C 26-bit Operations on ARM810

This appendix describes the 26-bit operations on ARM810.

26-bit Operations on ARM810

C.1 Introduction

To maintain compatibility with earlier ARM processors, it is possible to execute code in 26-bit operating modes **usr26**, **fiq26**, **irq26** and **svc26**. Details of how to do this have already been written for earlier ARM processors, and these have been included here for your information.

This appendix summarises how 26-bit binary code will be able to run on the ARM810 processor. The details below show the instruction and performance differences when ARM810 is operated in 26-bit modes. The last section describes the hardware changes that affect 26-bit operation.

Use of 26-bit modes for any reason other than executing existing 26-bit code is strongly discouraged, as this will no longer be supported in ARM processors after the ARM810. It is also worth noting that ARM810's performance in 26-bit modes may be poorer than in 32-bit modes.

C.2 Instruction Differences

When ARM810 is executing in a 26-bit mode, the top 6 bits of the PC are forced to be zero at all times. The following restrictions must be obeyed to avoid problems due to the Prefetch Unit having prefetched an unknown distance beyond the current instruction:

- Do not enter any 26-bit mode when at an address outside the 26-bit address space.
- Do not execute code sequentially from address 0x03FFFFFC to address 0x00000000 in 26-bit code.

An additional requirement for 32-bit and 26-bit operations is that if a system contains code that is intended for execution in both 26-bit and 32-bit modes, an IMB instruction must accompany any change from any 26-bit mode to any 32-bit mode, and vice versa. It is therefore advisable to keep code intended for 26-bit modes and code intended for 32-bit modes completely separate.

26-bit operation removes some of the instruction constraints placed on 32-bit code. 26-bit code must obey the constraints laid out for 32-bit code with the following exceptions:

1 CMN, CMP, TEQ, TST
A second form of these instructions becomes available, which is encoded in the instruction by setting the Rd field to “1111” and in the assembler syntax by using:

```
<opcode>{cond}P
```

in place of the normal

```
<opcode>{cond}
```

In all modes, the normal setting of the CPSR flags is suppressed for the new form of the instruction. Instead, the normal arithmetic result is calculated (Op1+Op2, Op1-Op2, Op1 EOR Op2 and Op1 AND Op2 for CMN, CMP, TEQ and TST respectively) and used to set selected CPSR bits. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

The CMNP, CMPP, TEQP and TSTP instructions take a base of 3 cycles to execute, along with the extra cycles listed in ***4.5.8 Instruction cycle times*** on page 4-14 for complex and register-specified shifts..

2 Data processing instructions with destination register R15 and the S bit set
These become valid in User mode, and their behaviour in all modes is altered.

In all modes, the normal setting of the CPSR flags from the current mode’s SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the arithmetic result; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the arithmetic result.

3 LDM with R15 in Register list, and the S bit set
This becomes valid in User mode, and its behaviour in all modes is altered.

In all modes, the normal setting of the CPSR flags from the current mode's SPSR is suppressed. In user mode, the N, Z, C and V bits of the CPSR are set to bits 31 to 28 of the value loaded for R15; in non-user modes, the N, Z, C, V, I, F, M1 and M0 bits of the CPSR are set to bits 31 to 26, 1 and 0 of the value loaded for R15.

4 Address Exceptions
The address exceptions which occur on true 26-bit ARM processors cannot occur on ARM810. If required, these should now be generated externally to the ARM810 as aborts, along with an abort handler routine which recognises the address exception.

Note: Some unusual coding cases may present problems: for example, LDMs and STMs wrapping around from the top of 26-bit memory space to the bottom. It is thought that such cases are not in common use, and so should not present any difficulties.

C.3 Performance Differences

This information is provisional at this release of the data sheet. Implementation details may affect performance.

There is no cycle count performance degradation for operating in 26-bit mode; the cycle counts are the same as those for 32-bit mode operations. However, there may be degradation due to the additional software overheads in getting to and from 32-bit-mode-only operations.

C.4 Hardware Compatibility Issues

This section describes the ways in which the ARM810 will differ from previous ARM processors, as far as its hardware is concerned, for 26-bit compatibility.

This section is up-to-date, but is not necessarily complete.

C.4.1 Configuration

ARM810 will not have the two configuration bits, **DATA32** and **PROG32** that could be found on previous ARM610 and ARM710 processors.

As such, the processor's normal mode of operation is in full 32-bit modes: as if both of these bits were configured in their active HIGH state. Aborts on Read and Write of the Exception Vectors can be done by the Memory Manager, thus stimulating the original hardware configurations in software.

D

Comparisons with 26-bit ARM Processors

This appendix describes the differences between 32-bit ARM processors and earlier 26-bit ARM processors:

- 32-bit ARM processors are the ARM6 family, and all later processors including the ARM7 family, the ARM8 family and StrongARM.
- 26-bit ARM processors are ARM2, ARM3 and ARM2aS.

This information is included here for completeness as it provides further details about the differences between 26-bit and 32-bit codes to that described in ***Appendix C, 26-bit Operations on ARM810***. The information in the rest of the datasheet supersedes this appendix.

Comparisons with 26-bit ARM Processors

D.1 Introduction

The ARM6 family, and all later processors including the ARM7 family, the ARM8 family and StrongARM, are ARM processors that have 32-bit program counters. Earlier ARMs (ARM2, ARM3 and ARM2aS) had a 26-bit program counter (PC). This appendix describes the major differences between the two types of processor.

D.2 The Program Counter and Program Status Register

The introduction of the larger program counter has meant that the flags and control bits of R15 (the combined PC and PSR) have been moved to a separate register. The extra space in the new register (the CPSR, Current Program Status Register) allows for more control bits. A further 3 mode bits have been added to allow for a larger number of operating modes.

The removal of the PSR to a separate register also means that it is no longer possible to save these flags automatically in R14 when a Branch with Link (BL) instruction is executed, or when an exception occurs. Program analysis has shown that the saving of these flags is only required in 3% of subroutine calls, so there is only a slight overhead in explicitly saving them on a stack when necessary. To cope with the requirement of saving them when an exception occurs, 5 further registers have been provided to hold a copy of the CPSR at the time of the exception. These registers are the Saved Program Status Registers (SPSRs). There is one SPSR for each of the modes that the processor may enter as a result of the various types of exception.

The expansion of the PC to 32 bits also means that the Branch instruction, being limited to +/-32 Mbytes, can no longer specify a branch to the entire program space. Branches greater than +/-32 Mbytes can be made with other instructions, but the equivalent of the Branch with Link instruction will require a separate instruction to save the PC in R14.

D.3 Operating Modes

There are a total of 10 operating modes in two overlapping sets. Four modes—**User26**, **IRQ26**, **FIQ26** and **Supervisor26**—allow the processor to behave like earlier ARM processors with a 26-bit PC. These correspond to the four operating modes of the ARM2 and ARM3 processors. A further four operating modes correspond to these, but with the processor running with the full 32-bit PC: these are **User32**, **IRQ32**, **FIQ32** and **Supervisor32**.

The final two modes are **Undefined32** and **Abort32**, and are entered when the Undefined instruction and Abort exceptions occur. They have been added to remove restrictions on Supervisor mode programs which exist with the ARM2 and ARM3 processors. The two sets of User, FIQ, IRQ and Supervisor modes each share a set of banked registers to allow them to maintain some private state at all times. The Abort and Undefined modes also have a pair of banked registers each for the same purpose.

D.4 Instruction Set Changes

The instruction set is changed in two major areas: new instructions have been introduced and restrictions have been placed on existing ones.

D.4.1 New Instructions

The new instructions allow access to the CPSR and SPSR registers. They are formed by using opcodes from the Data Processing group of instructions that were previously unused. Specifically, these are the TST, TEQ, CMP and CMN instructions with the S flag clear. They are now known as MSR to move data into the CPSR and SPSR registers, and MRS to move from the CPSR and SPSR to a general register. The data moved to CPSR and SPSR can be either the contents of a general register or an immediate value.

D.4.2 Instruction Set Limitations

When configured for 32-bit program and data space, 32-bit processors support operation in 26-bit modes for compatibility with ARM processors that have a 26-bit address space. The 26-bit modes are **User26**, **FIQ26**, **IRQ26** and **Supervisor26**. When a 26-bit mode is selected, the programmer's model reverts to that of existing 26 bit ARMs (ARM2, ARM3, ARM2aS). The behaviour is that of the ARM2aS macrocell with the following alterations:

- Address exceptions are *never* generated. The OS may simulate the behaviour of address exception by using external logic such as a memory management unit to generate an abort if the 64 Mbyte range is exceeded, and converting that abort into an "address exception" trap for the application.

Note Address exceptions are still possible when the processor is configured for 26-bit program and data space.

- The new instructions to transfer data between general registers and the program status registers remain operative. The new instructions can be used by the operating system to return a 32-bit operating mode after calling a binary containing code written for a 26-bit ARM.
- All exceptions (including Undefined Instruction and Software Interrupt) return the processor to a 32-bit mode, so the operating system must be modified to handle them.
- 32-bit processors include hardware which prevents the write operation and generates a data abort if the processor attempts to write to a location between &00000000 and &0000001F inclusive (the exception vectors) when operating in 26-bit mode. This allows the operating system to intercept all changes to the exception vectors and redirect the vector to some veneer code. The veneer code should place the processor in a 26-bit mode before calling the 26-bit exception handler.

In all other respects, 32-bit processors behave like a 26-bit ARM when operating in 26-bit mode. The relevant bits of the CPSR appear to be incorporated back into R15 to form the PC/CPSR with the I and F bits in bits 27 and 26. The instruction set behaves like that of the ARM2aS macrocell with the addition of the MRS and MSR instructions.

D.5 Transferring between 26-bit and 32-bit Modes

A program executing in a privileged 32-bit mode can enter a 26-bit mode by executing an MSR instruction which alters the mode bits to one of the values shown below:

M[4:0]	Mode	Accessible register set
00000	usr26	PC/PSR, R14..R0, CPSR
00001	fiq26	PC/PSR, R14_fiq..R8_fiq, R7..R0, CPSR, SPSR_fiq
00010	irq26	PC/PSR, R14_irq..R13_fiq, R12..R0, CPSR, SPSR_irq
00011	svc26	PC/PSR, R14_svc..R13_svc, R12..R0, CPSR, SPSR_svc

Table D-1: MSR instruction altering the mode bits

Transfer between 26-bit and 32-bit mode happens automatically whenever an exception occurs in 26-bit mode. Note that an exception (including Software Interrupt) arising in 26-bit mode will enter 32-bit mode and the saved value in R14 will contain only the PC, even though the PSR was also considered part of R15 when the exception arose.

In addition, the MSR instruction provides the means for a program in a privileged 26-bit mode to alter the mode bits to change to a 32-bit mode.

E

Implementing the Instruction Memory Barrier Instruction

This appendix is written to help Operating System designers understand and implement the IMB Instructions. It firstly describes the generic approach that should be used for future compatibilty and then goes on to ARM810-specific details.

Implementing the Instruction Memory Barrier Instruction

E.1 Introduction

This appendix describes the processor specific code that must be included in the SWI handler to implement the two Instruction Memory Barrier (IMB) Instructions:

- IMB
- IMBRange

These are implemented as calls to specific SWI numbers. Please refer to ***4.17 The Instruction Memory Barrier (IMB) Instruction*** on page 4-64 for further details of this and for examples of use.

Two IMB instructions are provided so that when only a small area of code is altered before being executed the IMBRange instruction can be used to efficiently and quickly flush any stored instruction information from addresses within a small range rather than flushing all information about all instructions using the IMB instruction.
By flushing only the required address range information, the rest of the information remains to provide improved system performance.

E.2 ARM810 IMB Implementation

For ARM810, executing the SWI instruction is sufficient in itself to cause the IMB operation. Also, for ARM810, both the IMB and the IMBRange instructions flush *all* stored information about the instruction stream.

This means that for ARM810, all IMB instructions can be implemented in the Operating System by simply returning from the IMB/IMBRange service routine AND that the service routines can be exactly the same. The following service routine code can be used for ARM810:

```
IMB_SWI_handler
IMBRange_SWI_handler


        MOVS  PC, R14_svc; Return to the code after the SWI call
```

Note: It is strongly encouraged that in code from now on, the IMBRange instruction is used whenever the changed area of code is small: even if there is no distinction between it and the IMB instruction on ARM810. Future processors may well implement the IMBRange instruction in a much more efficient and faster manner, and code migrated from ARM810 will benefit when executed on these processors.

E.3 Generic IMB Use

Using SWI's to implement the IMB instructions means that any code that is written now will be compatible with any future processors - even if those processors implement IMB in different ways. This is achieved by changing the Operating System SWI service routines for each of the IMB SWI numbers that differ from processor to processor.

Below are examples that show what should happen during the execution of IMB instructions. These examples are taken from ***4.17.3 Examples*** on page 4-65.

The pseudo code in the square brackets shows what should happen to execute the IMB instruction (or IMBRange) in the SWI handler.

E.3.1 Loading code from disk

Code that loads a program from a disk, and then branches to the entry point of that program, must execute an IMB instruction between loading the program and trying to execute it.

```
IMBEQU 0xF00000
 .
 .
 ; code that loads program from disk
 .
 .
 SWI IMB
    [branch to IMB service routine]
    [perform processor-specific operations to execute IMB]
    [return to code]
    .
 MOV PC, entry_point_of_loaded_program
    .
    .
```

E.3.2 Running BitBlt code

“Compiled BitBlt” routines optimise large copy operations by constructing and executing a copying loop which has been optimised for the exact operation wanted. When writing such a routine an IMB is needed between the code that constructs the loop and the actual execution of the constructed loop.

```
IMBRange EQU 0xF00001

.
.
; code that constructs loop code
; load R0 with the start address of the constructed loop
; load R1 with the end address of the constructed loop
SWI   IMBRange
      [branch to IMBRange service routine]
      [read registers R0 and R1 to set up address range
            parameters]
      [perform processor-specific operations to execute
            IMBRange within address range]
      [return to code]
; start of loop code
.
.
```

Implementing the Instruction Memory Barrier Instruction

Index