Application Note 274

Migrating from IA-32 to ARM

Document number: ARM DAI 0274

Issued: October 2011

Copyright ARM Limited 2011

ARM

Application Note 274
Migrating from IA-32 to ARM

Copyright © 2011 ARM Limited. All rights reserved.

Release information

The following changes have been made to this Application Note.

Change history

Date	Issue	Change
August 2011	A	First release
October 2011	B	Minor revisions

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

The purpose of this document is to highlight areas of interest for those involved in migrating software applications from IA-32 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 IA-32 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 ARM 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 0406B

Cortex-A15 Technical Reference Manual, ARM DDI 0438A

Cortex-A9 Technical Reference Manual, ARM DDI 0388F

Cortex-A8 Technical Reference Manual, ARM DDI 0344K

Cortex-A5 Technical Reference Manual, ARM DDI 0433B

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

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

ARM Compiler Toolchain Using the Assembler, ARM DUI 0473D

ARM Compiler Toolchain Assembler Reference, ARM DUI 0489D

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 GNU Tools and Libraries, ARM DAI 0212A

Barrier Litmus Tests and Cookbook, ARM GENC 007826

We are also aware of a training course from Mindshare Inc,

"Comprehensive ARM Architecture with x86 Comparisons"

You can find more details at www.mindshare.com/arm

(Please note that ARM is not responsible for the content of this training course.)

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 IA-32, 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 IA-32 architecture is different. There, a "word" refers to a 16-bit quantity, "dword" to a 32-bit quantity.
>
> This document uses the ARM terminology exclusively, in which a word refers to a data size of 32-bits.

Data size (bits)	ARM	IA-32
8	byte	byte
16	halfword	word
32	word	dword
64	doubleword	quadword

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-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-A15 processor.

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 Coretx-A15 processor provides 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 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-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, thus allowing implementation of an efficient hardware-assisted virtualization solution.

These extensions are supported by 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 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.

User/System	Exception Modes						
	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					Security Extensions Only	Virtualization Extensions Only
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

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 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-A15 implements 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 IA-32 and ARM compared

The IA-32 architecture, sometimes also referred to as x86-32, has a long heritage which can be traced back to the earliest implementations of the 8086 processor from Intel in 1978. The 8086 itself was launched as a 16-bit extension of the earlier 8080. Although not officially binary compatible with its predecessor, the migration path was deliberately simple.

Over the intervening period, the architecture has been extended several times, almost always with backwards compatibility. The need to maintain backwards compatibility has often been cited as a drawback in the IA-32 evolution as it necessitates maintaining the ability to handle a variety of different word and instruction sizes in one processor core. The architecture supported full 32-bit operation with the introduction of the 80386 in 1985.

The physical address space was extended to 36 bits with the Pentium Pro in 1995. The architecture has since been extended with several 64-bit elements. 64-bit versions of the architecture are not discussed in this document.

The architecture has been implemented in processors from a number of manufacturers besides Intel, notably AMD, VIA, Cyrix and others.

The Atom E6xx series support the IA-32 architecture, with the following extensions:

- Intel Virtualization Technology

- Intel Hyper-Threading Technology

- Enhanced Intel Speedstep Technology

- Deep Power Down Technology

- Intel Streaming SIMD Extensions 2 and 3 (SSE2 and SSE3) and Supplementation Streaming SIMD Extensions 3 (SSSE3)

The discussion in this document uses the Intel Atom E6xx series of devices as a reference point for devices supporting the IA-32 architecture.

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

Unlike ARM, which has a set of 16 identical registers (of which 12 are truly general purpose and three are treated specially by some instructions), the IA-32 architecture has several sets of registers, many with specific purposes.

The table below lists the eight general-purpose registers, together with their sizes and functions. These are used to store register variables and to address items in memory (usually in conjunction with the segment registers described later). Note that these registers are, in many cases, not truly "general-purpose" as many instructions will only work with specific registers or combinations of registers.

Name	Size	Use
EAX	32-bit	Accumulator
EBX	32-bit	Pointer to data in DS segment
ECX	32-bit	Counter for string and loop operations
EDX	32-bit	I/O pointer
ESI	32-bit	Pointer to data in DS segment, source for string operations
EDI	32-bit	Pointer to data in ES segment, destination for string operations
ESP	32-bit	Stack pointer, relative to SS segment
EBP	32-bit	Pointer to data in SS segment

The four "accumulator" registers (EAX, EBX, ECX and EDX) can be accessed as a single 32-bit value or as smaller pieces as shown in the following table. The table shows the names used to access the sub-registers of EAX – the naming conventions are similar for the others.

32	16	15	8	7	0
EAX					
		AX			
			AH		AL

There are also six 16-bit segment registers, which contain segment selectors and are used to define base addresses of memory regions. To access a region of memory, a segment register must first be loaded with a pointer (selector) to the appropriate segment descriptor defining the location and attributes of that region (segment) of memory..

Name	Purpose
CS	Code Segment
DS	Default Data Segment
SS	Stack Segment
ES	Extra Data Segment
FS	Additional Data Segment
GS	Additional Data Segment

The notation DS:BX is used to indicate an address formed using the value in BX as an offset into the segment addressed by DS. This segmentation model provides considerable flexibility in memory management but is largely unused by IA-32 software because most of the memory management features can be (and typically are) implemented via paging. You can find more information on this in section 3.4.

The simplest memory model involves setting all the segment registers to zero. In this case, there are six overlapping 4GB memory regions which are used by the program

There is a single 32-bit Instruction Pointer register (EIP) which is used, in conjunction with the CS segment register, to address and fetch instructions from the current code segment.

There are additional registers associated with floating point (8 x 80-bit data registers together with control and status registers) and also registers which are part of the Streaming SIMD Extensions (8 x 128-bit data registers).

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 other registers. This allows many novel uses of instructions which modify or access the pc to control program flow. ARM's pc register is analogous to IA-32's EIP register but in IA-32 there are very few operations which can be performed directly on or with EIP.

If the NEON and/or VFP extensions are implemented, these have their own register set which is separate to the core registers. If both are implemented, they share the same register file. The NEON and VFP register sets can be viewed as analogous to IA-32's SIMD registers.

3.1.2   Status registers

The ARM architecture specifies a single Current Program Status Register (CPSR) which contains both mode and status information. The contents are, in many ways, similar to the corresponding IA-32 EFLAGS register. Both contain a set of ALU status flags and interrupt enable bits.

This table shows the contents of the IA-32 EFLAGS register.

Bit(s)	Name		Purpose
31-22	Reserved		
21	ID	ID Flag	Indicates support for the CPUID instruction
20	VIP	Virtual Interrupt Pending Flag	If enabled, indicates that an interrupt is pending
19	VIF	Virtual Interrupt Flag	Virtual image of the IF flag
18	AC	Alignment Check	Enables alignment checking of memory accesses (with AM bit in CR0)
17	VM	Virtual-8086 Mode	Enables Virtual-8086 mode
16	RF	Resume Flag	Controls the behavior when an instruction breakpoint is detected
15	Reserved		
14	NT	Nested Task	Set when current task is linked to previous task
13-12	IOPL	I/O Privilege Level	Indicates the minimum privilege level required to access I/O address space
11	OF	Overflow Flag	ALU status flag
10	DF	Direction Flag	

Bit(s)	Name		Purpose
9	IF	Interrupt Enable Flag	Enables mask-able interrupts
8	TF	Trap Flag	Controls single-stepping
7	SF	Sign Flag	ALU status flag
6	ZF	Zero Flag	ALU status flag
5	Reserved		
4	AF	Auxiliary Carry Flag	ALU status flag
3	Reserved		
2	PF	Parity Flag	ALU status flag
1	Reserved		
0	CF	Carry Flag	ALU status flag

CF, PF, AF, ZF, SF and OF are set to indicate results of arithmetic operations. Only the Carry Flag may be altered directly (using STC, CLC and CMC instructions). In contrast, ARM allows direct test and modification of all flags under program control.

The Direction Flag (DF) controls the direction of string processing instructions i.e. whether they use incrementing or decrementing addressing. It is set via the STD and CLD instructions.

Only a subset of the fields in the EFLAGS register are modifiable at privilege level 3 (the level of user programs). Fields like IF, IOPL, NT, RF, VM, VIF and VIP can only be modified with privilege level 0 (Operating System privilege).

The ARMv7-A CPSR is as shown in the following table.

Bit(s)	Name		Purpose
31	N	N flag	ALU status flag (Negative)
30	Z	Z flag	ALU status flag (Zero)
29	C	C flag	ALU status flag (Carry)
28	V	V flag	ALU status flag (Overflow)
27	Q	Q flag	ALU status flag (Sticky overflow)
26-25	IT[de]	IF THEN state bits	State of IF THEN block
24	J	J bit	Indicates processor in Jazelle state
32-20	Reserved		
19-16	GE	SIMD ALU status flags	Set by SIMD instructions
15-10	IT[abc]	IF THEN state bits	State of IF THEN block
9	E	Endianness	Endianness of data memory accesses
8	A	Abort	Enables detection of asynchronous aborts
7	I	IRQ enable	Enables IRQ interrupts

Bit(s)	Name		Purpose
6	F	FIQ enable	Enables FIQ interrupts
5	T	T bit	Indicates processor in Thumb state
4-0	Mode	Mode bits	Indicate current processor more

In general, only the ALU flags (NZCVQ) are modifiable when executing in user mode. 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.

Of the IA-32 arithmetic status flags, SF, ZF, CF and OF correspond to ARM's N, Z, C and V. Note though that the ARM carry flag (C) has opposite semantics during subtraction. IA-32's PF and AF are not supported on ARM.

3.1.3 Instruction set

The ARM instruction set is 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 IA-32 architecture is a variable-length instruction set architecture. There are no alignment requirements for instructions in IA-32.

Some of the major differences in the instruction sets are listed below.

1. ARM is a load-store architecture

Common among RISC architectures, this means data processing instructions cannot operate directly on the contents of memory, they only operate on registers. Conversely, load and store instructions can only transfer data between registers and memory. In the IA-32 instruction set, in contrast, data processing instructions are capable of operating directly on memory as well as on registers.

2. IA-32 supports separate I/O instructions which access I/O address space

The IA-32 instruction set includes a set of instructions which address the I/O address space directly. The basic forms, IN and OUT, read and write single data items to and from I/O ports. These I/O ports may be addressed as bytes, halfwords or words. ARM has no equivalent and assumes that all peripherals are memory-mapped within the standard 4GB address space.

3. It is not possible to embed an arbitrary 32-bit address in an ARM instruction

Due to the fixed size of ARM instructions, it is not possible to encode a 32-bit address in an ARM instruction. All memory accesses, therefore, are made to addresses held in a register which is specified in the instruction. Another way of looking at this is that all ARM memory accesses are indirect through a general-purpose register. Embedding a 32-bit address in an instruction is possible in IA-32 due to the variable length nature of the instruction set.

4. ARM instructions cannot include arbitrary 32-bit constants

Similarly to the previous point, it is not possible to embed an arbitrary 32-bit constant in an ARM instruction (ARM instructions are fixed at 32-bit width). This means that compilers

(and assembly code programmers) needing to load constants frequently need to place them in memory and then load them using LDR instructions. Typically, these are embedded in the code stream, in what are termed "literal pools", and then loaded at run-time using PC-relative loads.

The instruction set is optimized for this particular usage and some encodings of load and store instructions implicitly use PC as the base register. This makes position-independent code relatively easy to write on ARM processors.

When using the Thumb instruction set, it is possible to synthesize arbitrary 32-bit constants using a two-instruction sequence.

Similarly to the previous point, the IA-32 instruction set does permit constants up to 32 bits to be encoded directly in instructions.

5.   ARM instructions are typically 3-operand

The majority of ARM data processing instructions take three operands, all of which can be registers, one of which may be a constant. In contrast, IA-32 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 IA-32 processor.

IA-32	ARM
`mov cx, ax   ; ca = ax` `add cx, bx   ; cx = cx + bx`	`add r0, r1, r2   ; r0 = r1 + r2`

6.   ARM subroutine instructions do not use the stack

The ARM BL instruction (used to make a subroutine call) places the return address in the Link Register (r14), it does not place it on the stack. Similarly, the standard ARM return instruction 'bx lr' branches to a return address in the link register rather than to an address stored on the stack (though it is possible, and common, to do this using an ldmfd instruction which does do so).

The IA-32 CALL and RET instructions, on the other hand, do place the return address on the stack and take it off (respectively). Further, the ENTER and LEAVE instructions are provided to create and release fixed-format stack frames on entering and leaving a procedure. ARM has no equivalent of this as it is very simple to add or subtract a fixed amount from the stack pointer instead.

While the IA-32 solution has greater hardware support (and thus requires less intervention from software), the ARM solution allows leaf functions to avoid using the stack altogether and does not place any restrictions on the composition of a stack frame.

7.   IA-32 uses a segmented addressing model

All IA-32 memory accesses are relative to one of the segment registers and these must be set first. Larger offsets require larger instructions to encode the larger constant. ARM has no concept of segmented addressing and there is no equivalent of the segment registers.

8.   IA-32 has special instructions for accessing many control registers and system functions

IA-32 specifies a number of special-purpose registers for configuration and control of the processor. Many of these are accessed via dedicated instructions.

The ARM approach is to place all of these types of register within a notional coprocessor, CP15. They are then accessed using standard coprocessor instructions, removing the need for a large number of dedicated instructions for accessing specific registers. This also means that adding features to future versions of the ARM architecture can be done without polluting either the instruction set space or the memory map.

For example, the instruction INVD is defined in IA-32 to invalidate the processor's internal caches. The same operation on ARM processors is carried out by executing an MCR instruction which writes to notional registers within CP15.

In more recent revisions, many of these operations are achieved in IA-32 via Model Specific Registers (MSRs). These are accessed via RDMSR and WRMSR instructions, which are privileged.

9. IA-32 generates an exception on divide-by-zero error

In ARMv7-A processors which support them, the ARM UDIV and SDIV instructions do not detect a divide-by-zero condition and always returns a zero result. In ARMv7-R processors, generation of an Undefined Instruction exception upon detection of divide-by-zero is optional.

10. ARM does not provide an array-bound checking instruction

In IA-32, the BOUND instruction is intended for checking memory addresses against array boundaries prior to carrying out a potentially invalid access. The ARM instruction set has no equivalent of this.

11. Many complex IA-32 instructions have no direct ARM equivalent

The ARM processor does not generally include instructions which carry out very complex multi-cycle operations. Examples of this include AES encryption, BCD conversion and manipulation, dot product, trig and log functions etc. However, with careful coding many of these operations can be coded very efficiently in assembly or sourced from libraries.

To the application programmer, writing in a high-level language, most of these differences are of little or no 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.

3.1.4 Operating modes

IA-32 supports four operating modes. It is worth noting that the major use of the different modes in IA-32 is to provide backward compatibility with earlier versions of the architecture. As a result, mode changes are associated with significant changes in functionality and operation (for instance, changes to the way in which fundamental addressing modes work). They are also relatively expensive to carry out. For this reason, IA-32 programs do not change mode very often after completing the startup process.

- **Real mode**
  This mode supports 20-bit physical addressing via a segmentation scheme. IA-32 processors boot in real mode.

- **Protected mode**
  This mode enables virtual memory addressing using 32-bit virtual addresses and is capable of accessing a 36-bit physical address space. The instruction set in Protected mode is backwards compatible with Real mode.

- **Virtual 8086 mode**
  A mode attribute within Protected mode which provides the ability to run legacy

16-bit code which needs to execute in Real mode on a system running in Protected mode.

- **System Management mode**
  Entered on interrupt or signal from APIC (or on occurrence of certain events as configured by the BIOS). Used by OS/executive for carrying out platform-specific system management tasks. Executes in a separate address space after automatically saving prior context. Context is restored on exit.

The majority of applications for IA-32 devices will execute in Protected mode, following a Real mode boot sequence. This is because virtual memory systems are only really possible in Protected mode. Similarly, Protected mode offers memory protection features which are necessary for process isolation (these are not provided in Real mode).

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. However, the purpose of these modes is, in the main, quite different from those listed above for IA-32 devices. Rather than being associated with compatibility or significant changes in function, in the ARM architecture modes are used for processing particular system events, typically exceptions. Mode changing on ARM is therefore quick, efficient and largely automatic.

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 operation in IA-32 is achieved by the INT instruction or the SYSENTER/SYSEXIT pair). 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.

3.1.5   Stack

The stack instructions provided in IA-32 (PUSH/POP and variants) assume a Full Descending Stack using SS:ESP as the stack pointer. This is a fixed feature of the instruction set.

The ARM architecture, on the other hand, allows greater flexibility. The LDM/STM instructions which are typically used for stack operations allow both full and empty, ascending and descending stack models and can, in principle, use any general purpose register as the stack pointer. The coding conventions used (specified in the ARM ABI), however, standardize on a Full Descending stack using SP (R13) as the stack pointer.

(The Thumb instruction set does not provide this full flexibility, supporting only the conventional stack).

The IA-32 PUSH/POP instructions allow items of varying width (halfword, word or doubleword) to be placed on the stack, incrementing or decrementing the stack pointer by a different amount in each case. The ARM stack operations, in contrast, are always word-sized. This greatly simplifies the management of stack frames, at the modest expense of some additional stack space.

IA-32 also provides a single instruction for pushing/popping all the general purpose registers. ARM's LDM/STM instructions also achieve this in a single operation. The ARM version, however, is significantly more flexible in that any subset of the register set can be pushed/popped in a single instruction.

Operation	IA-32	ARM
PUSH single	PUSH EAX	STR r2, [sp, #-4]!
POP single	POP EAX	LDR r2, [sp], #4
PUSH multiple	PUSHAD	PUSH {r0-r12} // or STMFD sp!, {r0-r12}
POP multiple	POPAD	POP {r0-r12} // or LDMFD sp!, {r0-r12}
PUSH status	PUSHFD	MRS r0, cpsr STR r0, [sp, #-4]!
POP status	POPFD	LDR r0. [sp], #4 MSR cpsr, r0

In the table, only the word-sized variants of the IA-32 instructions are shown.

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

3.1.6 Code execution

Both IA-32 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.

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 IA-32 architecture achieves this using a combination of special purpose registers and dedicated instructions.

For example, to flush and invalidate the data cache on an IA-32 device, you use the INVD instruction. On an ARM processor, you use an MCR instruction which transfers parameters to a specific register in a 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 IA-32 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: IA-32 uses an Advanced Programmable Interrupt Controller (APIC), ARM uses a Generic Interrupt Controller (GIC).

3.4 Memory

IA-32 supports two distinct addresses spaces: memory and I/O. I/O space is accessed via dedicated instructions using a 16-bit address. I/O address space is essentially a legacy feature and is not used in many modern systems.

Accessing memory space in IA32 depends on the current operating mode. In Protected Mode, in order to access an area of memory, that area must be set up as a 'segment' which is defined with a segment descriptor. A single application or task could be made up of numerous different code and data segments, each with their own start address, size and attributes (e.g. read-only / read-write). This approach, referred to as the 'multi-segment model,' provides a lot of flexibility in memory management, but is not typically used.

More commonly, the 'flat memory model' is the one of choice. In this model, segments still have to be defined (a requirement to operate in Protected Mode), but to make things simple, only a single code segment and a single data segment are used for all applications. Both the code and data segment are defined to cover the entire address space (0 to 4GB) with a common set of attributes. This provides software with a flat, linear 4GB address space. What is lost with this approach is the ability to assign specific attributes to certain ranges of memory (e.g. read-only vs. read-write) as well as the ability to 'protect' the memory space of one application from another application. These attributes and protection mechanisms can be set up and managed using the paging structures, which is what most modern Operating Systems do

ARM, on the other hand, supports a single, flat, linear 4GB address space. All peripherals are memory-mapped and there is no separate address space. There is also no concept of segmentation.

3.4.1 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.

This corresponds approximately to the "Strong Uncacheable" memory type in the IA-32 architecture.

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.

3.4.2 Memory map

Intel Atom devices have a fixed memory map in which general purpose DDR RAM lies in the first 2GB of the address space, with the remaining space allocated to PCI peripherals and legacy functionality. PCI ports can be configured to lie anywhere in the 32-bit address range.

The I/O space memory map is fixed and allocated to various internal peripherals, control and configuration functions.

ARMv7-A devices do not impose any fixed memory map with the exception of the default location of the exception vector table, which lies at the bottom of the memory map. This can, however, be relocated under either software or hardware control.

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.

There is a recommended reference memory map for ARM devices.

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.

The details of this 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, read-write and execute permissions. Permissions can also be policed separately for user and privileged mode accesses.

3.4.5 Access types, endianness and alignment

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

Both architectures are also natively little-endian. ARM devices, though, can be configured to work with big-endian data memory systems so you should check the documentation for your device carefully to determine how your system has been configured.

Both architectures provide rich instruction set support for accessing and manipulating mixed-endian data structures.

Although both architectures treat memory as byte-addressable, the ARM architecture imposes alignment restrictions on both code and data memory accesses.

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.

IA-32 processors have no restrictions on instruction alignment.

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.

IA-32 processors do not have generic alignment requirements like this (except for some SIMD instructions which explicitly work with packed and aligned data). However, as with ARM, data is accessed more efficiently when it is aligned according to size.

3.4.6 Atomicity

Both architectures define that only accesses which are naturally aligned according to their size are guaranteed atomic. This applies for accesses up to word size. An exception to this is for doubleword exclusive accesses on ARM. Accesses which are not so aligned may cross the boundary of the unit of atomic access.

Note that these rules change somewhat when IA-32 instructions are modified with the LOCK prefix. See 4.13 for more information.

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.

The ARM system of memory typing (in which memory is defined as "Normal", "Device" or "Strongly Ordered") and the weakly-ordered memory model used by ARMv7-A and IA-32 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).

- The IA-32 MFENCE instruction corresponds to the ARM DMB instruction.

- ARM has no equivalent of the LFENCE and SFENCE instructions which serialize only loads and stores respectively. DMB can be used instead.

- IA-32 has no single equivalent of the DSB instruction but several "serializing" instructions perform the same function as a side-effect.

- IA-32 has no equivalent to the ISB instruction. Instead the architecture requires that some operations (which e.g. modify page tables) are followed by a JUMP instruction.

One example of this is the relatively common technique of executing dynamically-generated (or self-modifying) code on IA-32 platforms. The instruction set and cache coherency architecture make it possible to do this without explicit cache management and barrier operations. These are necessary on ARM platforms and are often privileged thus requiring OS intervention, making this kind of technique a little more cumbersome.

In order to provide for synchronization functions, IA-32 provides for bus-locking (via a LOCK prefix to certain memory access instructions) which guarantees atomicity for affected accesses. The most common use of this is to lock the memory transactions associated with the XCHG instruction to form an atomic exchange of a register with a value in memory. This can be used to implement higher-level semaphore and lock constructs as required by operating systems.

Earlier ARM architectures supported similar behavior via the SWP instruction. This instruction is now deprecated and has been replaced by a non-blocking mechanism using "exclusive" memory access instructions, LDREX/STREX. These work with internal and, optionally, external exclusive access monitor logic within the memory system to provide for atomic software constructs.

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

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).

3.4.9   Caches

Both architectures support Harvard L1 Data and Instruction caches backed by a unified L2 cache.

		ARM (Cortex-A15)	IA-32 (Atom)
L1 Data	Size	32KB	24KB
	Associativity	2-way	6-way
	Line length	64 bytes	64 bytes
L1 Instruction	Size	32KB	32KB
	Associativity	2-way	8-way
	Line length	64 bytes	64 bytes
L2	Size	Configurable	512KB
	Associativity	16-way	8-way
	Line length	64 bytes	64 bytes

Notes:

- ARM Cortex-A15 L2 cache can be configured at synthesis-time to be 512KB, 1MB, 2MB or 4MB.

In multi-processor systems, both architectures support a MESI-like protocol for maintaining coherency in the L1 data caches.

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.

3.5   Self-modifying code

Much legacy IA-32 code uses techniques such as writing small sequences to the stack and executing them there. This can be done relatively freely by application programmers because it is possible without access to privileged operations. The Instruction cache snoops Data cache translations so instructions which are written to data memory and then fetched via the instruction interface are automatically synchronized.

This is not the case in the ARM architecture. Specifically, instructions written to memory via the data cache will not be read back via the instruction fetch interface unless the data cache is flushed and the instruction cache is invalidated. Additionally, data and instruction barriers will be required to ensure that external memory transactions have completed before attempting to fetch the newly-written instructions. This means that it is not easy to use these techniques on ARM systems without access to both barriers and cache maintenance operations. While applications can insert barriers into code sequences, the

cache maintenance operations are restricted to privileged mode only. Applications need to use Operating System calls to access them and this will entail some overhead.

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 (although it is not an architectural requirement in IA-32, it is provided on the Atom class of processors). 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.

To assist with performance benchmarking, both processors 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

Most IA-32 devices (including Atom) support Speedstep power management technology. This is a power management scheme enabled by any standard BIOS and exposed to the Operating System. It is generally the responsibility of the Operating System to make use of these facilities in response to dynamic system load conditions and application performance requirements.

ARMv7-A devices support a range of power modes and incorporate facilities for linking this with device-wide power management schemes. Again, making use of these is the responsibility of the Operating System. It is usually highly platform-dependent and managed by the 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.

Additionally, many IA-32 devices (including Atom) support multi-threading capability at the hardware level.

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.

       Application Note 274
ARM DAI 0274B

3.9   Multimedia extensions

The Atom processor supports the SSE3 Streaming SIMD Extensions to the IA-32 architecture. This provides a set of instructions and associated registers for accessing, packing, unpacking and processing packed data in a variety of lengths.

ARMv7-A devices support the optional NEON engine. This provides a similar set of functions. The NEON instruction set is much more orthogonal than SSE in that, in general, all operations are available on all data types and sizes. The structured load capability of NEON is particularly powerful.

Both can be accessed in a variety of ways:

- Direct coding in assembly language

- Automatic vectorization by the C compiler

- C intrinsic functions

Although the two technologies provide similar features, they are not compatible at instruction level. Apart from instances where the instructions have been generated by the C compiler, translation from one to the other is essentially a manual process. Any hand-coded SSE3 assembly code or SSE-specific compiler intrinsic will have to be replaced.

Note that 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 protected mode on an IA-32 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 an exception) 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 software interrupt (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 IA-32 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	IA-32 (System V)	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/32-bit align	
`float`	32-bit	32-bit	IEEE single-precision
`double`	64-bit	64-bit/32-bit align	IEEE double-precision
`long double`	64-bit	96-bit/32-bit align	IA-32 supports IEEE extended-precision
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.

The length of various types are more standardized in software for the ARM architecture due to the more rigorous architectural definition of data types and alignment requirements.

All ARM types are naturally aligned on a boundary equal to their size. Note that this is not the case for all IA-32 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 IA-32 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 IA-32 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 IA-32 architectures. Vendors such as Microsoft and Greenhills, 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 IA-32 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 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 and delays

It is common in IA-32 applications to use the RDTSC instruction (or the serializing RDTSCP variant) to read the system Time-Stamp Counter. This is a 64-bit counter which increases with each processor clock cycle. Using this, it is simple to implement timing delays to a very high resolution. Use of this instruction is not restricted to privileged modes.

The nearest equivalent to this in ARMv7-A processors is the cycle counter included as part of the Performance Monitoring Unit. However, access to this requires execution of privileged instructions and, if it is available to application code at all, is usually provided through an Operating System API.

 Application Note 274
ARM DAI 0274B

New ARM processors have generic timers which provide a high-precision counter which can be configured to be accessible in user mode. The Operating System should make available an API for accessing these.

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 IA-32 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 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   Hardware discovery

The IA-32 CPUID instruction provides a mechanism for determining details about the underlying platform (e.g. cache architecture, performance monitoring capabilities, physical characteristics, architecture extensions etc). Similar information is available on ARM platforms but the instructions required to return it may be restricted to privileged operation. This means that application programmers are required to use Operating System APIs to determine this kind of information.

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 must, in general, be avoided 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.

IA-32, as mentioned earlier, incorporates a separate I/O address space which is accessed via dedicated instructions. 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 accesses IA-32 peripherals via the I/O space will need rewriting.

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 significantly more strict than IA-32. 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 e.g. the packed attribute. 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 IA-32 assembly code as extensive optimized libraries are available for common functions targeting ARM platforms.

Another alternative is to rewrite short, common sequences using compiler intrinsics. 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 but via very different mechanisms as explained in 3.4.7 above.

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

IA-32	ARM
```get_lock cmp [edx], 0 jne get_lock mov eax, 1 xchg eax, [edx] cmp eax, 0 jne get_lock ...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 mov [edx], 0```	```unlock DMB MOV r1, #0 STR r1, [r0]```

As mentioned earlier, the ARM mechanism relies on some logic (an "exclusive monitor") within the memory system for recording the fact that a reservation exists on a particular memory location. 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 IA-32 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 IA-32 code which makes use of extended-precision 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.

- IA-32 devices using x87 floating point use an 80-bit intermediate format when carrying out floating point calculations. This results in rounding behavior which is not IEEE-compliant. ARM devices are IEEE-compliant, so numerical results may differ slightly when executing identical code on the respective platforms. Note that IA-32 devices perform much more like ARM if SSE instructions are used for floating point rather than the x87 coprocessor. Many compilers include options to force this behavior.

- 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 are unlikely to 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 IA-32 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. JUMP 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.