(rn}_i \neq 0)\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{15:2}{:}0_2$
TBZ	rn, #i, rel_{16}	$\text{if(rn}_i = 0)\ \text{PC} \mathrel{+}= \text{rel}^{\pm}_{15:2}{:}0_2$

ARM8 A64 Conditional Branch Instructions

```
b.cond label              //Jump to program relative label if cond is true

cbz x1, label             //Jump to program relative label if x1 is equal to zero

cbnz x1, label            //Jump to program relative label if x1 is not equal to zero

tbz x1, #uimm6, label     //Jump to program relative label if bit number
                          //#uimm6 in x1 is equal to zero
tbnz x1, #uimm6, label    //Jump to program relative label if bit number
                          //#uimm6 in x1 is not equal to zero
```

ARM8 A64 Unconditional Branch Instructions

```
b label          //Jump to program relative label
                 //Jump to implement branch and loop

br x1            //Jump to address in x1
                 //Jump to implement branch and loop

bl label         //Jump to program relative label and write next instruction
                 //address in link register (lr) or x30
                 //Jump to call a function

blr x1           //Jump to address in x1 and write next instruction
                 //address in link register (lr) or x30
                 //Jump to call a function and function address is in x1

ret lr           //Jump to address in lr or x30 and hints the CPU that this a
                 //subroutine return. lr can be omitted in ret instruction.
```