Application Note 314

Migrating from SH-4A to ARM Cortex-A

Document number: ARM DAI 0314
Issued: March 2012

ARM

Application Note 314
Migrating from SH-4A to ARM Cortex-A

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
March 2012	A	First release

Proprietary notice

Confidentiality status

This document is Open Access. This document has no restriction on distribution.

Feedback on this Application Note

If you have any comments on this Application Note, please send email to errata@arm.com giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- an explanation of your comments.

General suggestions for additions and improvements are also welcome.

ARM web address

http://www.arm.com

Table of Contents

1 Introduction 4
1.1 The ARM architecture 4
1.2 ARM development tools 4
1.3 Scope 4
1.4 References and Resources 4

2 An overview of the ARM architecture 6
2.1 ARM architecture versions 6
2.2 Architecture ARMv7-A extensions 7
2.3 Programmer's model 7
2.4 Debug 10

3 SH-4 and ARM compared 11
3.1 Programmer's model 11
3.2 System control and configuration registers 22
3.3 Exceptions and interrupts 23
3.4 Memory 23
3.5 Self-modifying code 28
3.6 Debug 28
3.7 Power management 29
3.8 Multi-threading and multi-processing 29
3.9 Multimedia extensions 29

4 Migrating a software application 31
4.1 General considerations 31
4.2 Tools configuration 33
4.3 Operating system 33
4.4 Startup 33
4.5 Handling interrupts and exceptions 33
4.6 Timing of NOP instructions 33
4.7 Power Management 33
4.8 Hardware discovery 34
4.9 Accessing peripherals 34
4.10 C programming 34
4.11 Assembly language programming 35
4.12 Function pointers 35
4.13 Semaphores etc. 35

5 A porting checklist 37

1 Introduction

The purpose of this document is to highlight areas of interest for those involved in migrating software applications from SH-4A to ARM platforms. No attempt is made to promote either architecture over the other, merely to explain clearly the issues involved in a decision to migrate an existing software application from one to the other.

Familiarity with the SuperH Architecture is assumed and corresponding and additional ARM features are explained.

The ARM architecture is supported by a wide range of technology, tools and infrastructure available from a large number of partners in the ARM Connected Community. Pointers to these resources are given where appropriate, although ARM's own supporting technology is highlighted.

There is much related documentation available from ARM (see references below) which should be consulted where further detail is required.

1.1 The ARM architecture

The ARM architecture represents the most popular 32-bit embedded processor range in current use. In many ways, the architecture as it exists today reflects its original design goals of being simple, cheap to implement and to use minimal power. It embodies many attributes traditionally associated with RISC architectures but also embraces more complex instruction types and addressing modes which are not part of the RISC concept.

The current versions of the architecture are described in more detail below.

1.2 ARM development tools

Tools for developing software for ARM platforms are available from a wide selection of vendors. ARM itself produces the RealView and DS-5 tools for high-performance application development. The Keil Microcontroller Develop Kit (MDK) is a lower-cost solution for development with microcontroller products.

Many other toolsets are available from other vendors, including a free toolchain from GNU.

1.3 Scope

It is important to note that this document addresses the needs of those tasked with migrating software applications to an A

RM platform. We assume that the application is running under some kind of platform Operating System e.g. Linux, Windows, Android or similar. This Operating System will, to a large extent, shield the application programmer from many of the details of the underlying platform and, to some extent, from the architecture of the processor itself.

We do not, therefore, deal in detail with issues like virtual memory management, exception handling, operating mode etc except where they are of interest or when they have a direct effect on the application environment.

The Operating System developer will require a much greater in-depth knowledge of the platform and the processor architecture which is beyond the scope of this document.

1.4 References and Resources

(All ARM documentation referenced here may be downloaded directly from infocenter.arm.com. Some may require you to register for an account before downloading the document.)

ARM Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406

Cortex-A15 Technical Reference Manual, ARM DDI 0438

Cortex-A9 Technical Reference Manual, ARM DDI 0388

Cortex-A8 Technical Reference Manual, ARM DDI 0344

Cortex-A5 Technical Reference Manual, ARM DDI 0433

Cortex-A Series Programmer's Guide, ARM DEN 0013

ARM Compiler Toolchain Developing Software for ARM Processors, ARM DUI 0471

ARM Compiler Toolchain Using the Assembler, ARM DUI 0473

ARM Compiler Toolchain Assembler Reference, ARM DUI 0489

ABI - Procedure Call Standard for the ARM architecture, ARM IHI 0042

ARM Generic Interrupt Controller Architecture Specification, ARM IHI 0048

Application Note 212 – Building Linux Applications using RVCT4.0 and the GNU Tools and Libraries, ARM DAI 0212

Barrier Litmus Tests and Cookbook, ARM GENC 007826

2 An overview of the ARM architecture

ARM is a 32-bit architecture. As such, it has 32-bit registers and a 32-bit ALU. Additionally, in the classic ARM instruction set, all instructions are 32 bits wide. Individual ARM implementations may have wider internal and external buses for increased performance and throughput.

In contrast with SuperH, ARM has been a full 32-bit architecture since its origination in 1985. As such, there is no need to provide any backwards compatibility with earlier versions differing in key aspects such as word size, instruction size etc.

> The ARM architecture has a fixed word size of 32 bits. The fundamental memory access size is a single 32-bit word (although byte, halfword and doubleword transactions are supported, memory is addressed as a linear array of words). Because of its heritage in earlier 8-bit and 16-bit architectures, the terminology used by the SH-4 architecture is different. There, a “word” refers to a 16-bit quantity, “longword” to a 32-bit quantity and there is no term for a 64-bit quantity (referred to as a “doubleword” by the ARM architecture)
>
> This document uses the ARM terminology exclusively, in which a word refers to a data size of 32-bits.

Data size (bits)	ARM	SH-4A
8	byte	byte
16	halfword	word
32	word	longword
64	doubleword	N/A

2.1 ARM architecture versions

The ARM architecture has been through several revisions since its emergence in the mid 1980’s. The most recent version, ARMv7, is implemented in the Cortex range of processors. The architecture is defined in three “profiles”, the ‘A’ profile or Application-class processors, ‘R’ for Real-time and ‘M’ for microcontroller devices.

ARMv7-A is currently implemented in the Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9 and Cortex-A15 processors and supports fully-featured application class devices capable of running platform Operating Systems such as Linux, Windows Mobile etc. It provides full virtual memory support and optional media processing, security and virtualization extensions.

ARMv7-R is available in the Cortex-R4 and Cortex-R5 and is targeted at applications which require hard, predictable real-time performance. Devices incorporating a Cortex-R4 processor are used, for instance, in engine management systems, hard disk drive controllers and mobile baseband processors.

ARMv7-M is used in microcontroller-type devices, principally those based around the Cortex-M3 and Cortex-M4 processors. This profile supports a subset of features in the v7-A and v7-R profiles aimed at enabling devices which maximize power efficiency and minimize cost. The architecture incorporates many features common in the microcontroller world e.g. bit-banding, hardware interrupt pre-emption etc.

In this document, we assume that the target ARM platform is built around an ARMv7-A processor. Unless explicitly stated otherwise, we refer to the ARMv7-A architecture including the security, advanced SIMD, floating point, Java acceleration and multiprocessing extensions as described in section 2.2 below.

In addition, we consider implementations of the ARMv7-A architecture which include the 40-bit physical addressing (LPAE) and virtualization extensions described in sections 2.2.5 and 2.2.6 below. These extensions are supported by the ARM Cortex-A7 and Cortex-A15 processors.

2.2 Architecture ARMv7-A extensions

There are several optional extensions to architecture ARMv7-A. For further details of these extensions and their intended use, refer to the architecture documentation.

2.2.1 Security

The TrustZone security extensions were introduced in architecture v6K and are an optional extension to the ARMv7-A profile. They introduce an additional operating mode (Monitor mode) with associated banked registers and an additional “secure” operating state.

2.2.2 Advanced SIMD and Floating Point

Both floating point (VFP) support and SIMD (NEON) are optional extensions to the ARMv7-A profile. They may be implemented together, in which case they share a common register bank and some common instructions. Almost all NEON implementations also include floating point support.

2.2.3 Java acceleration

Two architectural extensions are available for accelerating Java and other dynamically compiled languages. Both Jazelle DBX (acceleration for Java only by implementing hardware support for execution of bytecodes) and Jazelle RCT (an extension to the Thumb instruction set providing acceleration for a wider set of dynamically compiled languages) are a required part of the ARMv7-A architecture (though “trivial” implementations are possible).

Note that these two extensions are not often used in ARMv7-A devices and Jazelle RCT is now deprecated. The Cortex-A15 and Cortex-A7 processors provide a trivial implementation – see the documentation for further details.

2.2.4 Multiprocessing

These provide for synchronization and coherency across a “cluster” of cores, operating either in Asymmetric or Symmetric Multi-Processing mode. This extension is currently supported by the Cortex-A5MP, Cortex-A9MP, Cortex-A7 and Cortex-A15 processors.

2.2.5 40-bit physical addressing

The Large Physical Address Extensions (LPAE) are an optional extension to the ARMv7-A profile. This extension to the VMSAv7 virtual memory architecture allows the generation of 40-bit physical addresses from 32-bit virtual addresses.

LPAE is supported by the Cortex-A7 and Cortex-A15 processors.

2.2.6 Virtualization

The virtualization extensions introduce an extra mode (Hypervisor mode) with associated banked registers. A new Hyp exception can be used to trap software accesses to hardware and configuration registers, thus allowing implementation of an efficient hardware-assisted virtualization solution.

These extensions are supported by the Cortex-A7 and Cortex-A15 processors.

2.3 Programmer’s model

The description presented here is standard for the ARMv7-A and ARMv7-R architecture profiles. The ARMv7-M microcontroller profile has a significantly different model for modes and exceptions.

2.3.1 Standard features

1. Operating modes

The ARM processor supports up to nine operating modes. All of these, with the exception of User mode, are privileged. Seven modes (Supervisor, Undefined, Abort, FIQ, IRQ, Hyp and Monitor) are associated with handling particular types of exception events. Applications generally run either in User mode (unprivileged) with the operating system running in Supervisor mode.

Hyp mode is only present in processors supporting the Virtualization extensions (this includes the Cortex-A15); Monitor mode is only in processors supporting the Security extensions (currently all ARMv7-A processors).

2. Register set

The ARM register set consists of a maximum 43 general-purpose registers, 16 of which are usable at any one time. The subset which is usable is determined by the current operating mode – see diagram below.

In addition to the general purpose registers, the CPSR (Current Program Status Register) holds current status, operating mode, instruction set state, ALU status flags etc.

Seven of the modes also provide an SPSR (Saved Program Status Register) which is used for taking a copy of processor state on entry to an exception handler.

The diagram shows the standard ARMv7-A register set. Where registers are not shown under a particular mode, the corresponding User mode register is used.

	Exception Modes						
User/System	**FIQ**	**IRQ**	**Abort**	**Undef**	**SVC**	**Monitor**	**Hyp**
R0	Shared with User Mode	Shared with User Mode	Shared with User Mode	Shared with User Mode	Shared with User Mode	Shared with User Mode	Shared with User Mode
R1							
R2							
R3							
R4							
R5							
R6							
R7							
R8_usr	R8_fiq						
R9_usr	R9_fiq						
R10_usr	R10_fiq						
R11_usr	R11_fiq						
R12_usr	R12_fiq						
SP_usr	SP_fiq	SP_irq	SP_abt	SP_und	SP_svc	SP_mon	SP_hyp
LR_usr	LR_fiq	LR_irq	LR_abt	LR_und	LR_svc	LR_mon	LR_hyp
PC	Shared with User Mode						
CPSR							
	SPSR_fiq	SPSR_irq	SPSR_abt	SPSR_und	SPSR_svc	SPSR_mon	SPSR_hyp

Security Extensions Only

Virtualization Extensions Only

3. Instruction sets

Current ARM processors support several instruction sets.

- The classic ARM instruction set, in which all instructions are 32-bit.

- The Thumb instruction set, introduced in ARMv4T and in which all instructions are 16-bit, greatly improves code density. In Cortex processors, Thumb-2 technology adds 32-bit instructions to the Thumb instruction set providing increased performance while maintaining the high code density of the original Thumb instruction set.
- The NEON instruction set is a wide SIMD processing architecture, optionally supported by ARMv7-A processors.

Of the ARM processors available on the market today, all support the ARM and Thumb instruction sets as a minimum, with the exception of ARMv7-M devices which support only the Thumb instruction set.

4. Exceptions and interrupts

ARM supports eight basic exception types. External interrupts are mapped to the FIQ and IRQ exceptions. Other exceptions are used for external events (e.g. bus errors), internal events (e.g. undefined instructions or memory address translation faults), or software interrupts. Software interrupts are caused synchronously by execution of an SVC (Supervisor Call), SMC (Secure Monitor Call) or HVC (Hypervisor Call) instruction.

Later ARM processors implement a standardized Generic Interrupt Controller, which provides interrupt prioritization, pre-emption, configuration, distribution, masking etc in hardware.

5. Memory architecture

ARM processors have a 32-bit address bus providing a flat 4GB linear physical address space. Memory is addressed in bytes and can be accessed as 8-byte doublewords, 4-byte words, 2-byte halfwords or single bytes. Configuration options in the processor determine the endianness and alignment behavior of the memory interface.

ARMv7-A processors implement the VMSAv7 Virtual Memory System Architecture. This provides 32-bit virtual-physical address translation functionality. In the latest processors, like the Cortex-A15, this is extended (in the form of the Large Physical Address Extensions) to provide 40-bit physical addressing (see 2.2.5 above).

The architecture supports up to 8 levels of cache, with current implementations typically supporting 2 levels. The architecture permits several options with respect to virtual or physical indexing and tagging of cache contents.

Multi-core processors (e.g. Cortex-A5MP, Cortex-A9MP, Cortex-A7 and Cortex-A15) provide coherency in the L1 data cache across up to four cores in a single cluster.

2.3.2 Extended features

This section describes the extended physical addressing and virtualization extensions to ARMv7-A. These are supported by the Cortex-A15 processor.

1. Large Physical Address Extensions

All ARMv7-A processors provide virtual-to-physical address translation via an integrated MMU. This is achieved by a two-level structure of page tables describing the address translation as well as memory attributes for each page. Page sizes from 16MB (termed a “supersection”) down to 4KB (termed a “small page”) are supported. A single level page table allows granularity of 1MB, with a second level of tables being required to allow smaller granularity.

In processors prior to the Cortex-A15, both virtual and physical addresses are 32-bit, allowing a linear 4GB address space.

The Cortex-A7 and Cortex-A15 implement the Large Physical Address Extensions (LPAE) which, via an extended translation scheme allows the generation of 40-bit physical addresses. The tables used in the LPAE extensions contain longer descriptors, providing mapping of addresses at granularity of 1GB, 2MB or 4KB using between one and three stages. In all cases, virtual addresses as issued by the processor are still 32-bit; it is the physical addresses issued to the memory system which can be up to 40 bits.

Processors implementing LPAE are backwards compatible with the existing 32-bit translation scheme and use of the extended addressing is optional.

6. Virtualization extensions

The virtualization extensions are intended to support implementation of a hypervisor environment using a combination of hardware and software support. The architectural extensions are in several parts.

- There is support for a second stage of virtual memory translation which is managed by the hypervisor. Note that this second stage of translation is supported via the LPAE translation mechanism so it follows that implementation of LPAE is an integral part of the virtualization extensions. This second stage of translation allows a hypervisor complete control of the physical address map output by the processor and this can be changed dynamically to support the needs of different “guest” systems. In this way, guest systems can be kept isolated from each other and each can be presented with a complete virtual memory system which it “owns”.
- A defined set of control and configuration registers are “banked” in hypervisor mode so that each guest system sees a different, private set of the registers. Access to these registers by a guest system causes a trap into Hypervisor mode so that the hypervisor can take appropriate steps to configure the system accordingly.
- A defined set of system events (e.g. exceptions) can be configured to cause direct entry to Hypervisor mode instead of taking the standard exception handling action. The hypervisor code can then process the exception event before scheduling a “virtual” exception to be handled by the appropriate guest system.

The combination of these features allows a hypervisor to manage and control system configuration to maintain isolation between guest systems. Each guest system operates within a separate virtual machine.

2.4 Debug

ARM provides debug using the industry-standard JTAG port. As standard, this uses a 5-wire connection. A 2-wire debug port is also available for use in applications where pin-count is at a premium.

Program trace, if implemented, is provided via a combination of additional logic within the chip and an external Trace Port Adapter unit connected to a Trace Port on the chip itself.

ARM’s CoreSight on-chip debug infrastructure allows chip designers to specify and build complex multi-core debug systems which allow synchronous trace and debug of multiple processors within a single device.

3 SH-4 and ARM compared

A device based around the SH-4A processor core supports the SH-4A version of the SuperH architecture. The SuperH architecture was developed by Hitachi in the early 1990s, beginning with the SH-1. The SH-2, SH-2A, SH-3, SH-4 and SH-4A have continued the evolution of the architecture with various extensions and performance improvements. After brief ownership by SuperH Inc (a joint venture between Hitachi and ST Microelectronics), the SuperH product line is now developed by Renesas Technology. They market a large range of devices using SuperH cores, the current highest performance devices being those built around the SH-4A processor.

Having 32-bit general purpose registers, SuperH is referred to as a 32-bit architecture. However, unusually it has a 16-bit instruction set. This dates from the need, at its introduction, to enable efficient use of narrow memory devices. The result is impressive code density, similar to that achieved by ARM processors when using the Thumb instruction set.

The SH-4A processor core is used in a huge range of devices, matching different capabilities, peripheral sets, performance points and packaging requirements. The discussion in this document uses the SH7724 series of devices as a reference point for devices supporting the SH-4A architecture. This is a single-core device aimed at mobile applications.

Similarly, there are many devices supporting the ARMv7-A architecture and the discussion below applies equally to all of them. Although underlying implementations may differ, the architecture guarantees identical functional behavior at instruction level.

3.1 Programmer's model

3.1.1 Register set

The SH-4A architecture supports 16 general purpose registers. There is a banking scheme, similar in some ways to that found in the ARM architecture, which provides an alternate set of R0-R7 which can be accessed in privileged mode. The remaining registers are unbanked and some are only accessible in privileged mode.

The table below only shows the general purpose registers.

Register Name	User Mode		Privileged Mode (RB=1)		Privileged Mode (RB=0)	
R0 : R7	R0 : R7	BANK0	R0 : R7	BANK1	R0 : R7	BANK0
R8 : R15	R8 : R15		R8 : R15		R8 : R15	
R0_BANK : R7_BANK			R0 : R7	BANK0 (LDC/STC only)	R0 : R7	BANK1 (LDC/STC only)

The table shows how the RB bit selects the active register bank when in privileged mode. Note also that the other, non-selected, bank is also accessible in privileged mode but can only be accessed using the LDC/STC instructions. Again, this is analogous to the ARM architecture, which provides access to the User mode registers via LDM/STM instructions when in a privileged mode.

If the optional FPU extension is included, there are thirty two 32-bit floating point registers, organized in two banks of sixteen registers per bank. The FR bit in the FPSCR indicates which bank is currently selected for use. These registers are named FR0-FR15, with the alternate bank accessible as XF0-XR15.

The names DR0-DR14 may be used to access pairs of registers to store double-precision values. Four-element single-precision vectors may be accessed using the names FV0-FV4. XMTRX refers to all 16 alternate registers as a single 4x4 matrix.

3.1.2 Status registers

Like the ARM architecture's CPSR, SH-4A defines a single Status Register, SR. There is considerable overlap in the content.

This table shows the bit assignments in the SH-4A Status Register (SR).

Bit(s)	Name & Function	ARM Equivalent
31	Reserved	
30	**MD - Processing Mode** 0: User Mode 1: Privileged Mode	The 5-bit Mode field in the CPSR sets the current operating mode. With the exception of User mode, all are privileged.
29	**RB - Register Bank Specification** Selects privileged mode register bank	None. ARM register banking is an automatic feature of operating mode.
28	**BL - Exception/Interrupt Block** Masks interrupts	The I and F bits in the CPSR mask IRQ and FIQ respectively.
27-16	Reserved	
15	**FD - FPU Disable** Disables Floating Point Unit	Access to coprocessors is enabled or disabled via CP15.
14-10	Reserved	
9-8	**M, Q** Used by DIV0S, DIV0U, DIV1	None. ARM division instructions do not need initial conditions to be set, nor do they set ALU status flags on results.
7-4	**IMASK - Interrupt Mask Level** Masks interrupts whose priority is equal to or less than the value of IMASK	The eight basic exception types have a hard-wired priority scheme. For external interrupts, prioritization is a feature of the interrupt controller rather than the core itself.
3-2	Reserved	
1	**S** Used by MAC	None.
0	**T** Indicates True/False, Carry/Borrow or Overflow/Underflow	The CPSR contains a full-featured set of ALU status flags (NZCV) which can be set by data processing instructions.

The BL bit is automatically set following reset and on entry to an exception or interrupt handler. Similarly, the ARM core automatically disables IRQ on entry to any exception and also disables FIQ on entry to an FIQ exception handler.

The SR can only be modified in privileged mode (because the LDC instructions are not accessible in user mode).

For comparison, the ARMv7-A CPSR is as shown in the following table.

Bit(s)	Name	SH-4A Equivalent
31-28	**NZCV – Condition code flags** Set by data processing instructions according to the result.	SH-4A provides only the T bit and this is only set by comparison instructions
27	**Q – Sticky Overflow Flag** Records cumulative saturation.	None
26-25	**IT[de] – IF THEN state bits** Record state information during execution of an IT block. Not user-modifiable.	None
24	**J – Jazelle bit** Indicates that the processor is in Jazelle state.	None
32-20	Reserved	
19-16	**GE – SIMD ALU status flags** Used by SIMD instructions to record multiple status bits from up to four arithmetic operations.	None
15-10	**IT[abc] – IF THEN state bits** Record state information during execution of an IT block. Not user-modifiable.	None
9	**E – Data Endianness** Controls the endianness of the data memory interface.	None. SH-4A allows endianness to be changed via a hardware configuration signal at reset only.
8	**A – Asynchronous Abort Disable** Enables/disables handling of asynchronous data aborts.	None.
7	**I – Interrupt disable** Enables/disables IRQ interrupts.	The BL bit disables all interrupts.
6	**F – Fast interrupt disable** Enables/disables FIQ interrupts	The BL bit disables all interrupts.
5	**T – Thumb execution state** Indicates that the processor is in Thumb state.	None
4-0	**Mode – Mode field** Indicates the current operating mode.	The MD bit selects between Privileged and User modes, the only choices available.

User mode programs may read all bits in CPSR but can only modify the ALU flags (NZCVQ). Other fields, with the exception of T, J and IT, may be modified directly when in privileged modes. T and J bits may only be changed indirectly by execution of instructions like BX and BXJ; the IT field is used by the Thumb-2 IT instruction and is not user-modifiable.

3.1.3 Other special-purpose registers

The ARM architecture does not define any other registers - configuration and control functions which are not covered by the CPSR are accessed via registers in the notional coprocessor 15 (CP15). These are accessed via standard coprocessor instructions.

SH-4A by comparison defines a number of special-purpose registers. Some are concerned with the operation of specific instructions (e.g. the MACH/MACL registers which hold results from MUL or MAC instructions, the PR register which is used to save the return address for a subroutine call), others with specific functional blocks (e.g. FPUL and FPSCR which are used to configure and control the FPU).

The following additional registers are defined in the SH-4A architecture. Where appropriate, the ARM equivalent is also given.

Register	Description	ARM Equivalent
SSR	**Saved Status Register** SR is automatically copied to SSR on entry to an exception or interrupt handler.	CPSR is automatically saved to SPSR (of which there is one for each exception mode) on entry to an exception.
SPC	**Saved Program Counter** PC is automatically copied to SPC on entry to an exception or interrupt handler in order to provide a return address.	The return address is stored in Link Register (R14) for the appropriate exception handling mode on entry to an exception.
GBR	**Global Base register** Used as a base register in several addressing modes.	There is no equivalent as all general purpose registers can be used as base registers in all addressing modes.
VBR	**Vector Base Register** Holds the base address for exception and interrupt vectors. Each exception vectors to a fixed offset from this address.	ARMv7-A devices which support the virtualization extensions use CP15 registers to configure this. In other devices the vector table is fixed (either at 0x0 or 0xFFFF0000) but may be remapped in physical space via the MMU.
SGR	**Saved General Register 15** R15 is automatically saved to SGR on entry to an exception or interrupt handler.	ARM has no equivalent to this. Instead, the register banking scheme automatically preserves at least three registers on entry to any exception.
DBR	**Debug Base Register** When “user break debugging” is enabled, this is used as the vector base address for user break handling instead of VBR.	None
MACH **MACL**	**Multiply-and-Accumulate High/Low** Used to store the 64-bit accumulator in MAC operations and the result of a MUL operation. ARM has no equivalent as these instructions use the general purpose register set.	None. Multiply and Multiply-accumulate results are stored in general purpose registers.
PR	**Procedure Register** Used to store the return address when a subroutine call (BSR, BSRF, or JSR) is executed. The RTS instruction branches to the address in this register. ARMv7-A uses R14 (also referred to as LR, or the Link Register).	Return addresses from BL or BLX instructions are placed in the Link Register (R14).

Register	Description	ARM Equivalent
PC	**Program Counter** Indicates the address of the execution currently being executed.	The program counter in the Program Counter (R15).
FPUL	**Floating Point Communication Register** Used to transfer information between the FPU and the CPU.	FPU control and configuration registers are in CP10 and CP11.
FPSCR	**Floating Point Status and Control Register** Contains control bits and status bits for the FPU.	FPU control and configuration registers are in CP10 and CP11.

The SH-4A architecture also defines two regions of memory-mapped registers (address range 0x1C00:0000 to 0x1FFF:FFFF and an alias at 0xFC00:0000 to 0xFFFF:FFFF). The regions contain the same registers. The lower region (referred to as “Area 7”) is accessible in user mode but only via the MMU (i.e. it must use translated physical addresses); the upper region is accessed directly (not via the MMU) but only in privileged mode.

ARM devices do not define any architectural memory-mapped registers like this. Internal control and configuration operations are carried out using coprocessor instructions.

Name	Description	ARM equivalent
TRA	**TRAPA exception register** Contains 8-bit immediate code from TRAPA instruction.	The ARM core does not extract the SWI number from a SWI instruction. If required, the handler must extract it in software.
EXPEVT	**Exception event register** Contains a code identifying the exception type on entry to an exception. Needed because most exceptions vector to the same location.	ARM exceptions are divided into eight types which vector to separate addresses. How to decoding the exact event which caused the exception depends on which exception has occurred.
INTEVT	**Interrupt event register** Contains a code identifying the source of the external interrupt on entry to the handler. Note that all interrupts, including NMI vector to the same location.	FIQ and IRQ vector to separate locations so can easily be distinguished. IRQ determination is provided by the GIC.
PTEH	**Page table entry high** Contains the Virtual Page Number and ASID to be loaded into the TLB. Also captures these values when a TLB exception occurs.	Translation tables are held in external memory and TLB entries are populated automatically by the MMU. TLB exception information is captured in the MMU Fault Status Registers (IDSR and DFSR) on a TLB error.
PTEL	**Page table entry low** Holds the Physical Page Number and page management information to be loaded into the TLB. Also captures these values when a TLB exception occurs.	
TTB	**Translation table base** Holds the base address of the current page table. Has no associated hardware functionality.	The MMU carries out page table walks automatically and uses the TTB registers to determine the base address of the current page tables.

Name	Description	ARM equivalent
TEA	**TLB exception address** Holds the virtual address at which an MMU or address exception occurs.	The MMU DFAR and IFAR registers record essentially the same information following an address error.
MMUCR	**MMU control** Contains configuration and control bits for the MMU.	Similar functions are configured via CP15 registers. See the more detailed description of the MMUCR below for ARM equivalents.
PASCR	**Physical address space control** Contains one bit for each 64MB region of physical memory which enables or disables the write buffer for that region.	
IRMCR	**Instruction re-fetch inhibit** Enables or disables automatic instruction re-fetching when a range of resources are modified e.g. TLB or MMU. Generally, barrier sequences are recommended instead to avoid the resulting performance penalty.	ARM provides an Instruction Synchronization Barrier instruction which needs to be inserted following modifications of this type.
CCR	**Cache Control Register** Provides cache maintenance and control operations e.g. invalidation, enable/disable, write-through/write-back.	All operations of this type are carried out via CP15.
QACR0 **QACR1**	**Queue address control 0** **Queue address control 1** Allows output from the respective store queue to be mapped into physical memory when the MMU is disabled.	None. ARM cores do incorporate write buffers but this feature is not available.
RAMCR	**On-chip memory control** Configures the numbers of ways in the Instruction and Data caches. Also access restrictions to on-chip memory.	ARM fixes the associativity of the caches in the architecture and this cannot be changed either at SOC build time or under software control. Access permissions are set via the page tables used by the MMU.
LSA0 **LSA1**	**L memory transfer source 0** **L memory transfer source 1** Set the source address for block transfers to pages 0 and 1 in on-chip memory.	None.
LDA0 **LDA1**	**L memory transfer destination 0** **L memory transfer destination 1** Set the destination address for block transfers to pages 0 and 1 in on-chip memory.	None.
CPUPOM	**CPU operation mode** Controls speculative instruction fetches on subroutine return and whether the current interrupt mask is changed to the current level when an interrupt is executed.	None.

Name	Description	ARM equivalent
PVR **PRR**	**Processor and product version** Read-only registers which return version information for the product and processor core.	A set of registers, including MIDR (Main ID Register) and REVIDR (Revision ID Register) provide similar information regarding ARM devices.

The CPU Operation Mode register (CPUPOM) contains only two defined bits, as shown in the following table.

Bit(s)	Name		ARM Equivalent
31-6	Reserved		
5	RABD	**Speculative execution** Enables speculative instruction fetches for subroutine returns.	None
4	Reserved		
3	INTMU	**Interrupt mode switch** Enables automatic setting of the current interrupt mask to the level of an accepted interrupt.	None
2-0	Reserved		

The RABD bit is set to disable speculative instruction fetching during subroutine returns. When this feature is enabled, the core will pre-fetch instructions speculatively to speed up subroutine return. If there is a risk that such accesses may occur to illegal regions of memory or have undesirable side-effects, the feature may be disabled. In ARM systems, regions of memory in which accesses may cause side-effects are defined as “Device” memory and pre-fetching is automatically disabled in such regions by the core. ARM, therefore, neither needs nor supports an equivalent to this feature.

The INTMU bit controls whether the IMASK bits in the SR are automatically set to the current interrupt level on accepting an interrupt. This changes the way in which interrupt priorities affect interrupt handling. The ARM core itself does not support any equivalent feature as interrupt prioritization is configured in the interrupt controller (GIC) in ARM systems.

The following table shows the SH-4A MMU Control Register (MMUCR).

Bit(s)	Name	ARM Equivalent
31-26	**LRUI - Least Recently Used ITLB** Indicates the Least Recently Used entry in the ITLB. Can be used to implement an LRU replacement strategy.	ARM does not support LRU replacement strategies in the TLBs.
25-24	Reserved	
23-18	**URB - UTLB Replacement Boundary** Places an upper limit on the values taken by the ULTB replacement counter, thus implementing lockdown of higher numbered entries.	ARM cores support TLB lockdown via CP15 operations.
17-16	Reserved	
15-10	**URC - ULTB Replace Counter** Indicates which TLB entry is replaced by a LDTLB instruction. Incremented on every TLB access, thus implementing a pseudo-random replacement strategy.	ARM cores support random and round-robin replacement strategies for the TLBs.
9	**SQMD - Store Queue Mode Bit** Indicates whether user mode has access to the store queues.	None. ARM does not provide program access to the contents of the write buffers.
8	**SV - Single/Multiple VM Mode** Indicates whether ASID is used in address translation.	ARM supports a very similar ASID scheme which is used to improve efficiency in multi-tasking systems.
7-3	Reserved	
2	**TI - TLB Invalidate** Writing 1 invalidates all TLB entries.	TLB invalidation is a CP15 operation.
1	Reserved	
0	**AT - Address Translation Enable** Enables and disables the MMU.	The MMU is enabled/disabled via the M bit in the CP15 System Control Register.

3.1.4 Instruction set

The ARM instruction set has a fixed-size, fixed-format instruction coding. It has been complemented in all architecture revisions since ARMv4T with the Thumb instruction set (which is a 16-bit coding of a subset of the ARM instruction set) and Thumb-2 technology which added 32-bit instructions into the Thumb instruction set in all architectures from ARMv6T2 onwards.

Current ARM devices, therefore, support a mixed-size instruction set consisting of 16-bit and 32-bit encodings. However, strict alignment requirements still apply based on the original fixed-size instruction set – see 3.4.5 below.

The SH-4A architecture is a fixed-length instruction set using a 16-bit coding scheme.

Both instruction sets area based around a load/store architecture (i.e. data processing instructions cannot operate directly on the contents of memory).

Some of the major differences in the instruction sets are listed here. To the application programmer, writing in a high-level language, many of these differences may be of little importance as the compiler will produce the most efficient code in each case using the available instruction set.

However, there are some effects which are exposed to the programmer and which affect the way high-level language code should be written to make best use of the architecture. Further advice is given in section 4 below.

1. SH-4A load instructions are always signed

When loading data from memory into a register, values shorter than 32 bits are always sign-extended on transfer into a register. ARM provides signed (sign-extend) and unsigned (zero-extend) load instructions

2. SH-4A has branch delay slots

SH-4A's branches are delayed. This means that the instruction immediately following the branch instruction is executed before the instruction at the branch destination. This instruction falls in what is called a "branch delay slot". The BF (Branch False) and BT (Branch True) instructions have both delayed and non-delayed forms. All other branches, include RTS and RTE are always delayed.

3. SH-4A does not have a full set of ALU status flags

While there are several different variants of CMP (CMP/EQ, CMP/HS, CMP/GT etc) only one bit, SR.T, is available for recording the results of the comparison. Additionally, data processing instructions have only the T bit in which to store carry, overflow and shift-outs from shift./rotate operations. As a result only one of these can be stored and different variants of the instructions determine which is stored in the T bit. The DT (Decrement and Test) instruction is an exception – this decrements a register argument and sets SR.T if the result is zero.

ARM, in contrast, provides a full set of condition codes (NZCVQ) and allows them to be set by any data processing instruction. This means that an explicit comparison is not always necessary before a conditional operation and several different comparison results may be tested without repeating the comparison instruction itself.

4. ARM supports conditional execution

In the ARM instruction set, almost all instructions can be conditionally executed, using a 4-bit condition code within the instruction word. This allows the virtual elimination of short forward branches, very common in almost all code.

While the Thumb instruction set does not support this directly, Thumb-2 technology introduces the If-Then instruction which allows the construction of conditional blocks of instructions. This allows for very dense and efficient code generation.

5. SH-4A restricts constants to 8 bits

In SH-4A, the 16-bit instruction coding allows constants only up to 8 bits. Longer constants must be placed in memory and accessed using a PC-relative load instruction.

ARM also places restrictions on constant size and format but allows much greater flexibility. An arbitrary 32-bit constant can always be placed in a register using at most two instructions. There is also the option of placing constants in the instruction stream as mentioned above for SH-4A.

6. ARM instructions are typically 3-operand

The majority of ARM data processing instructions take three operands. All may be registers, one may be a constant. In contrast, SH-4A instructions typically have only two operands, making them destructive in nature (i.e. one of the operands is overwritten with the result). This makes ARM more flexible at instruction level. For example, adding two registers together and placing the result in a third can be achieved using a single instruction. The same operation takes two instructions on an SH-4A processor.

SH-4A	ARM
`MOV R1, R0 ; R0 = R1` `ADD R2, R0 ; R0 = R0 + R2`	`add r0, r1, r2 ; r0 = r1 + r2`

7. ARM allows greater branch offsets for relative branches

PC-relative branch instructions in SH-4A have an offset of +/-256 or 4092 bytes (an 8-bit or 12-bit signed displacement is encoded in the instruction) with the conditional branches having the shorter range.

The equivalent ARM instructions support a range of +/- 16MB or 32MB (depending on the instruction set in use, with Thumb having the shorter range), with the range being the same for conditional and unconditional branches.

Both instruction sets support longer branches via instructions which specify the destination address in a register, allowing a variety of absolute branch instructions.

Note that SH-4A does not support conditional subroutine calls, while the ARM BL instruction can be fully conditional.

ARM supports some extra branch instructions with smaller ranges (CBZ/CBNZ, TBB/TBH) which are designed to optimize particular high-level language constructs.

8. ARM does not have a dedicated subroutine return instruction

The SH-4A instruction set contains the RTS instruction which branches to the address in the Procedure Register (PR). Note that if the return address was placed on the stack on entry to the procedure, it needs to be retrieved into PR before return is possible.

ARM permits more flexibility but does not have an explicit procedure return instruction. Specifically, to return to the address in the Link Register (LR), a BX LR instruction is used; to return to an address on the stack, the return address can be loaded directly into the Program Counter (PC) using either an LDR or LDM instruction.

9. SH-4A has dedicated control instructions

As mentioned earlier, ARM system management and configuration functions are generally mapped to coprocessor instruction targeting CP15.

In contrast, SH-4A defines a set of specific management instructions (e.g. ICBI for invalidating part of the instruction cache) which are dedicated to certain operations.

These instructions are described in more detail in later sections.

3.1.5 Operating modes

The SH-4A architecture supports two operating modes: user and privileged. As described above, the register set in use is partially determined by the current operating mode (see 3.1.1). Additionally, some instructions are restricted for use in privileged mode only.

Privileged mode is entered automatically on reset and on entry to an exception or interrupt handler. Operating mode is reflected in the MD bit in the Status Register (SR). This bit may only be changed when operating in a privileged mode, providing the necessary restrictions on access to privileged mode functionality.

An ARMv7-A device has up to 9 basic operating modes. The current mode is encoded in a single field of the CPSR and changing mode in software is generally achieved by directly modifying these bits.

The only ARM modes which involve enabling additional functionality are Hyp (which enables features for virtualization) and Monitor (which is used in the context of TrustZone for secure applications).

Mode	Description		
Supervisor (SVC)	Entered on reset and when a Supervisor Call (SVC instruction is executed	**Privileged modes**	**Exception modes**
FIQ	Entered when a high priority (fast) interrupt is raised		
IRQ	Entered when a normal priority interrupt is raised		
Abort	Used to handle memory access violations		
Undef	Used to handle undefined instructions		
Hyp	For hypervisor code		
Monitor	For secure TrustZone systems		
System	Privileged mode using the same registers as User mode		
User	Mode in which most Applications and OS tasks run	**Unprivileged mode**	

Generally, there is little need for an ARM application to change mode explicitly. The appropriate mode is entered when an exception is handled by the processor. For example, the processor will enter IRQ mode automatically when handling an IRQ exception as a result of an external interrupt.

As far as the programmer is concerned, the most common mode change is from User mode (in which most user programs and tasks execute) to Supervisor mode (in which the Operating System and drivers execute) in order to access privileged OS functionality. This is achieved by executing an SVC instruction (the equivalent instruction in SH-4A is TRAPA). This causes an automatic switch into Supervisor mode and enters the Operating System via the SVC handler. On completion of the handler, the processor will automatically return to User mode. This mode change is included in the operating system calls which are invoked by the application and there is no need for the application programmer to manually change mode at all.

Following reset (in Supervisor mode), the startup code completes all system initialization in privileged modes and then optionally switches into User mode (or System mode if the user application is to run with privilege) before calling the main application entry point. The mode bits can only be modified when running in a privileged mode so user tasks are prevented from accessing privileged mode functionality.

3.1.6 Stack

SH-4A systems use R15 as the stack pointer. This register has the advantage of being automatically preserved by exceptions as it is saved in the SGR register on exception entry. To permit re-entrant interrupts, it will need to be saved before interrupts are re-enabled.

The ARM ABI specifies that ARM systems must use R13 as the stack pointer. This is one of the banked registers, having a banked copy in all of the exception modes, so is also automatically preserved (for non-reentrant exceptions) by the mode change which occurs on exception entry. Similarly, to permit re-entrant interrupts, R13 will need to be saved prior to re-enabling interrupts.

The SH-4A instruction set does not include any dedicated stack operations.

Any ARM Load and Store instructions may be used for stack access. Additionally, the ARM instruction set defines a set of Load and Store Multiple instructions which are ideal for transferring large numbers of registers to and from the local stack. The Thumb instruction set defines PUSH and POP instructions which implement a Full Descending stack on R13. ARM stack accesses are always word sized.

Operation	SH-4A	ARM
PUSH single	MOV.L R2, @-R15	STR r2, [sp, #-4]!
POP single	MOV.L @R15+, R2	LDR r2, [sp], #4
PUSH multiple	N/A	PUSH {r0-r12} // or STMFD sp!, {r0-r12}
POP multiple	N/A	POP {r0-r12} // or LDMFD sp!, {r0-r12}
PUSH status	STC.L SR, @-R15	MRS r0, cpsr STR r0, [sp, #-4]!
POP status	LDC.L @R15+, SR	LDR r0. [sp], #4 MSR cpsr, r0

The ARM assembler provides PUSH and POP mnemonics which correspond to the LDMFD/STMFD instructions shown above.

ARM provides the SRS instruction which can be used to save SPSR and LR onto the stack during an exception entry sequence.

Note that the instructions shown for saving and restoring SR in the SH-4A architecture are only valid in privileged mode. With a few exceptions, it is not possible to load/store system registers when in user mode.

3.1.7 Code execution

Both SH-4A and ARMv7-A class processors employ pipelines to improve instruction throughput. They also implement multiple execution units so that several instructions can be executed in parallel.

Analysis of similarities or differences between the pipeline structures is beyond the scope of this document.

3.2 System control and configuration registers

The ARM architecture makes use of the coprocessor instruction space for system control and configuration. This provides instructions to control cache, memory systems, branch prediction, clocking etc.

The SH-4A architecture achieves this using a combination of special purpose registers and dedicated instructions.

The special purpose registers are listed in section 3.1.3 above. Special purpose instructions defined in SH-4A include the following:

Instruction	Purpose	ARM Equivalent
DIV0S **DIV0U**	Used to set initial conditions before iterating the DIV1 instruction to carry out signed or unsigned division	There are no status bits or registers specifically associated with division instructions. Note that not all ARMv7-A cores include hardware divide instructions.
CLRMAC	Clears the MACH/MACL registers used to store accumulation results during MAC operations.	All multiply and multiply-accumulate operations use general purpose registers. No special registers are required.
ICBI **OCBI**	Invalidate Instruction or Operand (Data) Cache	These operations are carried out using CP15 instructions.
OCBP **OCBWB**	Write back Data cache block with optional invalidation	
LDTLB	Load TLB entry	Since the MMU automatically fetches translation information from page tables in memory, population of the TLB is automatic.

For example, to flush and invalidate the data cache on an SH-4A device, you use the OCBP instruction. On an ARM processor, you use an MCR instruction which transfers parameters to a specific register in the notional coprocessor 15.

The ARM mechanism avoids pollution of the register set (or memory map) with special purpose registers and also allows a large number of operations to be expressed using a small number of instructions, thus also conserving instruction set space and simplifying instruction decode logic.

3.3 Exceptions and interrupts

The exception and interrupt architecture of SH-4A and ARMv7-A processors is beyond the scope of this document. It is handled almost universally by the operating system and associated device drivers.

Both architectures support external (or hardware) interrupts and internal (or software) exceptions. Both are handled essentially in the same way. In both cases, external interrupt controllers are used to minimize the software effort required to prioritize, decode and vector exceptions. ARM uses a standard Generic Interrupt Controller (GIC); many SH-4A devices support the INTC architecture though this is often supplemented by an additional external interrupt controller.

3.4 Memory

Both ARM and SH-4A support an essentially flat 4GB byte-addressable memory space.

In ARM systems, both virtual and physical address spaces are 4GB (32-bit addressing). In systems which support the Large Physical Address Extensions, physical addressing is extended to 40-bit, giving a 1TB physical address space (in these systems, virtual addressing remains at 32-bit). See 2.2.5 above for further information.

SH-4A supports 32-bit (4GB) virtual addressing but translates this to a 29-bit physical address space (though most SH-4A devices support an extension of the physical address space to 32-bit for virtual addresses lying within a fixed 1GB region), It also defines several fixed regions of memory.

3.4.1 Memory map

SH-4A divides the virtual memory map into regions. When address translation is enabled (MMUCR:AT bit set to 1), virtual addresses are translated in the regions indicated in the table. Address translation is not available in other regions.

Address	User Mode	Privileged Mode
FFFF:FFFF : E600:0000	No access	**P4 Area** Non-cacheable No address translation
E5FF:FFFF : E500:0000	**On-chip memory**	
E4FF:FFFF : E400:0000	No access	
E3FF:FFFF : E000:0000	**Store queue area**	
DFFF:FFFF : C000:0000	No access	**P3 Area** Cachable Address translation
BFFF:FFFF : A000:0000		**P2 Area** Non-cacheable No address translation
9FFF:FFFF : 8000:0000		**P1 Area** Cacheable No address translation
7FFF:FFFF : 0000:0000	**U0 Area** Cacheable Address translation	**P0 Area** Cacheable Address translation

As you can see from the table, SH-4A also only permits cached accesses in certain regions of memory. Cache can be disabled in these regions via attributes in the TLB translation information.

Within the SH-4A memory map, there are also a set of memory-mapped control registers. These are mapped to 0x1C00:0000 and aliased at 0xFC00:0000. Accesses to the lower region is permitted in User mode but must be via virtual addresses, translated by the MMU; accesses to the higher region are only permitted in privileged mode.

When the optional 32-bit “Address Extended Mode” is available in SH-4A devices, areas P1 and P2 may be mapped anywhere within a 32-bit physical address space. The translation is carried out using information stored separately from the main TLB and is only available in privileged mode. It may be used for instruction or data accesses.

ARM, in contrast, treats the entire virtual address map as flat and does not impose any fixed memory map or special properties on any particular address regions. Instead, the MMU is used to apply attributes to memory regions which control the way in which the core accesses them.

In ARM systems the only item which has a fixed memory location is the exception vector table, which defaults to address 0x0000:0000. Even then, this can be relocated (either by hardware configuration at reset or under software control) to 0xFFFF:0000. Also, since it

lies in physical memory space and the core issues virtual addresses for exception vectors, it can be relocated to any address via the MMU.

Implementers of devices based on ARMv7-A processors are free to implement various types of memory anywhere within the memory map. You should refer to the documentation for the device to determine the amount and location of e.g. flash, RAM, ROM, peripherals etc. See section 3.4.2 below for more detail on memory types available in ARM systems.

There is a recommended reference memory map for ARM devices.

3.4.2 ARM Memory Types

The ARMv7-A architecture defines a set of memory types that can be set individually for each page in memory depending on the type of accesses which are to be made and the behavior which is required.

Normal

Normal Memory is the highest-performance memory in the system as it is subject to the fewest restrictions. For instance, the processor is free to carry out speculative reads, to repeat and re-order memory accesses (as long as program behavior is preserved). Normal memory may be cached and may use a write buffer. The majority of the memory in a typical system will be “Normal”.

Strongly Ordered

Strongly Ordered memory is subject to much greater restrictions and is only used in certain infrequent situations which require preservation of memory access ordering. Strongly Ordered memory may not be cached or buffered and the number, order and type of accesses must be as in the program. Speculative accesses are also forbidden.

Device

Device memory is intended for regions which cover memory-mapped peripherals and other regions in which memory accesses may have side-effects. Access number, type and order is guaranteed. Device memory is uncached but write buffers may be used.

In a virtual memory system (as will be the case with almost all platform operating systems), this information is handled as part of the virtual-physical memory translation configuration. Processors start up with address translation disabled and all memory treated as Strongly Ordered. The translation configuration and definition of memory types for all active regions is part of Operating System initialization.

3.4.3 Virtual memory

Both architectures include support for virtual-to-physical address translation and this can be used to implement a full demand-paged operating system environment. Both include explicit support for the task switching and context-saving operations typically used by platform operating systems.

In SH-4A systems, explicit software support is required to load translation data into the TLB. This is triggered by an MMU or address error exception. ARM MMU’s contain hardware which automatically retrieves translation information from external page tables and this software support is not required.

Note that SH-4A systems only allow address translation in certain address regions (see the table above). ARM systems allow translation across the entire address range.

Note also that the smallest page size supported by the ARM MMU is 4KB. The SH-4A supports page sizes down to 1KB.

The details of implementing a virtual memory system are beyond the scope of this document.

3.4.4 Memory access control

In both architectures, memory access control is implemented as part of the virtual memory system.

Both architectures provide facilities for read-only and read-write permissions. Permissions can also be policed separately for user and privileged mode accesses. ARM also provides the ability to set execute permissions.

3.4.5 Access types, endianness and alignment

Both architectures support byte, halfword and word accesses as standard in the instruction set. ARM systems also support doubleword accesses.

Both architectures support connection to big-endian or little-endian data memory systems. SH-4A devices sense an external pin on reset to set the endianness. ARMv7-A processors also do this but additionally support the ability to change the data endianness at run-time under software control.

In SH-4A systems, some memory-mapped registers are always accessed in big-endian mode, even when little-endian mode is selected.

In SH-4A systems, the endianness configuration applies to both instruction and data memory interfaces. ARM systems require that code memory is little-endian and that all instructions are stored in little-endian byte order, regardless of the endianness configuration of the data memory interface.

Both architectures include instructions for swapping byte order within a register to assist with efficient processing of mixed-endian data. ARM's, though, are more flexible.

Both architectures are most efficient when loading/storing to addresses which are aligned according to the transfer size e.g. (using the ARM terminology) words to/from word-aligned addresses. Both do support the ability to use non-aligned addresses but in slightly different ways.

1. Code alignment

ARM processors require instructions to be correctly aligned in memory. When operating in ARM state, all instructions are 32-bit and must be word aligned; in Thumb state, instructions may be either 16-bit or 32-bit and must all be halfword aligned.

The SH-4A instruction set encoding is entirely 16-bit. All instructions must be properly aligned in memory.

2. Data alignment

All ARM processors access data in memory more efficiently if it is aligned according to size (words on word boundaries, halfwords on halfword boundaries etc.). All ARMv7-A processors, however, are capable of accessing unaligned data, albeit with a slight performance penalty (this is due to the need for the memory interface to make multiple accesses and is hidden, functionally, from the programmer). Notable exceptions are stack accesses which are all word-sized and must be word-aligned. Some memory access instructions (e.g. LDM/STM and LDREX/STREX) do not support unaligned addresses.

ARMv7-A devices may be configured to support unaligned data access via a bit in CP15. This feature is normally disabled at reset to maximize backwards compatibility. When enabled, unaligned data access is transparent to the program but may involve a small performance penalty as the bus interface unit still needs to carry out multiple aligned accesses to retrieve or store the data. This allows compilers to produce code which is safe regardless of run-time alignment.

Use of an unaligned address with a standard MOV instruction on an SH-4A processor causes an address error. Instead, software should use the MOVUA instruction which supports transfers to/from unaligned addresses. The need to use a separate instruction means that compilers need to know alignment information at compile-time so that the right instruction can be used. This instruction involves an extra cycle compared to standard aligned transfers.

3.4.6 Atomicity

Naturally aligned accesses up to word size are guaranteed atomic in ARM systems. In some circumstances e.g. exclusive accesses, doubleword accesses are not guaranteed atomic.

3.4.7 Barriers and synchronization

There are cases in program execution where it is necessary or desirable to ensure that certain memory accesses are completed in a known order. Examples are self-modifying code and accesses to peripheral registers.

The ARM system of memory typing (in which memory is defined as “Normal”, “Device” or “Strongly Ordered”) and the standard memory model used by ARMv7-A and SH-4A both ensure that this is the case in normal code execution. However, there may be cases where the program needs to explicitly ensure ordering. Both architectures provide for this via barrier and synchronization instructions.

ARM provides three memory barrier instructions.

DMB – Data Memory Barrier
This ensures that all memory accesses prior to the barrier are completed before any memory accesses following it.

DSB – Data Synchronization Barrier
A DSB ensures that no instructions following the barrier execute until all memory accesses prior to the barrier have completed.

ISB – Instruction Synchronization barrier
An ISB ensures that any instructions following the barrier are refetched from cache prior to being executed (equivalent to flushing the pipeline and any prefetch buffers).

SH-4A provides the SYNCO instruction, which stalls the pipeline until all previous instructions (including any resulting data accesses) have completed. This corresponds broadly to a DSB instruction. There is no direct SH-4A equivalent to DMB.

SH-4A does not define an equivalent to the ISB instruction. For self-modifying code, SH-4A documentation recommends a barrier sequence consisting of a SYNCO followed by a cache operation which purges and reloads the Instruction Cache. The exact sequence depends on the caching mode for the particular area of memory involved.

Note, though, that the necessary cache management operations for self-modifying code may only be executed in privileged mode in the ARM architecture. This makes such techniques less efficient than some other architectures.

For further information, see the ARM document “Barrier Litmus Tests and Cookbook“, listed in the references. See also 4.13 below.

3.4.8 Shared memory

ARM supports (via bits in the page tables) the definition of shared and non-shared regions of memory on a per-page basis. This information is used by the system when implementing coherency and also when determining whether to use a Local or Global monitor to arbitrate on exclusive accesses. Some ARM processors route accesses to non-shared memory regions via a separate, private memory bus (e.g. the Private Peripheral Interface found on some Cortex-A processors).

Multi-core SH-4A systems do not support an equivalent feature.

3.4.9 Caches

Both architectures support Harvard L1 Data and Instruction caches backed by a unified L2 cache. SH-4A devices though do not always have separate L1 Instruction and Data Caches, unified L1 cache is an option.

(Note that the Data Cache is often referred to as the “Operand Cache” in SH-4A documentation.)

Both architectures allow cache size to be configured at synthesis time. SH-4A devices also allow other parameters (e.g. the line length) to be configured also.

The differences in cache architecture (set associativity and size) are in general transparent to the programmer. However, these may have an effect on the performance of certain applications. When migrating, it is not necessary to address these issues from a functional perspective but it may be advisable to examine whether performance could be improved by revisiting them at a later stage.

Note that SH-4A devices permit direct access to the cache (and write buffer) contents via memory-mapped addresses. This is not possible in ARM systems.

3.4.10 Cache Aliasing

The SH-4A cache architecture suffers from a problem often referred to as “cache aliasing”. This occurs in systems which use Virtually-Indexes-Physically-Tagged (VIPT) caches and in which the size of a cache way is larger than the MMU page size. In such systems it is possible for the same page of physical memory to be mapped by several cache lines for which some bits of the physical tag differ. Avoiding such problems can be difficult in software and usually involves cleaning/invalidating the cache whenever relevant virtual address translation settings are changed.

ARMv7-A systems avoid this by implementing caches which are Physically-Indexed-Physically-Tagged (PIPT). At the expense of some additional complexity in the cache hardware, this removes the problem by ensuring that the mapping between cache contents and external memory does not change when the MMU configuration is modified.

3.4.11 SH-4A Store Queues

SH-4A devices have a pair of memory-mapped “store queues” which can be used to initiate burst transfers to memory. It is possible to implement very efficient DMA-like memory transfers using these.

ARM has no direct equivalent to this feature. However, similar behavior can be generated either by allowing store operations to merge in the write buffer (though this will not happen to Device memory) or by using STM instructions.

3.5 Self-modifying code

Self-modifying code is possible in both architectures. In both cases, sequences of serializing, cache management and/or barrier instructions are required to ensure that the correct instructions are executed after they have been written.

Refer to the documentation for further information. Note though, as has already been mentioned above, that the cache management operations which are usually required are only executable in privileged modes on ARM systems. This can make such techniques less efficient as Operating System calls are required to access them.

Note that this is separate from the need to generate code at run-time as part of a Dynamic Compilation of Just-in-Time Compilation environment. Solutions for this are widely available for ARM systems.

3.6 Debug

Both architectures provide for debug over the standard JTAG connections (often referred to as H-UDI in the SH-4A architecture). The underlying implementation, however, is rather different.

ARM CPUs support a debug “state” in which the processor is halted and isolated from the rest of the system. The processor can then be controlled from the external system via some on-chip logic (EmbeddedICE). ARM terms this “halt mode” debugging. By using a resident monitor, it is also possible to carry out “running-system debug” on ARM platforms – the method for doing this varies between debuggers.

Both architectures provide features which support instruction and data breakpoints and program trace. In general, the underlying implementation of debug facilities is transparent to the developer.

ARMv7-A processors also incorporate a range of configurable counters which can be used to capture data in a non-intrusive manner.

In general, programmers can expect the debug experience to be similar even though the underlying architecture is somewhat different.

3.7 Power management

As would be expected from processors which are often used in portable, battery-powered devices, both architectures support a variety of power-saving modes.

SH-4A devices provide the SLEEP instruction which causes the processor to enter either Sleep or Standby mode. This instruction can only be executed in privileged mode. Exit from Sleep mode is via an interrupt. Support is also provided for controlling the power/clock state of peripherals and other on-chip modules.

ARMv7-A devices support a range of power modes and incorporate facilities for linking this with device-wide power management schemes.

In both cases, making use of these is the responsibility of the Operating System. It is usually highly platform-dependent and managed by platform-specific firmware.

From the point of view of the application programmer, it is important to ensure that tasks indicate to the Operating System when they are idle. This gives the Operating System maximum opportunity to reduce power consumption or even power down the system as far as is possible. Consult your Operating System documentation for details of how you can best do this.

3.8 Multi-threading and multi-processing

Both architectures support multi-processing platforms i.e. devices in which two or more cores share a single memory system.

Making use of these facilities is the responsibility of the Operating System, provided that the application programmer has designed a suitable multi-threaded structure - how to do this efficiently is beyond the scope of this document.

Multi-core and multi-threaded versions of most standard Operating Systems are available for both architectures so porting applications is, in general, a trivial task.

3.9 Multimedia extensions

The SH-4A architecture does not define any instructions aimed at multimedia or vector processing. The optional FPU does support limited vectored and matrix operations on combinations of registers.

ARMv7-A devices support the optional NEON engine. This provides a set of vector processing instructions on a variety of data types, including floating point. The structured load capability of NEON is particularly powerful when processing large quantities of structured data.

Note that the FPU, if present, is automatically enabled at reset on SH-4A devices; on ARM systems which support FPU or NEON extensions, the relevant coprocessor instructions are automatically disabled and need to be enabled (under privileged software control) before any NEON/FPU instructions can be executed.

NEON instructions can be accessed in a variety of ways:

- Direct coding in assembly language

- Automatic vectorization by the C compiler
- C intrinsic functions

Some SH-4A devices incorporate DSP extensions via additional hardware. This is not compatible with NEON at an instruction level and code will need to be re-compiled. Assembler instructions will need to be rewritten completely.

There are several standard libraries available which implement standard DSP, filtering and SIMD processing functions using either architecture and it may be simplest to port your application to use one of these standard APIs. These libraries are often provided as part of the Operating System environment.

4 Migrating a software application

We assume that the majority of software applications are written in a high-level language such as C. It is accepted that small amounts of assembly code will be required to handle things like reset, initialization, interrupts and exceptions but that these code segments will be contained within the operating system and are therefore outside the scope of this document. To a lesser extent, assembly code may be used to obtain higher performance (e.g. in memory copy and floating-point arithmetic routines).

4.1 General considerations

4.1.1 Operating mode

A stand-alone application will most likely execute in privileged mode on an SH-4A device and either supervisor mode or system mode on ARM. In this case, no action is required as all other mode changes (on ARM, as a result of exceptions) will be automatic.

In an operating system environment, ARM applications will execute in user mode with the operating system in supervisor mode (or possibly system mode in some circumstances). By and large, the mode transitions are also automatic in this case, with supervisor mode being entered automatically on an exception and on execution of a Supervisor Call (SVC) instruction. The transitions back to user mode will happen automatically on return from the resulting exception.

Since entry to the operating system will be contained within a defined API, the application programmer need not be concerned with the details of changing mode.

However, the programmer needs to be aware that many operations on ARM systems cannot be carried out in User mode, as this mode is not privileged.

Examples include:

- Cache and TLB maintenance operations
- CPU ID and capability determination
- Cache architecture determination

In addition, NEON and Floating Point instructions can be restricted to privileged mode execution only. In practice, this feature is employed by Operating Systems to support “lazy context switching” and is not generally of concern to application programmers.

Applications needing to know information about the system and to access other privileged operations will need to use an Operating System API in order to do so.

4.1.2 Memory map

The memory map to be used by an application will be defined by the operating system environment. Setting the build tool configuration to match this requirement will result in an application which will run under the operating system.

4.1.3 Data types and alignment

Various standards exist for data types in the SH-4A architecture. Which is in use depends on which ABI is in force. The following example shows the ABI for System V. The correspondence between data types and the underlying machine is not part of the ARM architecture but is specified in the ARM Embedded Application Binary Interface (see references).

Type	ARM EABI	SH-4 (Hitachi compiler)	Notes
`char`	8-bit unsigned	8-bit signed	
`short`	16-bit	16-bit	
`int`	32-bit	32-bit	
`long`	32-bit	32-bit	
`long long`	64-bit	64-bit	
`float`	32-bit	32-bit	
`double`	64-bit	64-bit	
`long double`	64-bit	64-bit	
pointer	32-bit	32-bit	

Be careful with the default for sign of character types. This often differs between the two architectures but depends on the particular ABI in use.

As you can see, the basic types are generally in agreement in the example shown. You should check carefully, though, the ABI documentation for the tool chain and platform you are using to ensure that this is the case in your project.

All ARM types are naturally aligned on a boundary equal to their size. Note that this is not the case for all SH-4A types. For instance, a 64-bit “long long” is aligned on a doubleword boundary on an ARM platform but on a word boundary on an SH-4A platform. This can cause issues when accessing data structures which do not have natural alignment e.g. byte-oriented network data. Even after re-compilation, code which functions on an SH-4A system may not work correctly on an ARM system. This applies most often when accessing data structures which require non-native alignment. It can also apply where the programmer has not followed strict casting and aliasing rules in C. In these cases the compiler must be informed of potential mis-alignment. See chapter 5 below for more details.

Note that the length of a pointer type is still 32 bits in ARM systems which support the Large Physical Address Extensions. This extension to the architecture allows the processor to access a 40-bit physical address space via an extra address translation stage. This produces external 40-bit addresses but the input, from the program, is still a 32-bit pointer.

4.1.4 Calling conventions

When interfacing assembler code with high-level languages, it is necessary to conform to the correct conventions for usage of registers.

For ARM processors, almost all tools conform to the ARM Executable Application Binary Interface (EABI). The ARM Architecture Procedure Call Standard (AAPCS) is part of this and documentation can be found on ARM’s website (see references in 1.4 for details).

4.2 Tools configuration

Several compiler toolchains exist which support both ARM and SH-4A architectures. Vendors such as Greenhills and Microsoft, for instance, sell such products. Several open-source options are also available.

If you are already using a toolchain which supports ARM as a target architecture, the easiest option is to continue with the same tools.

In general, very little of the configuration of the tools will need to change beyond the following.

- Memory map, code and data placement
- Any options which relate to particular target SH-4A architectures, platforms, processors or boards. When deciding on the ARM options, it is good practice to be as specific as possible with respect to the processor and architecture you are using.
- If your application uses floating point, then you will need to configure carefully for either hardware floating point or soft emulation.

There is also the option of using the ARM tools. Refer to the documentation (all available on ARM's website) for further information on this.

More information on support for a variety of tools can be found here:

http://www.arm.com/community/software-enablement

4.3 Operating system

If you are currently using one of the many platform Operating Systems available within the industry, it is likely that a port will already exist for the ARM architecture. More details can be found here:

http://www.arm.com/community/software-enablement

4.4 Startup

The startup sequence of the processor is usually transparent to the application developer and is taken care of entirely within the Operating System.

ARM processors boot in SVC mode (which is privileged) and the OS will carry out all necessary platform configuration and software initialization before starting any user processes. User processes will generally execute in User mode.

4.5 Handling interrupts and exceptions

The handling of interrupts and exceptions is within the domain of the Operating System and related device drivers. Applications will use an OS-provided API to access this functionality.

4.6 Timing of NOP instructions

Programmers sometimes use NOP instructions to add delay into short timing loops.

SH-4A systems generally take one cycle to execute a NOP; ARM systems do not guarantee that a NOP instruction will consume any time at all (the branch prediction and issue hardware in the pipeline may, for instance, fold the instruction out).

4.7 Power Management

The power management options in an ARM-based device are likely to be more varied and than those available with an SH-4A device. See section 3.7 above for a more detailed description of the power management features provided by a typical ARM processor.

When using an operating system or real-time scheduler, it is likely that the power management features will have been built into the kernel and an API provided via which applications can signal changes of status to the Operating System power management

framework. Because power management infrastructure on ARM-powered devices is largely vendor and system dependent, pay careful attention to the documentation for the platform you are using.

When writing a bare metal application, you must insert appropriate instructions into your code to allow the hardware to sleep when possible. For instance, busy-wait loops should have WFI/WFE instructions inserted. However, it is more power-efficient to avoid polling in general and implement an interrupt-driven system with power management instructions in the main loop.

4.8 Hardware discovery

SH-4A systems define two registers which contain information about the core and system: Product Register (PRR) and Product Version Register (PVR). These registers contain version numbering information from which software must infer whether certain features are present or not.

ARM systems define a much larger set of version and feature registers (with CP15) which give similar information. In addition, they provide information regarding cache configuration, feature availability, instruction set and architecture extensions etc. Since much of this is configurable by the designer of the chip, software often needs to be able to determine this information during the startup sequence.

Details of features, extensions and version information is found in the ARM Architecture Reference Manual and also in the documentation for the specific device in use.

4.9 Accessing peripherals

In ARM-powered systems, all peripherals are memory-mapped. Implementation and system-dependent code is required to define the registers involved and locate them in memory at the appropriate addresses. These are then accessed using standard memory access instructions.

Points to note:

- LDM and STM instructions should be used with care as the architecture permits such access sequences to be abandoned and restarted in the event of e.g. an exception.
- Peripheral memory regions must be marked as Device memory to ensure access ordering is correctly observed.

The situation is generally similar to SH-4A systems.

Note that, when using a platform Operating System, this functionality will usually be within the kernel or associated device drivers.

4.10 C programming

In general, provided that the C source code is well-written and type-safe, there should be relatively few problems when re-compiling for ARM.

Clearly any inline assembler or architecturally-specific intrinsic functions will need to be removed, replaced or rewritten. Cache and memory management features are significantly different between the two architectures and will need rewriting.

ARM’s rules on data alignment are a little stricter than SH-4A. However, ARM processors supporting architecture ARMv6 and later are capable of supporting unaligned accesses in hardware. In Cortex-A processors, this feature is permanently enabled; on earlier processors which support backwards compatibility the feature defaults to disabled and can be enabled, if required, by setting the U bit in CP15 register c1. This minimizes issues when porting but special care must still be taken with packed or byte-oriented data.

Be careful though with any data which has been declared using special alignment attributes or pragmas. The declaration may need to be corrected to use the __packed keyword when using the ARM tools.

Check for code which depends on whether single-byte types (char) are signed or unsigned.

4.11 Assembly language programming

Any assembly code will need to be either replaced or rewritten.

In many cases, it will not be necessary completely to rewrite SH-4A assembly code as extensive optimized libraries are available for common functions targeting ARM platforms.

Another alternative is to rewrite short, common sequences using compiler intrinsic functions. These have the advantage of being more easily portable between different versions of the ARM architecture and avoid the need to hand-code in assembler.

4.12 Function pointers

In ARM programs, the least significant bit of a function pointer is used to indicate whether the target function is in ARM instructions (bit 0 of address is 0) or Thumb instructions (bit 0 of address is 1). The linker normally sets this bit when fixing up relocations using attributes in the object files to indicate the instruction set in use at each point.

Since instructions are always at least halfword-aligned, the actual address of the instruction can be determined simply by masking this bit before addressing through the pointer.

This can affect code which accesses or modifies jump tables, for instance.

4.13 Semaphores etc.

Implementation of semaphores, mutexes and similar constructs requires some architectural mechanism for carrying out an atomic exchange, typically between a register and a memory location. Both architectures support this via similar mechanisms of linked load and store operations.

The following shows a simple “test-and-set” lock construct implemented in both architectures.

SH-4A	ARM
get_lock **MOVLI.L @R0, R1** **CMP/EQ #0, R1** **BF get_lock** **MOV #1, R1** **MOVCO R1, @R0** **BF get_lock** **NOP** **SYNCO** **...critical code here...**	**get_lock** **LDREX r1, [r0]** **CMP r1, #0** **BNE get_lock** **MOV r1, #1** **STREX r2, r1, [r0]** **CMP r2, #0x0** **BNE get_lock** **DMB** **...critical code here...**
unlock **SYNCO** **MOV #0, R1** **MOVLI.L @R0, R1**	**unlock** **DMB** **MOV r1, #0** **STR r1, [r0]**

The SH-4A mechanism uses a flag, internal to the processor, to record the link between the load and store. It is capable of monitoring only one linked operation at a time and its scope if the entire address space.

In ARM systems, this “exclusive monitor” is notionally within the memory system. It is implementation-defined (i.e. up to the licensee) whether there is more than one monitor

and at what granularity it operates. You should refer to the documentation for your device to determine the exact configuration.

There are two points about this of note to software developers.

- The granularity at which the reservation is recorded is implementation-defined within a range of between 2 and 512 words. The size of this region is termed the “Exclusives Reservation Granule”. While correct operation will not be affected by doing so, it is good practice to avoid placing more than one lock variable within the same granule.
- Locks which are shared between processors must be located in memory regions marked as shared. This ensures that a global monitor is used i.e. one which is visible to both processors.

It is important to note and obey any guidelines in respect of clearing reservations during e.g. context switches. ARM provides the CLREX instruction to explicitly clear any outstanding reservations.

The Operating System will provide an API for a range of exclusion and interlocking operations.

5 A porting checklist

The following list of points may prove useful when porting source code from SH-4A to ARM.

- Recompile C/C++ code using tools which target the ARM architecture. In general, well-written, standards-conformant code should recompile without error when targeted for ARM.
- Check for data items and data structure members which are not naturally aligned. Depending on the defaults for the software platform, these may require adjustment. This issue can only arise where externally defined data are mapped by a program as compilers will align data naturally by default.
- Check carefully for code which depends on whether single-byte types are signed or unsigned. The default for ARM is unsigned. The default may be changed using the –signed_chars option to the compiler. However, doing this may introduce compatibility issues with standard libraries.
- Any instances of self-modifying or dynamically-generated code will need to be examined very carefully. Apart from the need to rewrite the assembly code involved, such sequences may not function correctly on ARM systems (see 3.5 above for explanation).
- Assembler procedures or inline assembly segments in C/C++ source code will need to be identified and rewritten, either in C/C++ or in ARM assembler. For ARM-standard components (e.g. VFP) C reference implementations are usually available which can simply be compiled. This may provide sufficient performance in the absence of an assembler version.
- Drivers for integrated devices (e.g. interrupt controllers, timers, MMU/TLB, hardware debug etc) will need to be replaced with ARM equivalents. When using an OS (e.g. Linux) which supports both architectures, there may be little or no impact here as much of this code will be contained with the OS.
- Drivers for hardware accelerators and other platform-dependent devices may need rewriting. In many cases, however, switching platform will remove these devices and possibly replace them with ARM equivalents. In these cases, the implications will largely be dealt with by changing drivers and compilation tools.
- Locate all accesses to system registers, system calls, platform-dependent driver calls etc and ensure that they are replaced with ARM-specific or platform-specific equivalents.
- Identify all uses of memory barriers and synchronization instructions in SH-4A source code and ensure that they are replaced with the ARM equivalents. Also examine carefully any code that may rely on the barrier side-effects of e.g. RTE instructions. It may be necessary to insert additional explicit barrier instructions and cache maintenance operations when porting.
- The power management strategy will need to be reformulated to match the features available on the target ARM device. This will be a combination of platform-dependent drivers and the interface with facilities provided by the OS.