

Application Note **245**

Migrating from Power Architecture to ARM

Document number: ARM DAI 0245

Issued: August 2012

Copyright ARM Limited 2012

The ARM logo, consisting of the letters 'ARM' in a bold, sans-serif font.

Application Note 245

Migrating from Power Architecture to ARM

Copyright © 2012 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
February 2011	A	First release
August 2012	B	Correction to section 3.4.6 on memory barriers

Proprietary notice

Words and logos marked with ® or © are registered trademarks or trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

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	References and Further Reading	4
2	ARM architecture features	5
2.1	ARM architecture versions.....	5
2.2	Architecture ARMv7-A extensions	5
2.3	Programmer's model	6
2.4	Debug	8
3	Power Architecture (PPC) and ARM compared	9
3.1	Programmer's model	9
3.2	System control and configuration registers.....	15
3.3	Exceptions and interrupts	15
3.4	Memory	18
3.5	Debug	22
3.6	Power management.....	23
4	Migrating a software application.....	25
4.1	General considerations.....	25
4.2	Tools configuration	27
4.3	Operating system.....	27
4.4	Startup	27
4.5	Handling interrupts and exceptions	28
4.6	Timing and delays.....	30
4.7	Power Management.....	30
4.8	Semaphores etc.....	31
4.9	Accessing peripherals.....	32
4.10	C programming	32
5	A porting checklist.....	33

1 Introduction

The purpose of this document is to highlight areas of interest for those involved in migrating software applications from Power Architecture to ARM platforms. No attempt is made to promote one 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 Power 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. It is, in essence, a RISC architecture. However, its evolution over the last 24 years has introduced many features and extensions which do not necessarily sit comfortably with the RISC ideal.

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 range of 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 References and Further Reading

(All ARM documentation referenced here may be downloaded directly from infocenter.arm.com)

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

Cortex-A9 Technical Reference Manual, ARM DDI 0388F

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

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

ARM Generic Interrupt Controller Architecture Specification, ARM IHI 0048A

Application Note 212 – Building Linux Applications using RVCT4.0 and the GNUT tools and Libraries, ARM DAI 0212A

Barrier Litmus Tests and Cookbook, ARM GENC 007826

2 ARM architecture features

ARM is a 32-bit architecture. As such, it has 32-bit registers, ALU, data paths, address and data buses. Additionally, in the native ARM instruction set, all instructions are 32 bits wide. Individual ARM implementations may have internal 64-bit paths for increased performance and throughput.

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 processor cores. 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-A8 and Cortex-A9 processors and supports fully-featured application class devices capable of running platform Operating Systems such as Linux, WinCE etc. It provides full virtual memory support and optional media processing, security and virtualization extensions.

ARMv7-R is available in the Cortex-R4 and is targeted as applications which require hard, predictable real-time performance. Devices incorporating a Cortex-R4 processor core 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 core. 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 core. Unless explicitly stated otherwise, we refer to the ARMv7-A architecture including the security and Java acceleration extensions as described in sections 2.2.1 and 2.2.3 below.

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 (VFPv3) 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.

2.2.3 Java acceleration

Two architectural extensions are available for accelerating Java and other dynamically compiled languages. Both Jazelle (acceleration for Java only by implementing hardware support for execution of bytecodes) and ThumbEE (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.

2.2.4 Multiprocessing

These provide for synchronization and coherency across a “cluster” of processor cores, operating either in Asymmetric or Symmetric Multi-Processing mode. This extension is currently supported by the Cortex-A9MP 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. It includes an extension to the VMSAv7 virtual memory architecture which allows the generation of 40-bit physical addresses from 32-bit virtual addresses.

These extensions are currently implemented only in the Cortex-A15 processor.

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, this allowing implementation of a software hypervisor.

These extensions are currently implemented only in the Cortex-A15 processor.

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 different model for modes and exceptions.

2.3.1 Standard features

1. Operating modes

The ARM core supports seven operating modes. All of these, with the exception of User mode, are privileged. Five modes (Supervisor, Undefined, Abort, FIQ and IRQ) are associated with handling particular types of exception events. Applications generally run either in User mode (unprivileged) or System mode (privileged) with the operating system, if one is used, running in Supervisor mode.

For ARMv7-A profile cores, the optional virtualization and security extensions introduce another two modes. These are Hyp, which is intended to support implementation of a hypervisor, and Monitor, which acts as a gatekeeper between the secure and non-secure operating states.

2. Register set

The ARM register set consists of 37 general-purpose registers, 16 of which are usable at any one time. The subset which is usable is determined by the current operation mode (see above).

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.

Five of the operating 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 User mode register is used.

User/System	FIQ	IRQ	Abort	Undef	SVC
R0					
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8	R8				
R9	R9				
R10	R10				
R11	R11				
R12	R12				
R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)	R13 (SP)
R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)	R14 (LR)
R15 (PC)					
CPSR					
	SPSR	SPSR	SPSR	SPSR	SPSR

In cores which implement the virtualization and security extensions to the ARMv7-A profile, there are extra banked registers associated with Hyp mode and Monitor mode. These are not discussed in this document.

3. Instruction sets

Current ARM cores may support several instruction sets:

- The native ARM instruction set, in which all instructions are 32-bit.
- The Thumb instruction set (introduced in ARMv4T), in which all instructions are 16-bit. This provides greatly improved code density.
- Java bytecode – cores which include the Jazelle-DBX extension are capable of executing Java bytecode directly in hardware.

The ARM and Thumb instruction sets have both been extended several times since their introduction. In particular, the Thumb-2 extension introduces 32-bit instructions into the 16-bit Thumb instruction set providing greatly increased performance without compromising the code density benefits of Thumb.

Of the ARM cores 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-2 instruction set.

ARMv7-A profile cores also support the ThumbEE extension described in section 2.2.3 above.

4. Exceptions and interrupts

ARM supports seven basic exception types. External interrupts are mapped to the FIQ and IRQ exceptions. Other exceptions are used for external errors (e.g. bus errors), internal errors (e.g. undefined instructions or memory address translation faults), or software interrupts (caused synchronously by executing an SVC instruction).

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

ARMv7-A cores with the virtualization extensions implement an additional exception (Hyp) which provides entry into a Hypervisor mode which can be used for implementation of a full software hypervisor environment.

5. Memory architecture

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

All of the current ARMv7-A profile cores provide two-level caching with full virtual memory address translation. In the latest cores, this is extended (in the form of the Large Physical Address Extensions) to provide 40-bit physical addressing (see 2.2.5 above).

Cores implementing the multiprocessing extensions (see 2.2.4 above) implement coherency in the L1 cache across up to four processors in a single multiprocessing cluster.

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 is provided via a combination additional logic within the chip (Embedded Trace Macrocell) 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 cores within a single device.

3 Power Architecture (PPC) and ARM compared

The Power Architecture has a long history and has been under the control of various consortia and companies. There are several standard extensions to the architecture (as well as several non-standard ones). For this document, we will use the Power 2.03 ISA as documented in the Book III-E specification. The term “PPC” will be used to refer to this. For ARM, we will use architecture ARMv7-A as the basis of comparison.

Many of the PPC architectural extensions have equivalent elements either in the standard ARMv7-A profile or in the extensions to it listed in section 2.2 above.

The PPC architecture supports two principle categories: “Server” and “Embedded”. The Server category supports both 32-bit and 64-bit operating modes, both of which support 64-bit registers and address generation. In 32-bit mode, the lower order 32 bits of the resulting address are ignored when accessing memory. The Embedded category also supports both 32-bit and 64-bit modes. However, the majority of instructions in 32-bit mode only use and affect the lower order 32 bits of registers.

In this document, we only consider the 32-bit Embedded version of the architecture (as described in Book III-E) as this has the clearest correspondence with ARMv7.

It is worth noting early on that the PPC architecture is natively big endian, while ARM is natively little endian. It is important to bear this in mind when reading the documentation of the two architectures since bytes within words and also bits within bytes are enumerated in opposite senses.

For instance, the lower 32 bits of a 64-bit register in the PPC architecture are referred to as bits 32 to 63; in the ARM architecture, these would be referred to as bits 0-31. For PPC systems operating in 32-bit mode, the majority of instructions use the lower 32 bits of notionally 64-bit registers. These bits are referred to as bits 32 to 63, with the upper bits usually being “undefined” except in certain special cases (e.g. double-precision floating point operations).

Likewise, ARM refers to the bottom 2 bits of an address as bits 0 and 1; PPC refers to these as bits 62 and 63. Further, PPC continues to use this numbering even in systems which only support 32-bit addressing.

3.1 Programmer’s model

Both PPC and ARM are “RISC” architectures. As such, they have compact instruction sets and large register sets. Both are load-store architectures (i.e. they cannot perform computation directly on the contents of memory locations).

The programmer’s models are therefore somewhat similar since both reflect the RISC approach. However there are some significant differences.

3.1.1 Register set

The PPC architecture supports a comparatively large register set. There are 32 general purpose registers, GPR0-GPR31. In the 32-bit embedded category, all of these are 32-bit registers.

There are also a number of special-purpose registers, including:

LR – 64-bit Link Register

LR holds the return address after a branch instruction and can hold the target address for a Branch Conditional to Link Register instruction.

CTR – 64-bit Count Register

CTR is intended to be used as a general loop counter. CTR can be decremented automatically by some branch instructions and can hold target of Branch Condition

to Count Register instruction. This avoids using one of the GPRs as a loop counter.

SPRG0-9 – 64-bit Software-use SPRs

These have no defined functionality but can be read or written by application software. Some can only be accessed in supervisor mode.

Devices which include the floating point category also support a number of floating point registers. The vector and single processing categories also have a number of additional registers. Since programs which make use of these features will be inherently non-portable at machine level, a porting analysis is somewhat unnecessary.

ARM has a smaller set of 31 general purpose registers, all 32 bits long. 16 of these (designated r0 to r15) are available for use at any one time with the remainder being associated with non-current operating modes. The core automatically switches between available register sets when the operating mode changes (either under program control or on execution of an exception). The value of “banked-out” registers is preserved across mode changes.

The ARM register set is more straightforward in that all registers, with very few exceptions, are fully accessible and behave identically. In particular, the program counter (r15, commonly referred to as pc) is generally accessible in the same way as any other register. This allows many novel uses of instructions which modify or access the pc to control program flow in efficient ways.

The larger register set offered by PPC can reduce register pressure for compiled code but greatly increases the cost of context save/restore operations frequently used either in operating system task switches or in exception entry/exit sequences. The need to use larger fields in instructions to specify register operands also somewhat reduces the flexibility of instructions. ARM instructions, for instance, often support more flexible addressing modes as there are bits available to encode more options.

Although the ARM register set is smaller, techniques like register renaming (in which architectural registers are dynamically mapped in the pipeline to a much larger set of physical registers) greatly increase the scope for code optimizations in ARM systems. The smaller register set can make for faster context switches and more efficient execution of interrupts due to the time-saving in context save and restore operations.

3.1.2 Status registers

In addition to the general purpose register set, both architectures provide status registers holding e.g. ALU status flags, interrupt status and current mode.

The PPC architecture supports a single 32-bit condition register, CR. Unlike the ARM CPSR (Current Program Status Register), CR is divided up into a number of identical 4-bit fields. Each field reflects status information from a different class of instructions. For instance, integer instructions affect CRF0 (the bottom four bits), while floating point instructions affect CRF1 (the next four bits). Comparison instructions can specify which field reflects the result of the comparison.

PPC floating point instructions set status bits in the Fixed Point Exception Register (XER) to indicate exception conditions as a result of floating point operations.

There is also a separate Machine State Register (MSR) which contains a number of configuration and status bits. These are described in the table below, which also lists ARM equivalents where these exist.

ARM, by contrast, has a single Current Program Status Register (CPSR) which holds both ALU status information and processor status (current operating mode etc). CPSR is as follows in the v7-A and v7-R profiles.

In general, only the ALU flags (NZCVQ) are modifiable when executing in user mode. Other fields, with the exception of T and J bits, may be modified directly when in privileged

modes. T and J bits are only changed indirectly by execution of instructions like BX and BXJ.

The following table lists the bits in the PPC status register and their ARM equivalents. Fields or bits not listed are reserved.

PPC			ARM	
Name	Location	Function	Name	Location
CM	MSR[32]	Computation mode (32 or 64-bit)	N/A	
ICM	MSR[33]	Interrupt CM	N/A	
UCLE	MSR[37]	User cache locking enable	N/A	Cache lockdown is only accessible in privileged modes
SPV	MSR[38]	Embedded floating point and Vector availability	CP15 Coprocessor Access Control Register	Access to NEON and VFP can be disabled
WE	MSR[45]	Wait state enable	N/A	ARM enters idle mode via WFI/WFE instructions
CE	MSR[46]	Critical enable	CPSR.IF	ARM interrupts are controlled by CPSR and GIC configuration
EE	MSR[48]	External enable	CPSR.IF	
PR	MSR[49]	Problem state	CPSR.MODE	Mode bits indicate when in privileged mode
FP	MSR[50]	Floating point availability	CP15 Coprocessor Access Control Register	Access to NEON and VFP can be disabled
ME	MSR[51]	Machine Check Enable	N/A	
FE0	MSR[52]	Floating Point Exception Mode 0	N/A	ARM FP exceptions are signaled via the Undefined Instruction exception
DE	MSR[54]	Debug Interrupt Enable	N/A	ARM debug does not function via exceptions (except for monitor mode) – see 3.5 below

PPC			ARM	
Name	Location	Function	Name	Location
FE1	MSR[55]	Floating Point Exception Mode 1	??	See FE0 above
IS	MSR[58]	Instruction Address Space	N/A	The two address spaces in PPC are analogous to ARM's dual page table system
DS	MSR[59]	Data Address Space	N/A	
PMM	MSR[61]	Performance Monitor Mark	N/A	ARM's Performance Monitor does not provide this feature

3.1.3 Instruction set

In the PPC architecture, instructions are classified as Branch, Fixed-Point or Floating-Point. There are additional classifications associated with extensions such as Signal Processing, Vector etc.

Broadly, the Fixed-Point instructions correspond to the set of ARM instructions which carry out ALU operations, together with Load and Store instructions; the Floating-Point instructions correspond to ARM's VFP and NEON instructions.

The following terms are also used when describing PPC architecture instructions:

PPC	ARM
Fixed-Point	Integer
Record	Set condition codes
Algebraic	Signed
Logical	Unsigned (in comparison instructions)

This table shows some example equivalent instructions in the two architectures.

PPC	ARM
<code>addi %r0, %r1, 2</code>	<code>ADD r0, r1, #2</code>
<code>lwzux %r0, %r1, %r2</code>	<code>LDR r0, [r1, r2]!</code>
<code>bdnzlr</code>	<code>BXNE lr</code>

PPC branch instructions always set LR to the address of the next instruction regardless of whether the branch is taken or not. This is done at the time that the branch destination address is calculated. This differs from the ARM behavior, in which the link register (lr) is only set if the branch is taken. PPC branch instructions also include "hint" bits which can

be used by the compiler to indicate the likelihood of the branch being taken. While useful on some processors, higher-end devices (both ARM and PPC) tend to have sophisticated statistical or dynamic branch prediction schemes which renders these bits largely obsolete (standing advice to compilers targeting such devices is to ignore these bits). ARM does not provide the facility for compilers to affect branch prediction behavior on a per-instruction basis in this way - on ARMv7-A class processors, such functionality is unnecessary.

PPC memory access instructions have more restricted addressing modes compared to ARM. The only available addressing modes are:

- register + constant
- constant
- register + register

PPC memory access instructions also have an “auto-update” form but this is restricted to storing the effective address in the base register. There is no post-index auto-update for as in the ARM architecture. ARM also permits the offset to be shifted as part of the addressing mode.

Both architectures provide unsigned and signed versions of load instructions for halfwords and words. PPC refers to these as “load and zero” and “load algebraic”, ARM as “load” and “load signed”.

Both architectures provide instructions for transferring multiple registers to or from a contiguous block of memory. The PPC Load Multiple Word (lwm) and Store Multiple Word (stmw) instructions can transfer up to 32 words into consecutive GPRs starting at an address specified in a base register and increasing in address. This addressing mode, corresponding to the ARM “increment after,” is the only option in the PPC architecture. ARM’s LDM and STM allow decrementing addressing as well and also support the ability to adjust the address before or after each memory access.

Some interesting PPC instructions have no direct ARM equivalent:

- Load String Word (lsw) and Store String Word instructions which copy byte-oriented string data into/from sequences of registers.
- PPC has “Trap” instructions which call system trap handler if a specified condition (e.g. register vs. constant, register vs. register) holds. ARM has no equivalent but the SVC instruction can be conditional.
- PPC has rotate/shift and mask instructions (e.g. rlwinm which rotates a register an arbitrary number of bits left and then masks with a mask generated by specifying start and stop bit positions.) ARM has no direct equivalent of these but v6 of the architecture introduces a powerful set of bitwise operations (e.g. bitfield clear, bitfield extract etc). Also, the ARM barrel shifter can be used inline on the second operand in very flexible ways.

The assemblers for both architectures make extensive use of pseudo instructions to implement operations which would otherwise be difficult. Examples include the PPC li and la instructions (for loading constants or addresses into registers). The ARM equivalents are LDR= and ADR.

Similarly, PPC has no “subtract immediate” instruction. The “subi” mnemonic, however, is permitted and is assembled as an addi instruction with the constant argument negated.

Both architectures provide similar mechanisms for implementing mutual exclusion semaphores. The PPC lwarx/stwx pair correspond to ARM’s LDREX/STREX. See 3.4.6 below for more information.

3.1.4 Operating modes

PPC architecture processors can operate in two modes: “User” and “Supervisor”. User mode is unprivileged, Supervisor mode is privileged. The current mode is controlled via the MSR(PR) bit. The processor automatically switches to Supervisor mode on entry to an exception handler. A program can cause a switch to supervisor mode by executing the sc instruction, which causes a System Call exception.

On a PPC device, privilege affects, among other things, page attributes (execute permission), memory accesses, debug events, access to performance monitor registers, access to SPRs,

An ARM device has 7 basic operating modes. The current mode is encoded in a single field of the CPSR and changing mode in software is generally by directly modifying these bits.

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		
System	Privileged mode using the same registers as User mode	Unprivileged mode	
User	Mode in which most Applications and OS tasks run		

Generally, there is little need for an application to change mode explicitly. The appropriate mode is entered when an exception is handled by the core. In particular, a program executing in User mode gains access to privileged OS features by executing an SVC instruction which causes an automatic switch into Supervisor mode. The transition back to User mode happens automatically on return from the SVC exception handler.

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.

Obviously, the mode bits can only be modified when running in a privileged mode. A User mode program can change to Supervisor mode by executing an SVC instruction (similar to the PPC sc instruction), which causes a Supervisor Call exception.

Note that the above table does not show the Monitor and Hyp modes which are associated with the security and virtualization extensions described in section 2.2 above.

3.1.5 Stack

By convention, both architectures generally implement a full-descending stack. PPC uses GPR1 for the stack pointer, ARM uses r13. This is by convention only in the PPC case as GPR1 has no properties which associated it particularly with stack operations. Although this was also true in earlier versions of the ARM architecture, current versions (ARMv7-A included) associate r13 explicitly with the stack.

As well as not having a dedicated stack pointer, the PPC instruction set does not include any push or pop instructions. Because of this, PPC functions generally create a single, large stackframe on entry and delete it on exit. This stackframe contains statically allocated fields for automatic variables, parameters, return address, callee-saved registers etc. These fields are then accessed using register-offset addressing within the procedure. The Power Embedded Application Binary Interface (Power EABI) lays down the rules for doing this.

ARM provides efficient single and multiple word stack access instructions as shown in the table below.

Operation	PPC	ARM
PUSH single	N/A	STR r2, [sp, #-4]!
POP single	N/A	LDR r2, [sp], #4
PUSH multiple	N/A	STMFD sp!, {r0-r3}
POP multiple	N/A	LDMFD sp!, {r0-r3}

Note that the ARM assembler provides PUSH and POP mnemonics which correspond to the instructions shown above.

3.1.6 Code execution

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

Implementations of the PPC architecture are much less standardized in terms of the micro-architecture than ARM cores. The majority of ARM licensees use the core as implemented by ARM, meaning that performance and execution behavior is essentially the same across a large range of devices. There are a few “architecture licensees” who implement the core using proprietary micro-architectures. This may result in observable differences in timing of individual instructions and instruction sequences.

3.2 System control and configuration registers

The ARM architecture makes use of an internal “coprocessor” for system control and configuration. This implements a set of registers used to control cache, memory systems, branch prediction, clocking etc.

The PPC architecture achieves this using a combination of special purpose registers and dedicated instructions.

For example, to invalidate part of the data cache on a PPC device, you use the dcbi instruction. On an ARM core, you use an MCR instruction to transfer parameters to a specific register in CP15.

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 conserving instruction set space as well.

3.3 Exceptions and interrupts

PPC supports three classes of exception: Base, Critical and Machine Check. Each has its own pair of registers for saving return address (xSRR0) and current processor status (xSRR1). All are handled in supervisor (privileged) mode.

ARM supports seven basic exception types, each with its own associated mode. Each mode provides registers for saving return address (lr) and processor status (spsr). All exception handling modes are privileged.

PPC supports two external interrupt sources: critical and non-critical. Both can be masked in the MSR. On handling a non-critical external interrupt, the processor automatically saves the return address in SRR0 and the current MSR in SRR1 (critical interrupts use CSRR0 and CSRR1). Both registers are restored on exit when an rfi instruction is executed at the end of the handler routine.

ARM also supports two external interrupt sources: FIQ and IRQ. FIQ has higher priority than IRQ and has extra banked registers to improve latency. Both can be masked via the I and F bits in CPSR.

Both architectures also support a number of internal interrupt courses e.g. memory permission violations, alignment errors, undefined instructions etc.

Since the interrupt classifications are heavily overloaded in the PPC architecture, an Exception Syndrome Register (ESR) is provided to differentiate between the several possible causes of each exception type.

Both architectures permit the use of an external interrupt controller to provide greater control over multiple hardware interrupt sources. In the case of ARM, the standard Generic Interrupt Controller architecture is fairly well standardized in the case of ARMv7-A class devices.

3.3.1 Interrupt prioritization and pre-emption

In both architectures, the relative priority of all defined interrupt types is architecturally fixed.

However, most systems have many interrupt sources connected to a single external interrupt signal (in the case of ARM, IRQ, for PPC “External Interrupt”). There is clearly a need to prioritize between these external sources.

For this reason, many systems of both architectures implement external interrupt controller hardware which is used to provide finer-grained prioritization and pre-emption control across a large number of external interrupt sources.

ARM has defined the “Generic Interrupt Controller” architecture which is used by the majority of ARMv7-A devices. This ensures a high degree of compatibility between ARM platforms.

The PPC architecture does not define a standard interrupt controller architecture, leading to variation in implemented systems. Many devices use standards such as the OpenPIC architecture.

3.3.2 External interrupts

ARM supports two external interrupt sources. They are vectored separately and FIQ has a higher priority than IRQ. There are some further optimizations to the FIQ mechanism which makes it faster than IRQ (extra banked registers, for example). The two can be enabled and disabled separately via the I and F bits in CPSR. There is no global interrupt enable flag.

PPC also supports two external interrupt sources “External Interrupt” and “Critical Input”. The relative priority is fixed, as you would expect, so that critical interrupts have a higher priority than external interrupts and will pre-empt them.

3.3.3 Internal interrupts and exceptions

Both cores support a range of internal exception types. ARM separates out many of these into separate vectors in the vector table, making determination of the cause simpler. The PPC vector table is more specific, having separate entries for most exception types. The

ESR (Exception Syndrome Register) mentioned above is provided to assist software in discriminating between the different exception types.

Note that PPC treats debug watchpoint triggers as an exception event. ARM handles these in the debug logic transparently to the programmer (except in “Monitor mode” debugging, in which case debug interrupts are overlaid on the Abort exception)

3.3.4 Vector table

The Embedded version of the Power architecture (Book E) does not define a “traditional” vector table. Instead, a base register and a set of offset registers are used to define a set of addresses at which exception handlers are located:

IVPR Interrupt Vector Prefix Register

This provides the upper 16 bits of the vector address for all exceptions.

IVOR Interrupt Vector Offset Register

There is one of these for all 64 possible exception vectors. The address of the handler is generated by adding the 16-bit offset value in the appropriate IVOR to the 16-bit base value in the IVPR

In this way, the exception vectors can be placed anywhere in memory and the need to vector indirectly through a table is avoided. However, the least significant four bits of the offset register are fixed at zero. This means that the individual vectors are spaced at least 16 bytes apart in memory, making the vector table a minimum of 1KB in size if the full 64 possible exception types are used. Obviously, if a particular handler is small enough to be written within the available 16 bytes, it can be wholly contained within the table but this will not be the case for the majority of exceptions. In these cases, the vector table entry contains a jump instruction to the start of the appropriate exception handler.

The ARM architecture defines a vector table containing a single entry for each of the seven defined exception types. This table is located by default at 0x0000:0000. It can be relocated (either under hardware control at reset or under subsequent software control) to an alternate address at 0xFFFF:0000.

Each entry in the table (see below) is a single executable ARM instruction which causes a branch to the relevant exception handler. The size of the ARM vector table is fixed at 8 words. In the majority of systems, the vector table entry will contain a load instruction which loads the address of the exception handler into the program counter.

Address	Exception Vector
0x0000:0000	Reset
0x0000:0004	Undefined Instruction
0x0000:0008	SVC (previously called Software Interrupt or SWI)
0x0000:000C	Prefetch Abort
0x0000:0010	Data Abort
0x0000:0014	Reserved (used for Hyp in the virtualization extensions)
0x0000:0018	IRQ
0x0000:001C	FIQ

3.3.5 Interrupt handlers

Exception handlers are very similar in the two architectures. Both are responsible for any software prioritization, saving and restoring context, interrupt dispatch and return to the interrupted code.

In both architectures, there are similar actions to be taken when re-enabling interrupts to allow nesting. In particular, the registers used to save the interrupted context need to be saved in order to provide a re-entrant interrupt handler.

3.4 Memory

3.4.1 Memory map

Neither the ARM nor the PPC architecture specifies a fixed memory map. However, individual implementers have significant freedom to define device-specific memory maps. Programmers need to refer to the documentation for the particular device concerned to find out which types of memory are provided, how much of each is present and where in the memory map it is located. Frequently, the location of different types of memory may be configured in software.

The ARM architecture allows access to a linear 4GB address space with the only fixed element being the exception vector table. This is located at physical address 0x0000:0000 but, as mentioned above, can be relocated to 0xFFFF:0000 either via hardware configuration at reset or under software control at runtime.

3.4.2 Virtual memory

Both architectures include support for virtual-to-physical address translation. There are some important differences which are described here.

The ARM Virtual Memory System Architecture implemented in ARMv7-A devices (VMSAv7) translates “virtual addresses” issued by the core into “physical addresses” used to access memory. The PPC architecture translates “effective addresses” issued by the processor into “real addresses” used to access memory.

In both cases, both virtual (effective) and physical (real) addresses are 32 bits, giving a 4GB physical address space. Some implementations of the PPC architecture support 36-bit addressing, allowing access to 64GB of physical memory space. Both systems support a variety of page sizes with the minimum page size being 4KB in both cases.

The translation mechanism is very different between the two architectures.

PPC defines a set of Translation Lookaside Buffers (TLBs) which contain the translation information in current use. Each entry in the TLB corresponds to a particular range of effective addresses and contains the information required to generate the corresponding real address, together with access control configuration. Software, typically in the OS, is required to manage the entries which are held within the TLB at any one time. If an effective address is used which does not match a current TLB entry, a TLB Miss exception is raised and the handler needs to install a new entry (this will usually involve deleting an existing entry) and then return to retry the translation.

ARM holds all this configuration in page tables in external memory. The MMU contains hardware which can access these tables automatically to locate and apply translations when virtual addresses are issued. The ARM architecture also allows for TLBs but they are used as caches for recently used translations and very little software management is required for them.

From a software point of view, the ARM scheme is easier to use since it requires considerably less software support. Once the page tables have been constructed, typically during system initialization, they need only be modified on events like context switches.

In both systems, the TLB configuration incorporates extra bits for identifying running processes and reducing the impact of context switches in an operating system environment.

Both architectures support a two-stage translation process. The PPC architecture achieves this via two hierarchical levels of TLB entries, the first supporting a range of page sizes, the second only supporting 4KB pages. The architecture does not mandate whether either level is unified or separated for instruction and data spaces. ARM provides a single translation scheme which includes up to two levels of page tables. The first level support translation at 1MB granularity, the second at 4KB granularity. If the device supports the LPAE extension to ARMv7-A, a complete third level of translation is available providing access to 40 bits of external physical address space.

The PPC architecture allows accesses to be separately translated in two separate “address spaces”. Which space is used is determined by the current value of the IS (for instruction accesses) and DS (for data accesses) bits in the MSR and these must match the corresponding bits in the TLB entry for the translation to succeed. ARM, instead, provides the ability to define two completely separate translation tables, switching between the two by changing a single CP15 register – this allows, for instance, OS and applications to use separate set of page tables without the need to rewrite large pieces of the tables.

3.4.3 Memory access control

In both architectures, memory access control is tightly bound up with the virtual memory system. In particular, memory protection attributes associated with each memory region are largely contained within the translation tables (or, in the case of PPC, the TLBs).

Both architectures provide facilities for read-only, read-write and execute permissions. Permissions can also be policed separately for user and privileged mode accesses.

Individual memory regions can also have cacheability and bufferability attributes. These are defined within the TLB in the PPC architecture and within the page tables in the ARM architecture.

3.4.4 Access types, endianness and alignment

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

Both architectures support big-endian and little-endian systems. System endianness in both cases is determined via a signal sensed at reset and can subsequently be changed under software control.

Note that ARM is natively little-endian, while PPC is natively big-endian. In many cases, this is more of a documentation issue than anything else but care must obviously be taken when porting involves a change of endianness. In particular, when configured as little-endian, some PPC systems behave as little-endian from a software point of view but actually store data in big-endian format in memory. This may need careful attention if this data is shared with external systems.

Both architectures have rules governing alignment of data in memory. Essentially, data items should be aligned on natural boundaries.

ARMv7-A cores and PPC cores are capable of accessing data which is not naturally aligned but there may be a performance penalty associated with this due to the need to carry out multiple bus transactions.

In both architectures, instructions are always 4 bytes long and must be word aligned.

One key difference is that the PPC architecture allows endianness to be defined on a per-page basis in the virtual memory system configuration. ARM allows data-side endianness to be changed on the fly.

3.4.5 Atomicity

Both architectures define that only byte, aligned halfword and aligned word accesses are guaranteed atomic. Other accesses (e.g. an access to an unaligned word) should be regarded as non-atomic in the sense that part of the memory content may be accessed before the instruction is abandoned and then this access will be repeated or restarted.

3.4.6 Barriers and synchronization

There are cases in program execution where it is necessary or desirable to ensure that certain classes of memory access are completed in a certain order.

The ARM system of memory typing (in which memory is defined as “Normal”, “Device” or “Strongly Ordered”) ensures that this is the case in the vast majority of circumstances. However, there may be cases where the program need to explicitly ensure ordering.

The following table indicates the ARM instructions corresponding to the barrier instructions defined by the PPC architecture.

PPC	ARM
isync	IMB Instruction Memory Barrier
sync (msync/hwsync)	DMB Data Memory Barrier
lwsync	DMB Data Memory Barrier
ptesync	DSB Data Synchronization Barrier (when changing MMU context, in which case note that an IMB may also be required)
eieio	N/A The default rules for Device memory are generally sufficient, though a DMB may be required in some circumstances e.g. to enforce ordering between device accesses which are in different regions of Device memory.
mbar (identical to the eieio instruction in earlier PPC architectures)	N/A Though see the notes for eieio above.

As stated in the table, the PPC **eieio** instruction can usually be omitted in corresponding ARM code. **eieio** is used primarily for enforcing access ordering when dealing with memory-mapped devices and, on ARM systems, the rules associated with Device memory are generally sufficient to remove the need for a barrier in these circumstances. However, you should examine carefully any cases where **eieio** has been used in cacheable memory on PPC systems. In this situation, a **DMB** may be required in the equivalent ARM code.

Additionally, both architectures define sets of instructions which can be used to implement common synchronization or exclusion sequences.

PPC	ARM
lwarcx Load word and reserve	LDREX Load Exclusive
stwcx Store word conditional	STREX Store Exclusive
N/A	CLREX Clear Exclusive

The fact that the PPC architecture does not have an equivalent to the ARM CLREX instruction means that a “dummy” store conditional instruction is required in a context switch in a PPC system. This is replaced with a CLREX instruction when moving to ARM. However, note that some other PPC instructions or operations also cause the lock to be released (e.g. data cache flush via the dcbf instruction). In these cases, it may be necessary to add CLREX instructions as the ARM equivalent cache operation may not release the lock as a side-effect.

For further information, see the ARM document GENC 007826 listed in the references.

3.4.7 Shared memory

Both architectures support the concept of memory regions which are shared between multiple processors or agents. Both support a memory model which is “weakly-ordered” and this requires barriers or other synchronization constructs at certain points to ensure ordering when necessary.

ARM, additionally, 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).

3.4.8 Caches

Both architectures support similar cache configurations: Harvard L1 caches, backed by an optional unified L2 cache. Neither architecture mandates any particular cache structure in terms of line length, associativity etc.

It is good practice to check whether the particular device you are using ensures that cache contents are invalidated following reset. Cortex-A8 and Cortex-A9, for instance, differ on this. Cortex-A8 automatically invalidates L1 and L2 caches on reset (though this behavior can be disabled via a hardware signal sensed during the reset sequence); Cortex-A9 does not do this meaning that the caches need to be manually invalidated by software during the reset sequence, if this is necessary.

Be careful, also, with the indexing and tagging methods used for caches. Whether virtual or physical indexes and tags make a difference from a software point of view depends crucially on the relationship between page and cache sizes. This differs from system to system and needs to be checked carefully when porting.

In all cases, cache maintenance operations will need to be translated from one architecture to the other.

In general, PPC operations correspond to ARM as shown in the table.

PPC	ARM	Mnemonic
	Invalid Inner Shareable	ICIALLUIS

PPC	ARM	Mnemonic
	Instruction cache to PoU	
	Invalidate Branch Predictor array Inner Shareable	BPIALLIS
	Invalidate all instruction caches to PoU	ICIALLU
icbi	Invalidate I cache by MVA to PoU	ICIMVAU
N/A	Invalidate Branch Predictor array	BPIALL
N/A	Invalidate Branch Predictor array by MVA	BPIMVA
dcbi	Invalidate data/unified cache by MVA to PoC	DCIMVAC
N/A	Invalidate data/unified cache by set/way	DCISW
dcbf (Cleaning as part of invalidation is implementation-defined)	Clean data/unified cache line by MVA to PoC	DCCMVAC
N/A	Clean data/unified cache line by set/way	DCCSW
dcbf1 (Cleaning as part of invalidation is implementation-defined)	Clean data/unified cache line to PoU by MVA	DCCMVAU
dcbf (Cleaning as part of invalidation is implementation-defined)	Clean and Invalidate data/unified cache line by MVA to PoC	DCCIMVAC
N/A	Clean and Invalidate data/unified cache line by set/way	DCCISW

It is clear that the set of cache maintenance operations provided by ARM is more comprehensive and more flexible. ARM, however, provides no matching instructions for `dcbz` and `dcbst`. `PLI` and `PLD` perform similar functions to the PPC `icbt` and `dcbt` instructions.

“Point of Unification (PoU)” and “Point of Coherency (PoC)” are terms which have specific meaning when referred to shared memory within ARM systems. Refer to the ARM Architecture Reference Manual for the precise definitions of these terms.

3.5 Debug

Both architectures provide for debug over the standard JTAG connections. The underlying implementation, however, is rather different.

PPC devices support a dedicated debug interrupt and a set of debug status/configuration registers (within the set of SPRs). These are used by external debug agents to provide

debug functions. The processor operates in “Internal Debug Mode” to handle a debug exception and a resident monitor provides debug functions. It is implementation-defined whether any or all of these debug facilities are available to external agents via a separate interface (e.g. JTAG).

ARM CPUs support a debug “state” in which the core is halted and isolated from the rest of the system. The core can then be controlled from the external system via some on-chip logic (EmbeddedICE). ARM terms this “halt mode” debugging – ARM chips also support an alternative, “monitor” mode debugging, in which debug commands are handled by a small resident monitor. This allows software to be debugged without halting the system. ARM debug functions are available via the JTAG port.

Both architectures provide features which support instruction and data breakpoints and program trace.

To assist with performance benchmarking, both cores incorporate a range of configurable counters which can be used to capture data in a non-intrusive manner.

Note that the PPC architecture allows the debug interrupt to be disabled via the MSR. ARM does not allow this, except in the case of disabling debug of secure applications when using the TrustZone security extensions.

3.6 Power management

Devices based on both PPC and ARM devices support a variety of power-saving modes. In each case, the exact degree of power saving available and the behavior in each power configuration is very dependent on the implementation of the core itself and the surrounding logic. Programmers should check the documentation for the target device to determine exact behavior.

PPC devices support the following power states:

Full on	Device is operating normally at full clock speed.
Doze	Instruction fetch and execution are suspended. The core is still clocked. This is a precursor to the “stop” state mentioned below. Doze state is entered via the “halt” signal.
Nap	Core internal clocks are stopped but timebase clock is still running. Nap state is entered via the “stop” signal.
Sleep	Core internal clocks and timebase are stopped.

These states are controlled via hardware inputs signals to the core. These are controlled via a combination of the MSW[WE] bit and bits within the H1D0 register. In all cases, interaction with external logic is required to manage the transition between the states and also to control any power management in the external system.

ARM devices are more standardized. In general, you will find support for a number of power-saving modes, offering progressively greater reductions in power consumption. If we take the Cortex-A9 as an example, the following modes are supported.

Full Run Mode	This is the normal mode of operation. The core, memories and supporting logic are fully powered and running at full clock speed. When in Full Run mode, components like Neon or VFP may or may not be powered. Clearly, they should be powered down when not actively required to minimize unnecessary power use.
WFI/WFE	This mode is entered following execution of a WFI or WFE instruction. The majority of the core clocks are stopped, reducing power usage to static leakage current only. Since the core is still powered, state is retained fully. The core will return to Full Run mode on an interrupt, debug request, or reset. In the case of WFE mode, assertion of the EVENTI signal will also wake the processor up.

Dormant In Dormant mode, RAMs and caches remain powered while the core itself is completely shut down and powered off. Prior to entry to this state, all core registers must be saved to external memory. Exit is via reset and the reset handler is responsible for restoring the saved registers and system state.

Shutdown In Shutdown mode, the core, caches and all memories are completely shutdown. Exit is via reset.

Dormant and Shutdown modes offer the greatest power saving but require an external power controller to manage entry and exit. This must be implemented at the time the chip is designed.

The external power controller is also responsible for managing the state of other components on the chip.

Other ARM cores support a similar range of power saving modes.

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. To a lesser extent, assembly code may be used to obtain higher performance.

4.1 General considerations

4.1.1 Operating mode

A stand-alone application will most likely execute in a privileged mode at all times in both systems – supervisor mode on PPC 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 an exception) will be automatic.

In an operating system environment, PPC applications will run in user mode while the operating system executes in supervisor mode. Similarly, 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 software interrupt (sc in PPC, SVC in ARM). The transitions back to user mode will happen automatically on return from the resulting exception.

4.1.2 Stack configuration

Because each mode has its own stack pointer, an ARM core in effect has several stacks. These must each be initialized during system startup following reset, certainly before interrupts are enabled. Stack pointers are generally initialized to reserved areas of memory during the initialization sequence.

A PPC system usually only has one stack, with exceptions saving context to dedicated registers or to pre-defined areas of a specially-created stack frame, so only the system stack requires initialization.

4.1.3 Memory map

Unless the system makes use of a particularly complex memory map, the default options in the build tools on the target ARM system will normally suffice. If your system has an MMU or MPU and you wish to make use of it, it will need to be configured. If you do not wish to use it, simply leave it alone.

4.1.4 Code and data placement

Both systems provide a flat memory map. Individual memory devices and individual items of code, data and peripherals may be placed almost anywhere within the available address space consistent with the layout of physical memory devices provided by the chip designer.

1. Code

The ARM memory map allows code to be placed anywhere where there is suitable memory. Most systems will benefit, in terms of performance, if code is relocated to RAM during startup. The scatter loading feature of the ARM development tools makes this relocation essentially automatic and it is specified completely at build time. In ARM systems, code memory is defined as “Normal” memory for maximum performance. Instruction memory should always be cacheable.

2. Data

In an ARM system, data memory can be placed anywhere, allowing data to be placed according to space or performance requirements. The scatter-loading scheme used by the ARM development tools makes it easy to make the correspondence between initial values (held in ROM) and initialized data segments (established and initialized in RAM at startup). Data memory in an ARM system is usually defined as “Normal” memory for maximum performance. It is usually also cacheable and bufferable.

3. Peripherals

In an ARM system, peripherals can be allocated anywhere in the memory map within the constraints of the chip design. Peripheral regions should be configured as Device memory, uncached and unbuffered. The scatter control file also allows you very easily to specify that these areas are not initialized automatically during startup (using the UNINIT directive).

4.1.5 Data types and alignment

Various standards exist for data types in the PPC architecture. The following uses those defined for the e500 as implemented by Freescale. This specification is broadly compliant with the System V specification for PowerPC.

Type	ARM	PPC	Notes
char	8-bit signed	8-bit unsigned	--unsigned_chars will change the ARM default if required
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	IEEE single-precision
double	64-bit	64-bit	IEEE double-precision
long double	64-bit	128-bit	128-bit support is optional on PPC
pointer	32-bit	32-bit	

All basic integer types are signed by default in both architectures, with the exception of the char type which is unsigned in the e500.

The major difference here when porting code is the sign of the 8-bit char type. When porting to ARM, the problem can be ignored if necessary by switching the default using the `--unsigned_chars` compiler switch.

4.1.6 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, the tools conform to the ARM Executable Application Binary Interface (EABI). The ARM Architecture Procedure Call Standard (AAPCS) is part of this. Documentation on this can be found on ARM's website.

4.2 Tools configuration

Several compiler toolchains exist which support both ARM and PPC architectures. Vendors such as Greenhills and CodeSourcery, for instance, sell such products.

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 PPC 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 RealView tools. Clearly, more significant reconfiguration will be required here and you should refer to the RealView documentation (all available on ARM's website) for further information on this.

More information on support for GNU tools can be found here:

<http://www.arm.com/community/software-enablement/linux.php>

4.3 Operating system

Many PPC-based systems use the Linux operating system. There are several versions of this available for the ARM architecture. Details of Linux support on ARM can be found here:

<http://www.arm.com/community/software-enablement/linux.php>

4.4 Startup

Both processors will exit reset with in privileged mode with interrupts disabled.

ARM processors will reset with the MMU disabled, giving a flat virtual-physical address mapping with caches disabled. PPC processors will boot with a single preset TLB entry which maps the top 4KB of the effective address range to the top 4KB of the real address range.

Reset software should carry out the following:

1. Invalidate instruction and data caches (if the implementation requires it)
2. Perform any memory re-mapping required
3. Initialize system memory for operating system or application code
4. Initialize the system stack pointer
5. Initialize the exception vector table
6. Initialize other processor registers as required
7. Initialize on-chip and off-chip peripherals
8. Transfer control to the operating system or application

The operating system initialization code should ensure that at least the following are carried out:

1. Enable branch prediction
2. Initialize exception and application stack pointer
3. Initialize page tables, enable caches and MMU
4. Execute main program

4.5 Handling interrupts and exceptions

4.5.1 Writing interrupt handlers

An ARM system usually requires at least two interrupt handlers: IRQ and FIQ.

Typically, a single interrupt source (that with the lowest latency requirement) is connected to FIQ. FIQ interrupts are not normally nested. At least a simple top-level FIQ handler must be written in assembler though this will usually call out to a C function to handle the interrupt before returning, via the assembler, to the interrupted application.

The ARM tools support writing simple (i.e. non-reentrant) IRQ handlers directly in C. For simple systems, this is a useful feature. However, most systems require re-entrant interrupts. This requires a small top-level IRQ handler in assembler which then calls out to a C function to dispatch to the correct handler before return, via the assembler, to the interrupted application.

The implementation of the assembler part of the handler is shown below. Note that this code will not be suitable for architectures prior to ARMv6.

Non-nested IRQ handler (v6 and later)	Nested IRQ handler (v6 and later)
<pre> IRQ_Handler PUSH {r0-r3, r12, lr} BL identify_and_clear_source BL C_irq_handler POP {r0-r3, r12, lr} SUBS pc, lr, #4 </pre>	<pre> IRQ_Handler SUB lr, lr, #4 SRSFD sp!, #0x1f CPS #0x1f PUSH {r0-r3, r12} AND r1, sp, #4 SUB sp, sp, r1 PUSH {r1, lr} BL identify_and_clear_source CPSIE i BL C_irq_handler CPSID i POP {r1, lr} ADD sp, sp, r1 POP {r0-r3, r12} RFEFD sp! </pre>

The nested IRQ handler re-enables IRQ interrupts (using the CPSIE instruction). However, before doing this, to prevent corruption of the return address, it must change to

System mode (the first CPS instruction). All registers (including the return address and SPSR) are saved on the System mode stack (this is shared with User mode).

The ARM ABI requires (as does the Power ABI) that the stack pointer is doubleword-aligned at public boundaries so exception handlers must check and, if necessary, correct the alignment of the stack prior to making any external function calls.

4.5.2 Vector table generation

In both architectures, the vector table consists of executable instructions (rather than the addresses found in many other systems). As explained in 3.3.4 above, the PPC vector table entries are spaced out at 16-byte intervals, offering the possibility of writing very simple handlers (or, at least, the first part of more complex ones) in place in the vector table itself. ARM, having a contiguous list of word-sized entries, offers this possibility only in the case of FIQ. The vector for FIQ is the last in the table, so the handler for FIQ can be located to start at this address and run contiguously from that point.

The ARM vector table is shown in section 3.3.4 above.

Each entry in the vector table typically consists of a branch instruction pointing to the start of the relevant exception handler. For a “small” application (i.e. one in which the exception handlers are within direct branch range of the vector table), branch instructions can be used.

```

AREA Vectors, CODE, READONLY

IMPORT Reset_Handler
; import other exception handlers

; ...

ENTRY
start
    B      Reset_Handler
    B      Undefined_Handler
    B      SWI_Handler
    B      Prefetch_Handler
    B      Data_Handler
    NOP    ; Reserved vector
    B      IRQ_Handler

    ; FIQ_Handler will follow directly

END

```

As shown, the vector table is typically placed in a named section of its own so that it can be explicitly located at link-time.

For larger systems, it is common to use a direct load to the program counter in each case and to place the handler addresses in a small data table near the handler. This method has other advantages – in particular it makes it much easier to modify the vectors at run time by simply re-writing the data table.

```

AREA Vectors, CODE, READONLY

IMPORT Reset_Handler
; import other exception handlers

; ...

ENTRY

start

    LDR    pc, Reset_Vector
    LDR    pc, Undefined_Vector
    LDR    pc, SWI_vector
    LDR    pc, Prefetch_Vector
    LDR    pc, Data_Vector
    NOP
    LDR    pc, IRQ_vector

; FIQ_Handler can follow directly

Reset_Vector      DCD    Reset_Handler
Undefined_Vector  DCD    Undefined_Handler
; etc

END

```

In this scheme, the FIQ exception handler may still run from address 0x0000.001C but it is restricted to smaller than 4KB since the table of vector addresses cannot be further than 4KB from the vector table (this is limited by the range of the constant offset in the LDR instructions).

If your ARM system incorporates a Vectored Interrupt Controller (the Generic Interrupt Controller, for example), the IRQ entry in the vector table is not used. Instead the core is provided with the address of the interrupt handler for the active interrupt automatically by the interrupt controller and branches directly to that routine. This routine must still be written following the IRQ handler conventions regarding context saving, register usage, re-entrancy and return instruction.

4.5.3 Interrupt configuration

In an ARM system, interrupts (external physical events) and exceptions (internal events) are configured slightly differently. Exceptions (i.e. Undefined Instruction, SVC, Prefetch Abort, Data Abort) are always enabled (though imprecise aborts can be disabled on ARMv7 cores). IRQ and FIQ must be enabled by clearing the I and F bits respectively in CPSR.

If an interrupt controller (e.g. ARM's Generic Interrupt Controller) is being used on an ARM system, the configuration will be required to at least enable and prioritize the external interrupt sources before they can be used.

4.6 Timing and delays

ARM cores do not provide a standard timer function as part of the architecture (ARMv7-M cores do provide the SysTick peripheral). However, almost all ARM-based devices will include a generic timer facility. Some ARMv7-A cores, particularly those implementing the multi-processing extensions, do provide a system timer as part of the interrupt controller – this time is common to all the cores in a cluster.

4.7 Power Management

The power management options in an ARM-based device are likely to be more varied and more standardized than those available with a PPC device. See section 3.6 above for a

more detailed description of the power management facilities provided by a typical ARM core.

When using an operating system or real-time scheduler, it is likely that the power management features will have been built into the kernel. In this case, it is simply a case of ensuring that your user processes make it clear to the scheduler when they are idle. How you do this will be system-dependent.

When writing a bare metal application, you will have to 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 Semaphores etc.

The PPC architecture provides the lwar and stwc instructions for construction of semaphores, mutexes etc. The ARM LDREX and STREX instructions operate in a very similar way. Both revolve around the concept of a load instruction which “reserves” a location for a subsequent store instruction which only succeeds if there has been no intervening accesses to the location in question.

The major difference is in the means of checking for success of the store instruction.

The PPC stwc instruction sets the SO bit in CR0 to indicate success or failure. This bit can be tested after the instruction. The ARM STREX instruction places its success/failure status in a register supplied as an argument to the instruction.

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

PPC	ARM
lock: lwarx r5,0,r3 cmpwi r5,0 bne- exit stwcx. r4,0,r3 bne- lock isync exit: ...	lock LDREX r1, [r0] CMP r1, #LOCKED BEQ lock_mutex MOV r1, #LOCKED STREX r2, r1, [r0] CMP r2, #0x0 BNE lock_mutex DMB
unlock: stw r7,data1(r9) mbar 0 stw r4,lock(r3)	Unlock DMB MOV r1, #UNLOCKED STR r1, [r0]

Both mechanisms rely on some logic within the memory system for recording the fact that a reservation exists on a particular memory location. The memory system is not required to record this at byte or word granularity. Instead, both architectures define the concept of the “reservation granule”. It is important to check on the size of the granule when porting as it may be different on the target system – this may result in erroneous operation of lock constructs when the addresses used are within the boundaries of a single granule.

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.

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:

- Some restrictions on instructions which can be used e.g. LDM/STM should be avoided.
- Mark peripheral memory regions as Device memory to ensure access ordering is observed correctly.

The PPC architecture, as well as allowing memory-mapped peripherals, also defines an arbitrary number of special purpose registers (DCRs) which can be used to manage peripherals. The number and functionality of these is not defined by the architecture but by the chip designer. They are accessed using special instructions (mfdcrux and mtdcrux). ARM has no equivalent to this and any ARM-powered devices will have been implemented using the standard memory-mapped peripheral mechanism. Any driver software which access PPC peripherals using these special purpose registers will need rewriting.

4.10 C programming

In general, the similarities between the two architectures will mean that C code which has been optimized for PPC platforms should perform well when re-compiled for ARM.

Clearly any inline assembler or architecturally-specific intrinsic functions will need to be removed, replaced or rewritten. Similarly, any cache or TLB maintenance code will need to be rewritten. Recall that TLB management in the two architectures is significantly different: in PPC systems, address translation depends on run-time support code to populate the TLB entries; in ARM systems, the page tables are initialized at startup and the MMU then reads these at run-time automatically.

Both architectures have similarly strict rules on data alignment. However, ARM cores supporting architecture v6 and later are capable of supporting unaligned accesses in hardware. In Cortex-A cores, this feature is permanently enabled; on earlier cores which support backwards compatibility the feature defaults to disabled and can be enabled, if required, by setting the U bit in CP15 register c1.

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

5 A porting checklist

The following list of points may prove useful when porting source code from PPC to ARM.

- Recompile C/C++ code using tools which target the ARM architecture. In general, code which has been optimized for the PPC architecture should recompile well when targeted for ARM.
- Identify any instances in the source code where it is important that 8-bit variables are unsigned. Either redeclare individual instances to be explicitly unsigned or use the “—unsigned_chars” compilation switch to reconfigure the entire ARM toolchain.
- Locate all accesses to system registers, system calls, platform-dependent driver calls etc and ensure that they are replaced with ARM-dependent or platform-dependent equivalents.
- 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.
- Identify any code which is, or may be, endian-dependent. Static analyzers (such as “Sparse”) can help with this. ARM devices do not support the per-page endianness configuration available on PPC. Any code which relies on this will need to be rewritten very carefully, making use of the ARM byte reversal instructions and data endianness configuration bit in the CPSR.
- 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.
- Any code which makes use of the extended 36-bit addressing supported by some PPC devices will need careful investigation.
- 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 (e.g. switching Altivec for NEON). In these cases, the implications will largely be dealt with by changing drivers and compilation tools.
- PPC code which makes use of the hypervisor and virtualization extensions available in more recent implementations will need to be recoded. If possible, you should target an ARM platform (e.g. Cortex-A15) which provides similar hardware features to support this.
- Identify all uses of memory barriers and synchronization instructions in PPC source code and ensure that they are replaced with the ARM equivalents. Also examine carefully any code may rely on the barrier side-effects of e.g. TLB management operations. It may be necessary to insert additional explicit barrier instructions 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.
- Any PPC code which makes use of 128-bit floating point variables will need to be examined closely. As a minimum, you will need to ensure that a suitable library is

in place to handle this as there is no support for this in hardware on ARM platforms.