ARMv8 Fundamentals

Resources

https://en.wikipedia.org/wiki/ARM_architecture#64.2F32-bit_architecture

First see this: 'Introducing the Arm architecture', ARM Developer documentation

---

Source

Chapter 3. Fundamentals of ARMv8

In ARMv8, execution occurs at one of four *Exception Levels (ELn)*. In AArch64, the Exception Level determines the level of privilege, in a similar way to the privilege levels defined in ARMv7. The Exception Level determines the privilege level, so execution at EL*n* corresponds to privilege PL*n*. Similarly, an Exception level with a larger value of *n* than another one is at a higher Exception Level. An Exception Level with a smaller number than another is described as being at a lower Exception Level.

Exception Levels provide a logical separation of software execution privilege that applies across all operating states of the ARMv8 architecture. It is similar to, and supports the concept of, hierarchical protection domains common in computer science.

The following is a typical example of what software runs at each Exception Level:

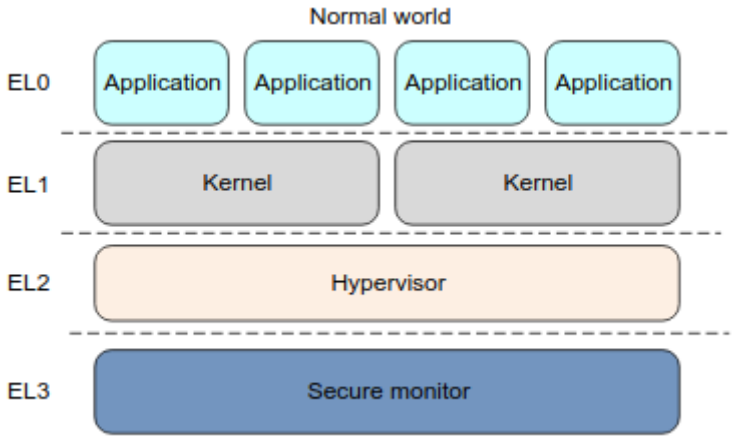
EL0
: Normal user applications.

EL1
: Operating system kernel typically described as *privileged*.

EL2
: Hypervisor.

EL3
: Low-level firmware, including the Secure Monitor.

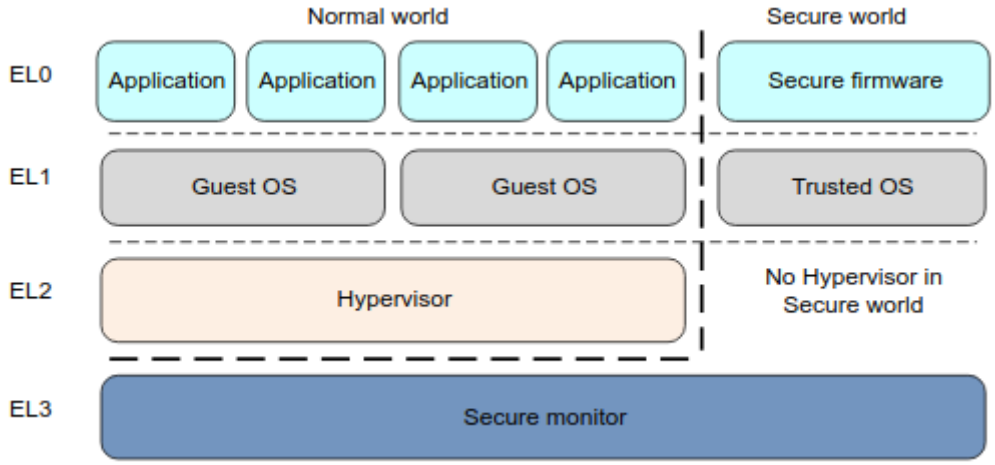
Figure 3.1. Exception levels



In general, a piece of software, such as an application, the kernel of an operating system, or a hypervisor, occupies a single Exception Level. An exception to this rule is in-kernel hypervisors such as KVM, which operate across *both* EL2 and EL1.

ARMv8-A provides two security states, Secure and Non-secure. The Non-secure state is also referred to as the *Normal World*. This enables an Operating System (OS) to run in parallel with a trusted OS on the same hardware, and provides protection against certain software attacks and hardware attacks. ARM TrustZone technology enables the system to be partitioned between the Normal and Secure worlds. As with the ARMv7-A architecture, the Secure monitor acts as a gateway for moving between the Normal and Secure worlds.

Figure 3.2. ARMv8 Exception levels in the Normal and Secure worlds



ARMv8-A also provides support for virtualization, though only in the Normal world. This means that hypervisor, or *Virtual Machine Manager* (VMM) code can run on the system and host multiple guest operating systems. Each of the guest operating systems is, essentially, running on a virtual machine. Each OS is then unaware that it is sharing time on the system with other guest operating systems.

The Normal world (which corresponds to the Non-secure state) has the following privileged components:

Guest OS kernels

> Such kernels include Linux or Windows running in Non-secure EL1. When running under a hypervisor, the rich OS kernels can be running as a guest or host depending on the hypervisor model.

Hypervisor

> This runs at EL2, which is always Non-secure. The hypervisor, when present and enabled, provides virtualization services to rich OS kernels.

The Secure world has the following privileged components:

Secure firmware

> On an application processor, this firmware must be the first thing that runs at boot time. It provides several services, including platform initialization, the installation of the trusted OS, and routing of Secure monitor calls.

Trusted OS

> Trusted OS provides Secure services to the Normal world and provides a runtime environment for executing Secure or trusted applications.

The Secure monitor in the ARMv8 architecture is at a higher Exception level and is more privileged than all other levels. This provides a logical model of software privilege.

Figure 3.2 shows that a Secure version of EL2 is not available.

<<
Source

What is TrustZone?
TrustZone is a set of security extensions added to ARMv6 processors and greater, such as ARM11, CortexA8, CortexA9 and CortexA15. To improve security, these ARM processors can run a secure operating system (secure OS) and a normal operating system (normal OS) at the same time from a single core. The instruction Secure Monitor Call or SMC bridges the secure and normal modes.

TrustZone technology is programmed into the hardware, enabling the protection of memory and peripherals. Since security is designed into the hardware, TrustZone avoids security vulnerabilities caused by proprietary, non-portable solutions outside the core. Security can be maintained as an inherent feature of the device, without degrading system performance, enabling device manufacturers to build security applications, such as DRM or mobile payment as protected applications that run on the secure kernel.

With TrustZone, user space applications operate in "normal" mode. The kernel runs "system" mode. The trusted kernel operates in "monitor" mode. Because of this architecture, even a "rooted" application cannot access protected regions within the trusted kernel. Any component can be designated as part of the trusted infrastructure,

from regions of the PCI-E address space to NAND memory. Overall, TrustZone offers a secure and easy-to-implement trusted computing solution for device manufacturers, without requiring additional hardware.

…

ARM has partnered with GlobalPlatform to define a new Trusted Execution Environment (TEE) API that covers all three aspects.

- •TEE Client API Specification: This specification is very similar to the ARM TrustZone API. Applications running in the normal OS can be ported from the ARM TrustZone API to the TEE Client API spec with simple wrappers.
- •TEE Internal API Specification: This defines the Secure OS implementation and enables easier porting of Secure Tasklets from one vendor to another. Think of it as another version of POSIX.
- •TEE System Architecture: Defines the overall software architecture.

…

Why a Trusted Execution Environment (TEE)?

…
The old two-level method of protection (kernel and user) is not enough as 'root' has full privilege to access everything in the system. TEE provides a third level of privilege where some key resources like device key can be protected from the normal or "rich" OS kernel. ARM TrustZone and Intel TXT all achieve this by providing a virtualized environment to run TEE and the normal OS in parallel.
…

Do Intel or AMD offer Trusted Execution Environments?
Yes, other processor architectures support TEE. Popular CPU Architectures and their TEE implementations:

1. ARM TrustZone
2. Intel TXT
3. AMD Secure Execution Environment

All three of these TEE implementations provide a virtualized Execution Environment for the secure OS and applications. To switch between the secure world and the normal world, Intel provides SMX Instructions, while ARM uses SMC. Programmatically, they all achieve very similar results.

…

>>

Aarch32 (A32) vs Aarch64 (A64)

Feature	***Aarch32/ARMv7***	***Aarch64***
Registers: number, naming	16 32-bit GPRs + CPSR; named `Rn` (`R0 to R15`);	31 64-bit GPRs ; named `Xn(X0 to X30)`
Register Width	Rn; 32 bit	Xn ; 64 bit; div into 2 parts: LSB 32-bit is the `Wn` part – directly equivalent to the `Rn`
Special purpose registers	- R13 - stack pointer, - R14 – link register, - - R15 – prg counter (PC)	- X29 – frame pointer - X30 – link register - no reg 31 but X31 sometimes represents the stack pointer
Access to system registers	Via Coprocessor – typically, CP-15	NO coprocessor; direct access to system registers via MSR/MRS machine instructions (Table 4-5 lists all System Registers)
Processor State	Saved in CPSR and (banked) SPSR (Saved Program Status Register)	No CPSR – an SPSR for each EL (exception level); SPSR_EL1, SPSR_EL2, SPSR_EL3. Just called PSTATE
Program Counter (PC) register	Programmable (weird)	Non-programmable, implicit
Zero register	~~None~~	XZR/WZR : read returns zero, no writes
Thumb instructions	T32 ISA supported	No Thumb instructions
Privilege levels	Upto 7 processor *modes*; LSB 5 bits of the CPSR	Modes replaced by *exception levels (ELn);* four Els, from lowest to highest: EL0, EL1, EL2, EL3; typically used for userspace (EL0), OS/supervisor (EL1 - possibly virtualized), hypervisor (EL2), trusted firmware or secure monitor (EL3). EL changed via interrupts or traps: `svc, hyp, smc` instructions – trap to supervisor, hypervisor or secure monitor EL
ABI	(See ARM-32 doc)	(see sec 9.1 “Register use in the AArch64 Procedure Call Standard”)

References

- *‘Learn The Architecture’ guides on ARM*
- *Manual: ARM Cortex-A Series : Programmer's Guide to the ARMv8-A (filename: DEN0024A_ARMv8_architecture_PG.pdf)*
- *Super Hexagon: A Journey from EL0 to S-EL3*

- *Transitioning from ARM v7 to ARM v8: All the basics you must know*
- *https://en.wikipedia.org/wiki/List_of_ARM_microarchitectures*

- *Arm64 hypervisor tutorial series*

ARMv8 Registers and their Usage (ABI)

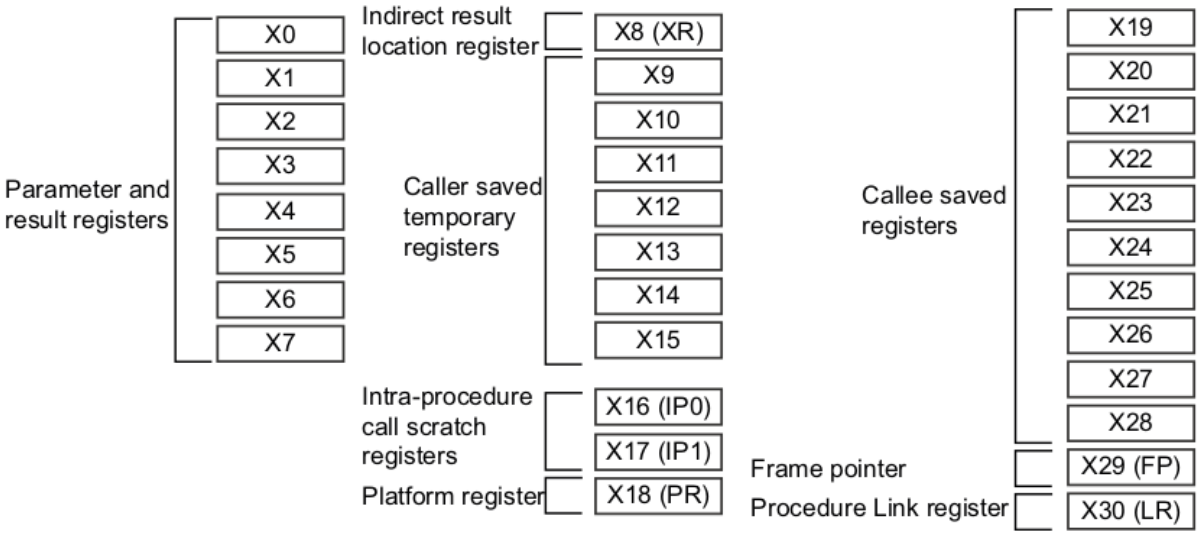


Figure 9-1 General-purpose register use in the ABI

(See manual page 9-3 *'Register use in the AArch64 Procedure Call Standard'* for details).

The 64-bit Xn registers shown above can be divided in two: into an LSB Rn 32-bit register (serving 32-bit apps) and a 32-bit MSB Wn part. Thus, the A64 (Aarch64) is binary compatible with the A32 (Aarch32); one can run a 32-bit userland (apps) on the 64-bit OS on an A64.

Below, from page 4-3:

In addition to the 31 core registers, there are also several special registers.

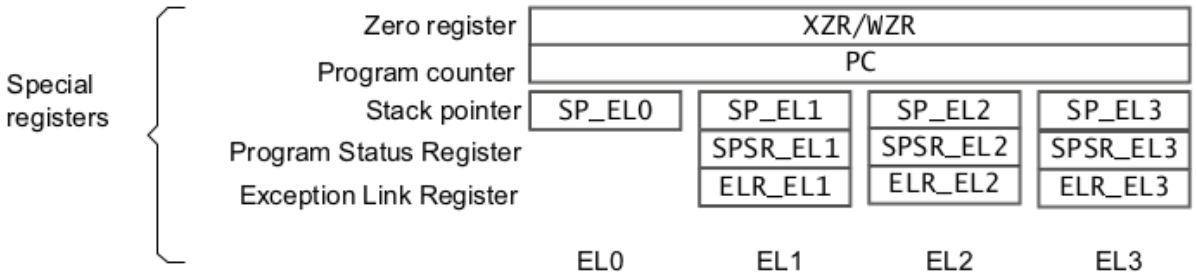


Figure 4-3 AArch64 special registers

NOTE- There is no register called X31 or W31. Many instructions are encoded such that the number 31 represents the zero register, ZR (WZR/XZR). There is also a restricted group of instructions where one or more of the arguments are encoded such that number 31 represents the Stack Pointer (SP).

Stack Pointer

In the ARMv8 architecture, the choice of stack pointer to use is separated to some extent from the Exception level. By default, taking an exception selects the stack pointer for the target Exception level, SP_ELn.

For example, taking an exception to EL1 selects SP_EL1. Each Exception level has its own stack pointer, SP_EL0, SP_EL1, SP_EL2, and SP_EL3.

When in AArch64 at an Exception level other than EL0, the processor can use either:
• A dedicated 64-bit stack pointer associated with that Exception level (SP_ELn).
• The stack pointer associated with EL0 (SP_EL0) *<< often done in Linux kernel ; see implementation of current on Aarch64! >>*
EL0 can only ever access SP_EL0.

Saved Program Status Register (SPSR)
Holds processor state; similar to the CPSR in ARMv7 (see pg 4-5).

Processor State is called PSTATE.
"In AArch64, you return from an exception by executing the ERET instruction, and this causes the SPSR_ELn to be copied into PSTATE . This restores the ALU flags, execution state, Exception level, and the processor branches. From here, you continue execution from the address in ELR_ELn."

System Registers

"In AArch64, system configuration is controlled through system registers, and accessed using MSR and MRS instructions. This contrasts with ARMv7-A, where such registers were typically accessed through coprocessor 15 (CP15) operations.

The name of a register tells you the lowest Exception level that it can be accessed from.
For example:
• TTBR0_EL1 is accessible from EL1, EL2, and EL3.
<< TTBR: Translation Table Base Register;
TTBR0_EL1 usually points to the base of the userspace process's paging tables
TTBR1_EL1 usually points to the base of the kernel's master paging table >>
• TTBR0_EL2 is accessible from EL2 and EL3.

<< FYI, VTTBR_ELn is the Virtualization Translation Table Base Register where n is 2 >>

Registers that have the suffix _ELn have a separate, banked copy in some or all of the levels, though usually not EL0. Few system registers are accessible from EL0, although the Cache Type Register (CTR_EL0) is an example of one that can be accessible.

Code to access system registers takes the following form:

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

Previous versions of the ARM architecture have used coprocessors for system configuration. However, AArch64 does not include support for coprocessors. *Table 4-5* lists only the system

registers mentioned in this book.”