arm

White Paper

# Arm Cortex-A32 – The Logical Choice for Rich Embedded

## When you need ultra-high efficiency and a rich embedded environment, Cortex-A32 is the logical choice

Chris Shore, Director Embedded Solutions

December 2018

# Introduction

Arm processors are widely used in the embedded space. Arm Cortex-A processors are typically used in applications requiring a rich operating system or high performance, Cortex-R processors for hard real-time performance and Cortex-M processors for small microcontroller-like applications.

Focusing on the Cortex-M processor cores, the current range is optimized for cost and power sensitive appplications. From the ultra–low power Cortex–M0+ processor through to the high performance Cortex–M7, the Cortex–M family of processors offers a wide range of performance points to suit variety of applications.

The Armv6-M, Armv7-M and Armv8-M architectures, implemented by Cortex-M processors, present a simple and logical programmer's model, designed for ease of use. The cores themselves are highly configurable, allowing a diverse range of implementations.

While the simplicity of Cortex-M cores is a powerful advantage across much of the embedded space, there is also a class of use cases which require a richer, more powerful environment. Such applications, while remaining highly sensitive to efficiency and power consumption, often necessitate a platform operating system, such as Linux or Android. Moving to such an operating system also opens up opportunities to leverage a much larger, richer and more sophisticated software ecosystem.

Cortex-M processors are not focused on these higher-level operating systems and hence do not include certain necessary features to run them. For instance, having no Memory Management Unit (MMU), they do not support virtual memory environments and therefore cannot support such operating systems. When an application calls for a richer operating environment, the first port of call is typically one of the "Ultra High Efficiency" Cortex-A cores. These cores provide the more advanced features needed by platform operating systems, while retaining a strong focus on power efficiency. All present a more advanced and flexible programmer's model.

As a result, Arm Cortex-A processors are deployed in a wide range of deeply embedded applications, particularly in markets where Linux or other rich operating systems are a requirement.

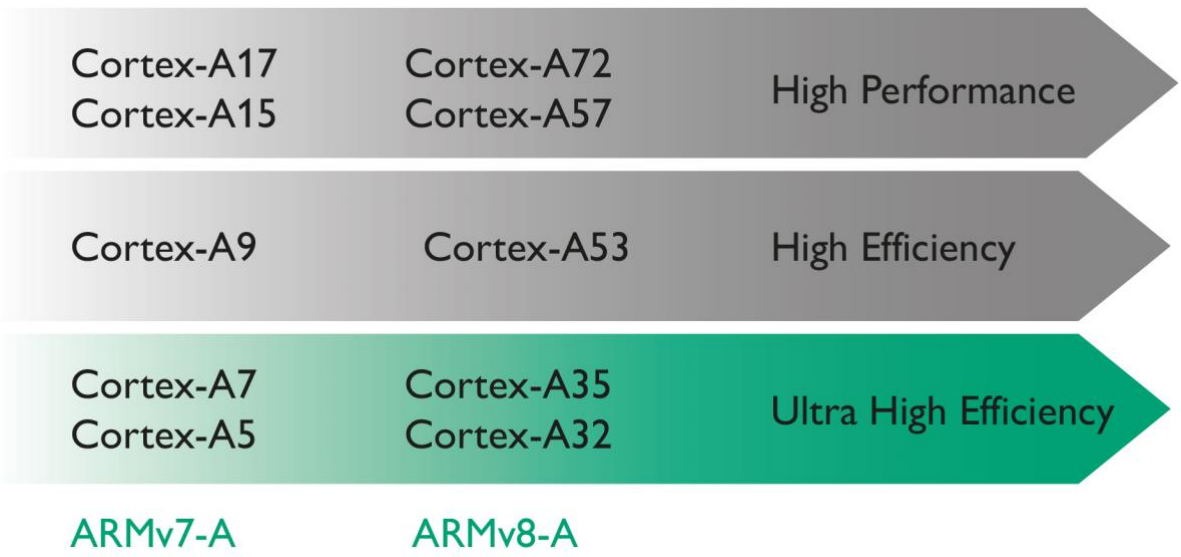


**Figure 1 - Cortex-A Processors and Architectures**

Figure 1 shows the current range of Cortex-A processors, highlighting the "Ultra High Efficiency" cores. This paper focusses on the latest addition to this category, the Cortex-A32.

The Cortex-A32 is an ideal stepping stone into the Cortex-A family for those applications which benefit from a richer operating system environment or from the performance and features benefits that Cortex-A processors can bring. It is the most power-efficient Armv8-A CPU to date and is an ideal choice for wearables, IoT and rich embedded applications, especially those which require the use of a platform operating system such as Linux.

# Introduction to Cortex-A32

Cortex-A32 occupies a unique place in the Arm architecture. It is built to the Armv8-A architecture, but is implemented as a 32-bit only processor. Figure 2 shows how Cortex-A32 fits into the Armv8-A architecture profile and how it compares with Cortex-A35.

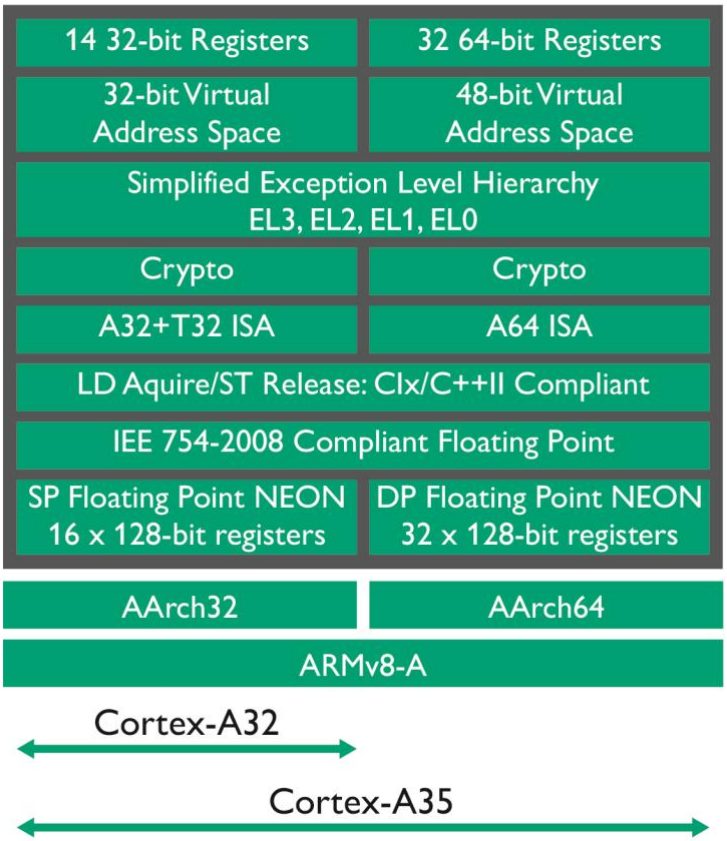


**Figure 2 - Cortex-A32 and Armv8-A**

You can see that Cortex-A35 implements both the 32-bit AArch32 and 64-bit AArch64 execution states to give full access to the 64-bit capability of the Armv8-A architecture. Cortex-A32, on the other hand, implements only the 32-bit AArch32 execution state. The decision to omit the 64-bit capability brings savings in area and even greater power efficiency for use cases which do not require 64-bit capability. While a number of applications within the embedded space benefit from 64-bit operation, many others are focused on 32-bit and are likely to remain so for the foreseeable future. It is at these applications that Cortex-A32 is targeted.

The AArch32 execution state is an evolution of the Armv7-A architecture used by earlier Cortex-A processors. While not offering 64-bit functionality, it offers some significant enhancements, making Cortex-A32 even more efficient when compared to both the Cortex-A7 and the Cortex-A5, and an ideal choice for evolving designs based upon these earlier Arm processors, or for new designs targeting this market segment.

Those AArch32 enhancements over Armv7-A include:

- A number of new instructions, providing increased performance for cryptographic functions
- New Load Acquire and Store Release instructions which provide more efficient memory ordering capability, matching the new C++11 memory ordering semantics
- Additional scalar and SIMD floating point instructions
- A wider range of system control instructions

These extra features provide greater performance compared to earlier 32-bit Armv7-A processors.

The inclusion of the Advanced Coherency Extensions (ACE) on the Cortex-A32 bus interface makes it possible to build fully coherent multiprocessing systems using Cortex-A32, giving an upgrade path to higher performance as required. However, if area or power consumption are the driving constraints, a variant of Cortex-A32 is available which is specifically optimized for uniprocessor applications, omitting coherency logic to provide greater power efficiency.

The addressable physical memory space is expanded for Cortex-A32, via the Large Physical Address Extension (LPAE), beyond the 32-bit (4GB) space offered by Cortex-A5 to a full 40-bit address space.

The core itself incorporates several additional advanced features which improve efficiency. These include more flexible power management, finer-grained power domains, and extensive use of retention power gating.

# Architectural comparison

## Armv7-M Features

An Arm Cortex-M processor is built to the Armv7-M architecture profile (or, in the case of the Cortex-M0 and Cortex-M0+, to the similar Armv6-M architecture). While this architecture shares many features with earlier Arm architectures, it was specifically designed to support deeply-embedded, low-cost, real-time microcontroller applications. Many features of earlier architectures were therefore removed and new ones added to produce a more "microcontroller-like" programmer's model.

Specifically, the changes from legacy processors, such as the popular Arm7TDMI, can be summarized as follows:

- The number of operating modes was significant reduced, from 7 or more to just 2: Handler mode and Thread mode. One of the modes (Handler mode) is optionally privileged.
- The register file was simplified. While the number of registers available to the programmer remains essentially the same at 16, the banked register mechanism used in earlier architectures was significantly reduced so that only the Stack Pointer (r13) is banked between the two operating modes. Use of the banked copy is optional and it may even be omitted.
- The most significant changes were made in the exception model. Since typical microcontroller applications may have a large number of on-chip peripheral interrupts, a standard Nested Vectored Interrupt Controller (NVIC) specification is included in the architecture and all Cortex-M cores include this. Similarly, the exception handling model was standardized on a vector table consisting of handler addresses. The context save and restore operations were implemented completely in hardware to simplify the software task of writing interrupt handlers. This allows implementations with very low and predictable interrupt latency.
- Armv7-M defines an optional memory protection architecture similar to that used on some earlier Arm processors. Virtual memory is not supported since it is generally not required for bare-metal systems or those running under an RTOS.

- To facilitate the implementation and porting of a wide range of Real Time Operating Systems (RTOS), some standard on-chip peripherals were also defined in the architecture e.g. a SysTick timer.
- To reduce the size of the processor cores, Armv7-M processors are restricted to the Thumb instruction set only (including the Thumb-2 extensions), with the smallest cores implementing just a minimal subset of it.

### Armv8-A AArch32 Features

Cortex-A processors are built to the Armv7-A or Armv8-A architecture profiles. Armv8-A processors provide the AArch32 execution state, which is a backwards-compatible evolution of the 32-bit Armv7-A architecture. These architectures enable features which are designed to support platform operating systems, such as Linux, Android, Windows etc., which require, among other things, a virtual memory environment.

Specific features which you will come across as being significantly different from the Cortex-M processor cores are:

- There are 7 or more operating modes: User, Supervisor, IRQ, FIQ, Undefined, Abort, System. Each is intended for handling a specific kind of event (e.g. IRQ mode is intended for handling IRQ interrupts). AArch32 also supports two additional modes, Hyp and Monitor, which are used for virtualization and Arm TrustZone security, respectively.
- While the number of available registers is the same (16), AArch32 has a number of "banked" registers which are associated with the operating modes listed above. These registers replace their User mode counterparts when the relevant operating mode is entered. This simplifies many aspects of exception handling but does mean that management of the machine and the effort required to initialize it is increased.
- The exception model is significantly different and has its origins in the earliest Arm architecture devices. Specifically, the vector table consists of a set of executable instructions rather than addresses, and the task of saving and restoring context is left almost entirely to the programmer.
- A major difference is the inclusion of a Memory Management Unit (MMU) which carries out translation between virtual addresses, as issued by the core, and physical addresses, as required by the memory system. This supports the implementation of a complete demand-paged virtual memory environment as used by platform operating systems, such as Linux.

## Differences between Armv7-M and AArch32

When moving from a system built around a Cortex-M processor to one based on the Cortex-A32 processor, there are many new features to be aware of.

While there are many similarities between the two architectures (the register bank and instruction set, for example, exhibit many commonalities) it is important to remember that many features which are included in the AArch32 execution state of Armv8-A architecture are based on features of earlier architectures.

This section describes the features present in AArch32 which are not found in Armv7-M, or which are found in a significantly different form.

### Operating modes

As shown in Figure 3, Armv7-M defines only two operating modes, Thread mode and Handler mode. Handler mode may optionally be privileged, though this feature does not have to be used in software if it is not required. Handler mode is used for handling exceptions and Thread mode is used for user processes. Transitions between the modes are essentially automatic, happening on certain events as shown in the figure. Handler mode, for instance, is automatically entered when an exception is taken and the return from Handler mode is automatic on completion of the exception.

The SVCall instruction is the primary method by which software can cause entry to Handler mode (it is also possible to set the pending status of an enabled IRQ to cause the handler to execute).

Compare this with Figure 4, which shows the operating modes supported in AArch32 execution state. There are seven basic modes, with five designated for handling specific exceptions. Fast Interrupt (FIQ) mode, for instance, is entered when an FIQ exception is taken; Undef mode is entered when an undefined instruction is encountered, and so on.

Transitions between the modes are usually automatic but it is possible to switch between the modes entirely under software control by writing the Mode field in the Current Program Status Register (CPSR). This is described in greater detail below. Similar to the SVCall instruction, the SVC instruction is provided for software to cause an SVC exception and entry into SVC mode.

There are two additional modes supported in AArch32 which are not shown in the figure (simply to save space). These are Hyp mode (for hypervisors) and Monitor mode (for TrustZone security). These are complex subjects and we do not deal with them in this paper.

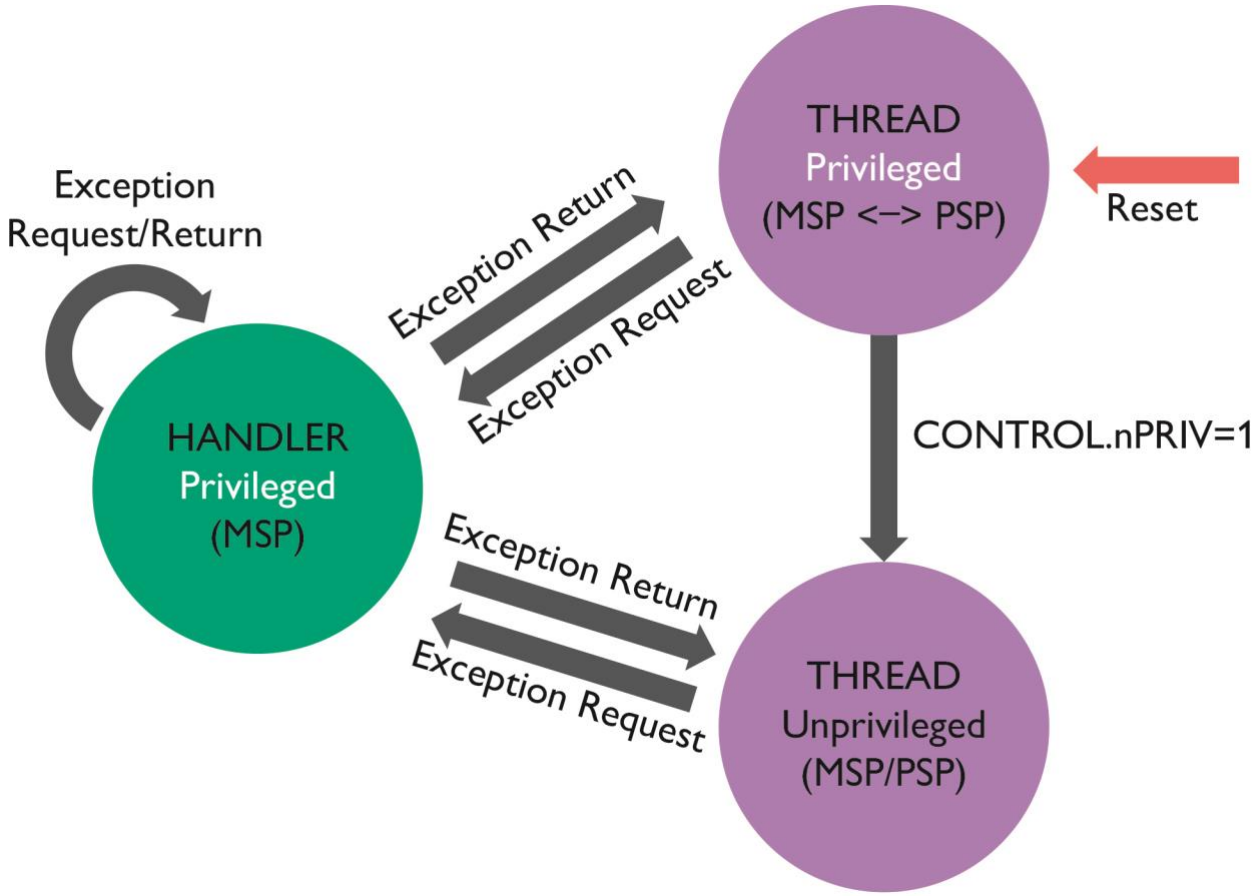


**Figure 3 - Armv7-M Operating Modes**

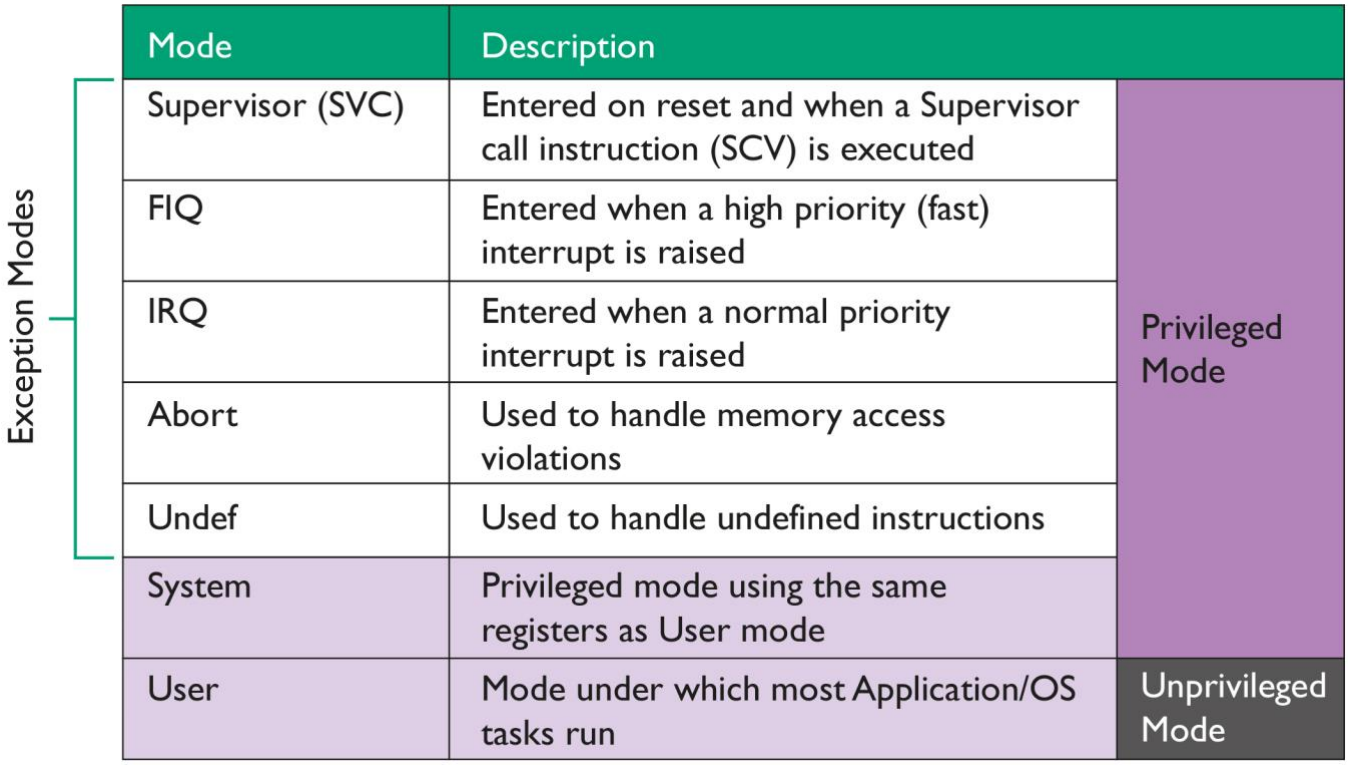


**Figure 4 - AArch32 Operating Modes**

## Register bank

Figure 5 and Figure 6 show the Armv7-M and AArch32 register banks respectively.

You can see that many of the registers are common – this arises from the fact that the two have a common heritage in the Armv6 and earlier architectures.

Most instructions have access to 13 general purpose registers, r0-r12. In both architectures, r13 is reserved as the Stack Pointer (SP), r14 as the Link Register (LR) and r15 as the Program Counter (PC). In Armv7-M access to these special registers is limited to a number of restricted use cases which reflect the function of those registers; in AArch32, these registers can be accessed in much the same way as any other general purpose register (though, clearly, changing the value of the PC may have undesirable side-effects!).

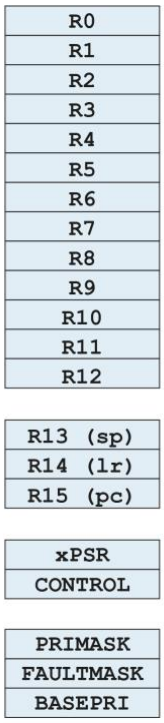


Figure 5 - Armv7-M Register Set

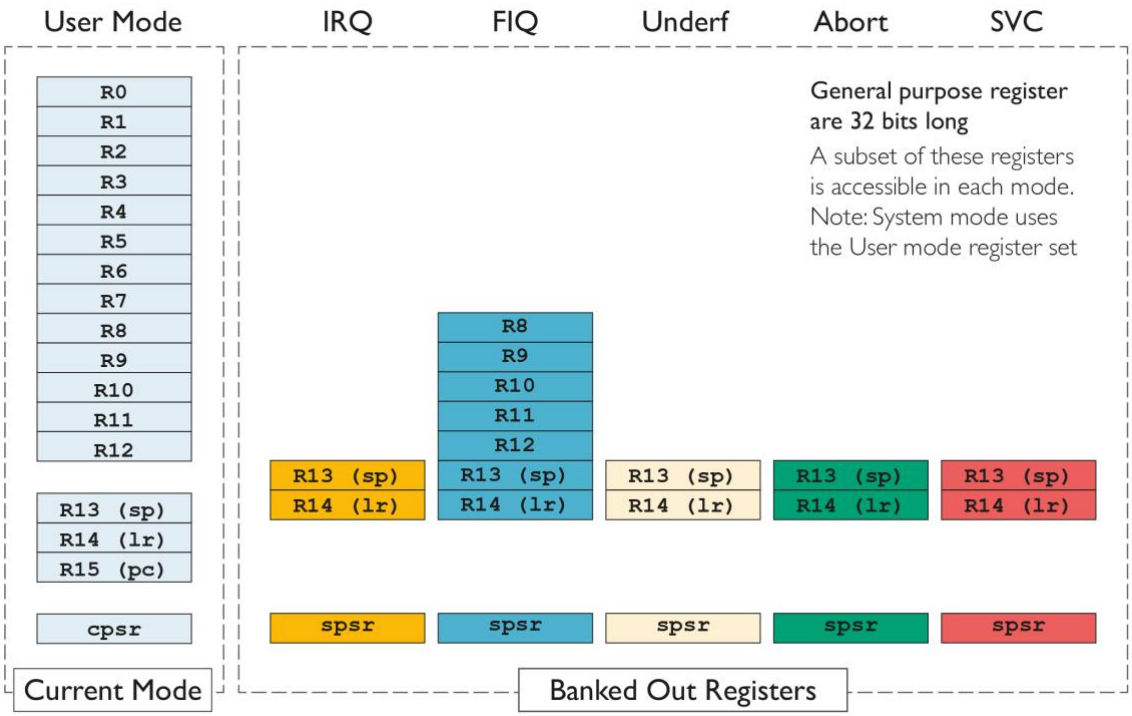


Figure 6 - AArch32 Register Set

Armv7-M specifies a small set of additional special purpose registers, PRIMASK, FAULTMASK, xPSR, CONTROL and BASEPRI, which are used for controlling and configuring the processor, and for managing exception handling.

## Instruction set

In Figure 6, you can see that AArch32 also provides a number of registers which are associated with particular operating modes. These registers exchange places with their User mode counterparts when the relevant mode is entered. They are otherwise inaccessible to all but a very few special instructions and cannot be accessed directly. Their values are preserved across mode changes, which aids with exception handling. In particular, each exception mode has its own private SP, which allows each exception to be handled on a separate stack. This makes for more robust and defensive exception programming. The LR in the relevant mode is set to the exception return address when an exception is taken.

Also shown against each exception mode is an additional register called the Saved Program Status Register (SPSR). The SPSR is used for taking a snapshot of the current CPSR value on entry to an exception thus, together with LR, providing an automatic context save.

Again, Mon and Hyp modes are not shown in the AArch32 figure. They each support a banked R13 and R14, just like the other modes.

In Cortex-A, there is a separate register bank associated with the Arm NEON SIMD instruction set (described below). This consists of 32 registers of 128-bit width. Each is addressable as a word, doubleword or quadword and the NEON instruction set supports vectored operations on everything from bytes to quadwords.

## Exception model

The exception models of the two architectures are significantly different. Both support both internal and external exceptions, caused either by system events or external peripheral interrupts.

Armv7-M supports a model which is much closer to that found on traditional microcontrollers, with all external interrupts separately vectored through a vector table consisting of handler addresses.

AArch32 supports a model much closer to that found in earlier Arm architectures in which there are only 8 exception types with distinct vectors. The vector table consists of executable instructions, which are usually branch instructions directed to the appropriate exception handler. Only two external interrupt sources are supported: FIQ and IRQ. Typically, a single high-priority interrupt is connected to FIQ, with the rest being connected to IRQ. This means that systems must either incorporate a software despatcher or, as is common on modern systems, include a Vectored Interrupt Controller (VIC) which can be programmed with individual vector addresses.

Many Cortex-A systems include a standard interrupt controller based on Arm's Generic Interrupt Controller (GIC) architecture. The GIC acts as the interface between many physical interrupts and the Arm core's two interrupt inputs (FIQ and IRQ). It handles prioritization, masking, individual interrupt enable/disable and pre-emption. See the GIC Architecture Reference Manual for further information.

## Instruction Set

The Arm instruction set has evolved since its introduction in the Arm1 over 25 years ago. A Cortex-A processor actually supports two instruction sets, each with a number of extensions.

- **Arm Instruction Set**
  The Arm instruction set is based on the original instruction set supported by the first Arm processor. It has been extended several times. It is a load-store instruction set with distinct groups of instructions for data processing, memory access, system control, and control flow. The modern Arm instruction set is very powerful and extensive. In this instruction set, all instructions are encoded as fixed-length 32-bit words and must be aligned on word boundaries.
- **Thumb Instruction Set**
  The Thumb instruction set is a subset of the Arm instruction set in which each instruction is encoded as a 16-bit halfword. They must be aligned on halfword boundaries. The original rationale for the Thumb instruction set was to improve code density by reducing the size of the most commonly used instructions when compiling from a high-level language, such as C. Due the smaller instruction size, it can also be beneficial when running from instruction caches as more instructions fit in a given cache line.
- **Advanced SIMD Extensions**
  The Advanced SIMD Extensions, also referred to as NEON, are a large set of instructions which provide SIMD vector processing capability using an extension register set.
- **Vector Floating Point (VFP)**
  The VFP instruction set operates on the same register bank as NEON. It provides a powerful set of IEEE-754 compliant single and double precision floating point operations.
- **Thumb-2 Technology**
  Thumb-2 is the name for a set of extensions which were introduced to the Thumb instruction set in Armv6T2 (first, with the Arm1156T2-S processor). This results in a mixed-length instruction set combining the code density of Thumb with the high performance and flexibility of the Arm instruction set.

If you have been developing using a Cortex-M microcontroller, you will be most familiar with Thumb-2. These cores support only Thumb-2, in various subsets ranging from the smallest (found in Cortex-M0 and Cortex-M0+) to the

largest (found in Cortex-M7). You will find that moving to a Cortex-A processor opens up many more possibilities for code generation.

In general, the majority of high-level code which is compiled for Cortex-A processors will target Thumb (with Thumb-2). This allows the compiler maximum freedom to make sensible choices about which instruction to use when there are multiple choices and achieve the maximum possible differentiation between compiling for code space and compiling for performance.

The Arm instruction set is generally used for code sections where maximum performance is critically important. Sometimes, these sections may be hand coded in assembler and typically the Arm instruction set is the right choice for that.

The NEON instruction set can be accessed in a number of ways:

- There are libraries available supporting common mathematical and analytical functions and algorithms.
- The compiler supports a comprehensive set of intrinsic functions which allow access to almost the complete NEON instruction set directly from C. Using this method, NEON operations can be interleaved with C statements in the most portable manner.
- NEON can be directly implemented by hand in assembler.
- The compiler also supports automatic vectorization of iteration loops. Provided that the code is written to some simple guidelines, the compiler can very effectively unroll and vectorize even fairly complex loops.

If you are familiar with Armv7-A processors, you will also notice some additional instructions which have been introduced as part of Armv8-A.

- **Cryptographic Extensions**
  These instructions are new in Armv8-A and are aimed at efficient implementation of common building-block algorithms for cryptographic functions. They operate on the NEON register bank.
- **Load-Acquire and Store-Release**
  These new instructions match the C++11 memory ordering semantics and make compiling these very efficient. They can also be used to reduce the need for data side memory barriers and partially eliminate the overhead associated with them.

There are some other minor extensions to the floating point and barrier instructions.

## Virtual memory support

The ability to support a full virtual memory environment is one of the major features of Armv8-A. It is this, more than anything else, which enables these devices to support platform operating systems such as Linux and Android. As such, virtual memory capability is often the critical criterion when selecting one of these cores.

A virtual memory environment allows an operating system to manage memory in a much more flexible way, allowing individual processes to, for instance, dynamically extend stack regions, enabling individual code and data regions to be paged in and out of external storage as required, and giving each user process an identical view of the system memory map.

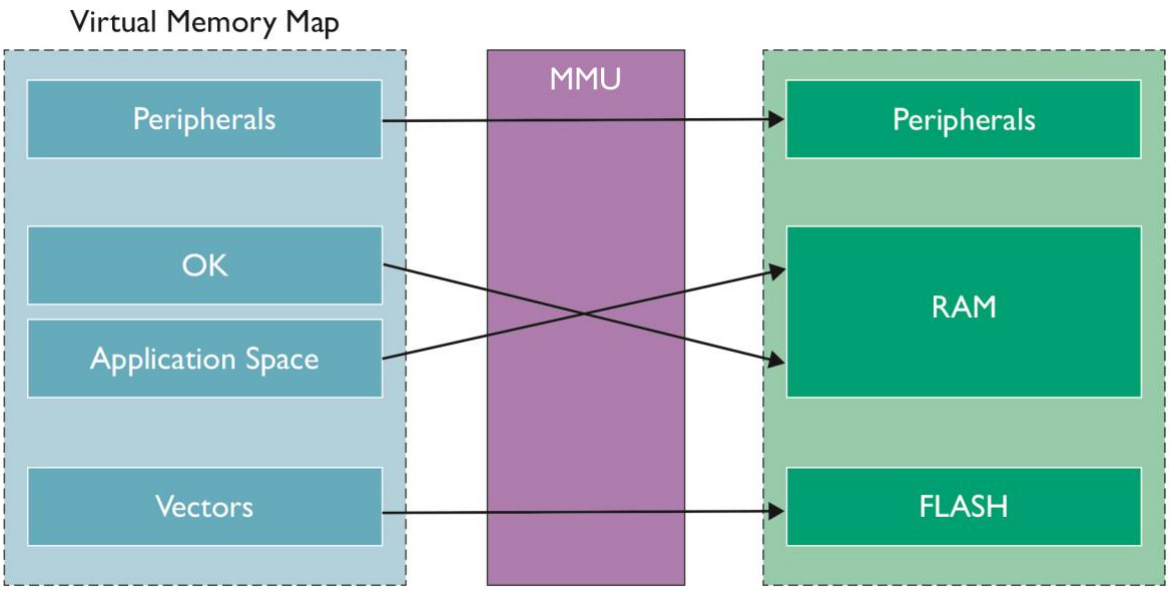


**Figure 7 - Virtual memory**

To achieve this, virtual memory introduces a "translation" on every address issued by the processor, as shown in Figure 7. The software executes in the "virtual address space" and a block, called a Memory Management Unit (MMU), translates this to the "physical address space". As well as giving the operating system complete control over such things as access permissions, this allows it to create a new virtual memory map for each user task in the system and for the operating system itself. Each task can execute in its own virtual memory space, as if it were the only task in the system. Only the operating system is aware of the real, physical location of the task's code and data regions in external, physical memory.

On a task switch, one of the jobs of the operating system is to reconfigure the MMU to enable the code and data used by the incoming task, while at the same time making the outgoing task's memory temporarily inaccessible. This enforces separation between tasks and is a crucial element of secure and resilient systems.

Without going into all the detail here, the MMU in Arm processors uses data contained in "page tables", which are held in external memory, to drive and control the translations. There are several optimizations incorporated into the system (e.g. the inclusion of Translation lookaside Buffers, or TLBs, which cache recently used translations to reduce the overhead of reading the page tables) which make the overhead of the translation process as small as possible.

# Software migration from Armv7-M to Armv7-A

Most high-level software will need simply to be recompiled. Software in the following areas will need closer attention:

- **Reset code and other exception handlers**
  If you are using an operating system, much of this will be taken care of by facilities provided by the operating system. In most cases, ports of common operating systems will be available either through the public domain distributions or from the supplier of the device.
  Due to the significant differences in the exception model, interrupt handlers will need to be rewritten. Again, the operating system will provide an infrastructure within which to do this, so the body of most interrupt handlers can simply be recompiled.

- **Peripheral drivers**
  When moving from an RTOS to a rich platform operating system, such as Linux, application code and peripheral drivers will need much clearer separation.
- **System configuration functions**
  There are significant differences in the way in which Cortex-M and Cortex-A based devices provide access to system configuration and control functions. Cortex-M processors are generally configured via named or memory-mapped registers which can be directly read and written to achieve the required function. Cortex-A processors (in AArch32 execution state as supported by the Cortex-A32) support this via a "system-control coprocessor". A notional "coprocessor 15" contains a large set of configuration registers which are read or written using dedicated instructions (look up MRC and MCR in the documentation). System configuration functions which are not carried out by the operating system will need to be rewritten to take account of this. That said, the operating system will usually provide an API for functions which need to be accessed by user software.
- **Assembly code**
  Clearly, assembly code will need some careful attention. Since one main reason for writing in assembly code is to maximize performance, these functions should be closely examined to see if the can benefit from being rewritten to access some of the extended instruction set capability, such as NEON. If legacy assembly code has been written using "Uniform Assembler Language" syntax (UAL), then the majority will simply reassemble into either Arm or Thumb instructions.

# More resources

There are many documents and books available about the Armv7-A and Armv8-A architectures. Details of all of them can be found on Arm's website at http://infocenter.arm.com.

The following, in particular, are recommended for those who are new to these architectures:

- Cortex-A Programmer Guide, Armv8-A edition
- Cortex-A Programmer Guide, Armv7-AR edition

The following are considerably more detailed and are the definitive reference material for the architectures:

- Armv8-A Architecture Reference Manual
- Generic Interrupt Controller Architecture Reference Manual
- Armv7-AR Architecture Reference Manual

For further detail on the Armv7-M architectures:

- Armv7-M Architecture Reference Manual
- Definitive Guide to Arm Cortex-M3 and Cortex-M4 Processors

Arm also provides a comprehensive range of training courses, covering software development on all current processors and architectures. Refer to http://www.arm.com/training for more details of the courses on offer.

# Trademarks