

Chapter A2

Programmers' Model

This chapter introduces the ARM® Programmers' Model. It contains the following sections:

- *Data types* on page A2-2
- *Processor modes* on page A2-3
- *Registers* on page A2-4
- *General-purpose registers* on page A2-6
- *Program status registers* on page A2-11
- *Exceptions* on page A2-16
- *Endian support* on page A2-30
- *Unaligned access support* on page A2-38
- *Synchronization primitives* on page A2-44
- *The Jazelle Extension* on page A2-53
- *Saturated integer arithmetic* on page A2-69.

A2.1 Data types

ARM processors support the following data types:

Byte	8 bits
Halfword	16 bits
Word	32 bits

Note

- Support for halfwords was introduced in version 4.
- ARMv6 has introduced unaligned data support for words and halfwords. See *Unaligned access support* on page A2-38 for more information.
- When any of these types is described as *unsigned*, the N-bit data value represents a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.
- When any of these types is described as *signed*, the N-bit data value represents an integer in the range -2^{N-1} to $+2^{N-1}-1$, using two's complement format.
- Most data operations, for example ADD, are performed on word quantities. Long multiplies support 64-bit results with or without accumulation. ARMv5TE introduced some halfword multiply operations. ARMv6 introduced a variety of Single Instruction Multiple Data (SIMD) instructions operating on two halfwords or four bytes in parallel.
- Load and store operations can transfer bytes, halfwords, or words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded. Load and store operations that transfer two or more words to and from memory are also provided.
- ARM instructions are exactly one word and are aligned on a four-byte boundary. Thumb® instructions are exactly one halfword and are aligned on a two-byte boundary. Jazelle® opcodes are a variable number of bytes in length and can appear at any byte alignment.

A2.2 Processor modes

The ARM architecture supports the seven processor modes shown in Table A2-1.

can read from
co-processor

Table A2-1 ARM processor modes

Processor mode		Mode number	Description
User	usr	0b10000	Normal program execution mode
FIQ	fiq	0b10001	Supports a high-speed data transfer or channel process
IRQ	irq	0b10010	Used for general-purpose interrupt handling
Supervisor	svc	0b10011	A protected mode for the operating system
Abort	abt	0b10111	Implements virtual memory and/or memory protection
Undefined	und	0b11011	Supports software emulation of hardware coprocessors
System	sys	0b11111	Runs privileged operating system tasks (ARMv4 and above)

for our
timer
interrupt
(& other)

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in User mode. When the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur (see *Exceptions* on page A2-16). This allows a suitably-written operating system to control the use of system resources.

The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change mode freely. Five of them are known as *exception modes*:

- FIQ
- IRQ
- Supervisor
- Abort
- Undefined.

These are entered when specific exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs (see *Registers* on page A2-4 for details).

The remaining mode is System mode, which is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks that need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.


A2.3 Registers

The ARM processor has a total of 37 registers:

- Thirty-one general-purpose registers, including a program counter. These registers are 32 bits wide and are described in *General-purpose registers* on page A2-6.
- Six status registers. These registers are also 32 bits wide, but only some of the 32 bits are allocated or need to be implemented. The subset depends on the architecture variant supported. These are described in *Program status registers* on page A2-11.

Registers are arranged in partially overlapping banks, with the current processor mode controlling which bank is available, as shown in Figure A2-1 on page A2-5. At any time, 15 general-purpose registers (R0 to R14), one or two status registers, and the program counter are visible. Each column of Figure A2-1 on page A2-5 shows which general-purpose and status registers are visible in the indicated processor mode.

Modes						
Privileged modes						
Exception modes						
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

 indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode

we currently run in system mode, will do user soon-ish.

private copies

Figure A2-1 Register organization

r13 = stack pointer sp
 r14 = link register lr
 r15 = program counter

r0-r3 = argument registers, caller-saved

A2.4 General-purpose registers

The general-purpose registers R0 to R15 can be split into three groups. These groups differ in the way they are banked and in their special-purpose uses:

- The *unbanked registers*, R0 to R7
- The *banked registers*, R8 to R14
- Register 15, the PC, is described in *Register 15 and the program counter* on page A2-9.

A2.4.1 The unbanked registers, R0 to R7

Registers R0 to R7 are *unbanked registers*. This means that each of them refers to the same 32-bit physical register in all processor modes. They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

A2.4.2 The banked registers, R8 to R14

r8-r12 only banked by FIQ

Registers R8 to R14 are *banked registers*. The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name (as described below) is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

————— Note —————

There are a few exceptions to this rule for processors pre-ARMv6, and they are noted in the individual instruction descriptions. Where a restriction exists on the use of banked registers, it always applies to all of R8 to R14. For example, R8 to R12 are subject to such restrictions even in systems in which FIQ mode is never used and so only one physical version of the register is ever in use.

Registers R8 to R12 have two banked physical registers each. One is used in all processor modes other than FIQ mode, and the other is used in FIQ mode. Where it is necessary to be specific about which version is being referred to, the first group of physical registers are referred to as R8_usr to R12_usr and the second group as R8_fiq to R12_fiq.

Registers R8 to R12 do not have any dedicated special purposes in the architecture. However, for interrupts that are simple enough to be processed using registers R8 to R14 only, the existence of separate FIQ mode versions of these registers allows very fast interrupt processing.

Registers R13 and R14 have six banked physical registers each. One is used in User and System modes, and each of the remaining five is used in one of the five exception modes. Where it is necessary to be specific about which version is being referred to, you use names of the form:

R13_<mode>
R14_<mode>

where <mode> is the appropriate one of usr, svc (for Supervisor mode), abt, und, irq and fiq.

Register R13 is normally used as a stack pointer and is also known as the SP. The SRS instruction, introduced in ARMv6, is the only ARM instruction that uses R13 in a special-case manner. There are other such instructions in the Thumb instruction set, as described in Chapter A6 *The Thumb Instruction Set*.

Each exception mode has its own banked version of R13. Suitable uses for these banked versions of R13 depend on the architecture version:

- In architecture versions earlier than ARMv6, each banked version of R13 will normally be initialized to point to a stack dedicated to that exception mode. On entry, the exception handler typically stores the values of other registers that it wants to use on this stack. By reloading these values into the register when it returns, the exception handler can ensure that it does not corrupt the state of the program that was being executed when the exception occurred.

If fewer exception-handling stacks are desired in a system than this implies, it is possible instead to initialize the banked version of R13 for an exception mode to point to a small area of memory that is used for temporary storage while transferring to another exception mode and its stack. For example, suppose that there is a requirement for an IRQ handler to use the Supervisor mode stack to store SPSR_irq, R0 to R3, R12, R14_irq, and then to execute in Supervisor mode with IRQs enabled. This can be achieved by initializing R13_irq to point to a four-word temporary storage area, and using the following code sequence on entry to the handler:

"store multiple, increment after"

```

STMIA R13, (R0-R3)    ; Put R0-R3 into temporary storage
MRS   R0, SPSR        ; Move banked SPSR and R12-R14 into
MOV   R1, R12          ; unbanked registers
MOV   R2, R13
MOV   R3, R14
MRS   R12, CPSR        ; Use read/modify/write sequence
BIC   R12, R12, #0x1F ; on CPSR to switch to Supervisor
ORR   R12, R12, #0x13 ; mode
MSR   CPSR_c, R12
STMFD R13!, (R1,R3)    ; Push original {R12, R14_irq}, then
STR   R0, [R13,#-20]! ; SPSR_irq with a gap for R0-R3
LDMIA R2, {R0-R3}      ; Reload R0-R3 from temporary storage
BIC   R12, R12, #0x80 ; Modify and write CPSR again to
MSR   CPSR_c, R12      ; re-enable IRQs
STMIB R13, {R0-R3}     ; Store R0-R3 in the gap left on the
                        ; stack for them

```

CPSR_c = only write control bits

! means write back to reg

- In ARMv6 and above, it is recommended that the OS designer should decide how many exception-handling stacks are required in the system, and select a suitable processor mode in which to handle the exceptions that use each stack. For example, one exception-handling stack might be required to be locked into real memory and be used for aborts and high-priority interrupts, while another could use virtual memory and be used for SWIs, Undefined instructions and low-priority interrupts. Suitable processor modes in this example might be Abort mode and Supervisor mode respectively.

The banked version of R13 for each of the selected modes is then initialized to point to the corresponding stack, and the other banked versions of R13 are normally not used. Each exception handler starts with an SRS instruction to store the exception return information to the appropriate stack, followed (if necessary) by a CPS instruction to switch to the appropriate mode and possibly

re-enable interrupts, after which other registers can be saved on that stack. So in the above example, an Undefined Instruction handler that wants to re-enable interrupts immediately would start with the following two instructions:

```
SRSFD    #svc_mode!
CPSIE    i, #svc_mode
```

The handler can then operate entirely in Supervisor mode, using the virtual memory stack pointed to by R13_svc.

Register R14 (also known as the *Link Register* or LR) has two special functions in the architecture:

- In each mode, the mode's own version of R14 is used to hold subroutine return addresses. When a subroutine call is performed by a BL or BLX instruction, R14 is set to the subroutine return address. The subroutine return is performed by copying R14 back to the program counter. This is typically done in one of the two following ways:
 - Execute a BX LR instruction.

Note

An MOV PC, LR instruction will perform the same function as BX LR if the code to which it returns uses the current instruction set, but will not return correctly from an ARM subroutine called by Thumb code, or from a Thumb subroutine called by ARM code. The use of MOV PC, LR instructions for subroutine return is therefore deprecated.

- On subroutine entry, store R14 to the stack with an instruction of the form:


```
STMFDP SP!, {<registers>, LR}
```

 and use a matching instruction to return:


```
LDMFDP SP!, {<registers>, PC}
```
- When an exception occurs, the appropriate exception mode's version of R14 is set to the exception return address (offset by a small constant for some exceptions). The exception return is performed in a similar way to a subroutine return, but using slightly different instructions to ensure full restoration of the state of the program that was being executed when the exception occurred. See *Exceptions* on page A2-16 for more details.

Register R14 can be treated as a general-purpose register at all other times.

Note

When nested exceptions are possible, the two special-purpose uses might conflict. For example, if an IRQ interrupt occurs when a program is being executed in User mode, none of the User mode registers are necessarily corrupted. But if an interrupt handler running in IRQ mode re-enables IRQ interrupts and a nested IRQ interrupt occurs, any value the outer interrupt handler is holding in R14_irq at the time is overwritten by the return address of the nested interrupt.

System programmers need to be careful about such interactions. The usual way to deal with them is to ensure that the appropriate version of R14 does not hold anything significant at times when nested exceptions can occur. When this is hard to do in a straightforward way, it is usually best to change to another

processor mode during entry to the exception handler, before re-enabling interrupts or otherwise allowing nested exceptions to occur. (In ARMv4 and above, System mode is often the best mode to use for this purpose.)

A2.4.3 Register 15 and the program counter

Register R15 (R15) is often used in place of the other general-purpose registers to produce various special-case effects. These are instruction-specific and so are described in the individual instruction descriptions.

There are also many instruction-specific restrictions on the use of R15. These are also noted in the individual instruction descriptions. Usually, the instruction is UNPREDICTABLE if R15 is used in a manner that breaks these restrictions.

If an instruction description neither describes a special-case effect when R15 is used nor places restrictions on its use, R15 is used to read or write the *Program Counter* (PC), as described in:

- *Reading the program counter*
- *Writing the program counter* on page A2-10.

Reading the program counter

When an instruction reads the PC, the value read depends on which instruction set it comes from:

- For an ARM instruction, the value read is the address of the instruction plus 8 bytes. Bits [1:0] of this value are always zero, because ARM instructions are always word-aligned.
- For a Thumb instruction, the value read is the address of the instruction plus 4 bytes. Bit [0] of this value is always zero, because Thumb instructions are always halfword-aligned.

This way of reading the PC is primarily used for quick, position-independent addressing of nearby instructions and data, including position-independent branching within a program.

An exception to the above rule occurs when an ARM STR or STM instruction stores R15. Such instructions can store either the address of the instruction plus 8 bytes, like other instructions that read R15, or the address of the instruction plus 12 bytes. Whether the offset of 8 or the offset of 12 is used is IMPLEMENTATION DEFINED. An implementation must use the same offset for all ARM STR and STM instructions that store R15. It cannot use 8 for some of them and 12 for others.

Because of this exception, it is usually best to avoid the use of STR and STM instructions that store R15. If this is difficult, use a suitable instruction sequence in the program to ascertain which offset the implementation uses. For example, if R0 points to an available word of memory, then the following instructions put the offset of the implementation in R0:

```
SUB R1, PC, #4    ; R1 = address of following STR instruction
STR PC, [R0]      ; Store address of STR instruction + offset,
LDR R0, [R0]      ; then reload it
SUB R0, R0, R1     ; Calculate the offset as the difference
```

Note

The rules about how R15 is read apply only to reads by instructions. In particular, they do not necessarily describe the values placed on a hardware address bus during instruction fetches. Like all other details of hardware interfaces, such values are IMPLEMENTATION DEFINED.

Writing the program counter

When an instruction writes the PC, the normal result is that the value written to the PC is treated as an instruction address and a branch occurs to that address.

Since ARM instructions are required to be word-aligned, values they write to the PC are normally expected to have bits[1:0] == 0b00. Similarly, Thumb instructions are required to be halfword-aligned and so values they write to the PC are normally expected to have bit[0] == 0.

The precise rules depend on the current instruction set state and the architecture version:

- In T variants of ARMv4 and above, including all variants of ARMv6 and above, bit[0] of a value written to R15 in Thumb state is ignored unless the instruction description says otherwise. If bit[0] of the PC is implemented (which depends on whether and how the Jazelle Extension is implemented), then zero must be written to it regardless of the value written to bit[0] of R15.
- In ARMv6 and above, bits[1:0] of a value written to R15 in ARM state are ignored unless the instruction description says otherwise. Bit[1] of the PC must be written as zero regardless of the value written to bit[1] of R15. If bit[0] of the PC is implemented (which depends on how the Jazelle Extension is implemented), then zero must be written to it.
- In all variants of ARMv4 and ARMv5, bits[1:0] of a value written to R15 in ARM state must be 0b00. If they are not, the results are UNPREDICTABLE.

Several instructions have their own rules for interpreting values written to R15. For example, BX and other instructions designed to transfer between ARM and Thumb states use bit[0] of the value to select whether to execute the code at the destination address in ARM state or Thumb state. Special rules of this type are described on the individual instruction pages, and override the general rules in this section.

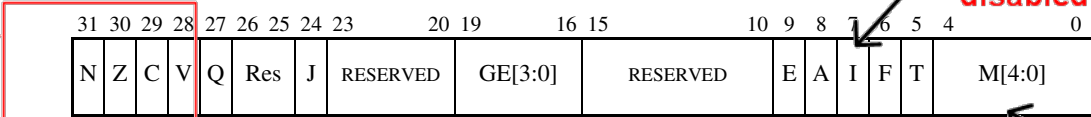
A2.5 Program status registers

The *Current Program Status Register* (CPSR) is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information. Each exception mode also has a *Saved Program Status Register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs.

————— **Note** —————

User mode and System mode do not have an SPSR, because they are not exception modes. All instructions that read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

The format of the CPSR and the SPSRs is shown below.



A2.5.1 Types of PSR bits

PSR bits fall into four categories, depending on the way in which they can be updated:

- Reserved bits** Reserved for future expansion. Implementations must read these bits as 0 and ignore writes to them. For maximum compatibility with future extensions to the architecture, they must be written with values read from the same bits.
- User-writable bits** Can be written from any mode. The N, Z, C, V, Q, GE[3:0], and E bits are user-writable.
- Privileged bits** Can be written from any privileged mode. Writes to privileged bits in User mode are ignored. The A, I, F, and M[4:0] bits are privileged.
- Execution state bits** Can be written from any privileged mode. Writes to execution state bits in User mode are ignored. The J and T bits are execution state bits, and are always zero in ARM state.

Privileged MSR instructions that write to the CPSR execution state bits must write zeros to them, in order to avoid changing them. If ones are written to either or both of them, the resulting behavior is UNPREDICTABLE. This restriction applies only to the CPSR execution state bits, not the SPSR execution state bits.

A2.5.2 The condition code flags

The N, Z, C, and V (Negative, Zero, Carry and oVerflow) bits are collectively known as the *condition code flags*, often referred to as *flags*. The condition code flags in the CPSR can be tested by most instructions to determine whether the instruction is to be executed.

The condition code flags are usually modified by:

- Execution of a comparison instruction (CMN, CMP, TEQ or TST).
- Execution of some other arithmetic, logical or move instruction, where the destination register of the instruction is not R15. Most of these instructions have both a flag-preserving and a flag-setting variant, with the latter being selected by adding an S qualifier to the instruction mnemonic. Some of these instructions only have a flag-preserving version. This is noted in the individual instruction descriptions.

In either case, the new condition code flags (after the instruction has been executed) usually mean:

- N** Is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.
- Z** Is set to 1 if the result of the instruction is zero (this often indicates an *equal* result from a comparison), and to 0 otherwise.
- C** Is set in one of four ways:
- For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.
 - For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
 - For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
 - For other non-addition/subtractions, C is normally left unchanged (but see the individual instruction descriptions for any special cases).
- V** Is set in one of two ways:
- For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
 - For non-addition/subtractions, V is normally left unchanged (but see the individual instruction descriptions for any special cases).

The flags can be modified in these additional ways:

- Execution of an MSR instruction, as part of its function of writing a new value to the CPSR or SPSR.
- Execution of MRC instructions with destination register R15. The purpose of such instructions is to transfer coprocessor-generated condition code flag values to the ARM processor.
- Execution of some variants of the LDM instruction. These variants copy the SPSR to the CPSR, and their main intended use is for returning from exceptions.
- Execution of an RFE instruction in a privileged mode that loads a new value into the CPSR from memory.
- Execution of flag-setting variants of arithmetic and logical instructions whose destination register is R15. These also copy the SPSR to the CPSR, and are intended for returning from exceptions.

A2.5.3 The Q flag

In E variants of ARMv5 and above, bit[27] of the CPSR is known as the Q flag and is used to indicate whether overflow and/or saturation has occurred in some DSP-oriented instructions. Similarly, bit[27] of each SPSR is a Q flag, and is used to preserve and restore the CPSR Q flag if an exception occurs. See *Saturated integer arithmetic* on page A2-69 for more information.

In architecture versions prior to ARMv5, and in non-E variants of ARMv5, bit[27] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

A2.5.4 The GE[3:0] bits

In ARMv6, the SIMD instructions use bits[19:16] as *Greater than or Equal* (GE) flags for individual bytes or halfwords of the result. You can use these flags to control a later SEL instruction, see *SEL* on page A4-127 for more details.

Instructions that operate on halfwords:

- set or clear GE[3:2] together, based on the result of the top halfword calculation
- set or clear GE[1:0] together, based on the result of the bottom halfword calculation.

Instructions that operate on bytes:

- set or clear GE[3] according to the result of the top byte calculation
- set or clear GE[2] according to the result of the second byte calculation
- set or clear GE[1] according to the result of the third byte calculation
- set or clear GE[0] according to the result of the bottom byte calculation.

Each bit is set (otherwise cleared) if the results of the corresponding calculation are as follows:

- for unsigned byte addition, if the result is greater than or equal to 2^8
- for unsigned halfword addition, if the result is greater than or equal to 2^{16}
- for unsigned subtraction, if the result is greater than or equal to zero
- for signed arithmetic, if the result is greater than or equal to zero.

In architecture versions prior to ARMv6, bits[19:16] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

A2.5.5 The E bit

From ARMv6, bit[9] controls load and store endianness for data handling. See *Instructions to change CPSR E bit* on page A2-36. This bit is ignored by instruction fetches.

In architecture versions prior to ARMv6, bit[9] of the CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

A2.5.6 The interrupt disable bits

A, I, and F are the interrupt disable bits:

- A bit

Disables imprecise data aborts when it is set. This is available only in ARMv6 and above. In earlier versions, bit[8] of CPSR and SPSRs must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.
- I bit

Disables IRQ interrupts when it is set.
- F bit

Disables FIQ interrupts when it is set.

A2.5.7 The mode bits

M[4:0] are the mode bits. These determine the mode in which the processor operates. Their interpretation is shown in Table A2-2.

Table A2-2 The mode bits

M[4:0]	Mode	Accessible registers
0b10000	User	PC, R14 to R0, CPSR
0b10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
0b11111	System	PC, R14 to R0, CPSR (ARMv4 and above)

Not all combinations of the mode bits define a valid processor mode. Only those combinations explicitly described can be used. If any other value is programmed into the mode bits M[4:0], the result is UNPREDICTABLE.

r15

A2.5.8 The T and J bits

The T and J bits select the current instruction set, as shown in Table A2-3.

Table A2-3 The T and J bits

J	T	Instruction set
0	0	ARM
0	1	Thumb
1	0	Jazelle
1	1	RESERVED

we don't use

The T bit exists on T variants of ARMv4, and on all variants of ARMv5 and above. On non-T variants of ARMv4, the T bit must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

The Thumb instruction set is implemented on T variants of ARMv4 and ARMv5, and on all variants of ARMv6 and above. Instructions that switch between ARM and Thumb state execution can be used freely on implementation of these architectures.

The Thumb instruction set is not implemented on non-T variants of ARMv5. If the Thumb instruction set is selected by setting T == 1 on these architecture variants, the next instruction executed will cause an Undefined Instruction exception (see *Undefined Instruction exception* on page A2-19). Instructions that switch between ARM and Thumb state execution can be used on implementation of these architecture variants, but only function correctly as long as the program remains in ARM state. If the program attempts to switch to Thumb state, the first instruction executed after that switch causes an Undefined Instruction exception. Entry into that exception then switches back to ARM state. The exception handler can detect that this was the cause of the exception from the fact that the T bit of SPSR_und is set.

The J bit exists on ARMv5TEJ and on all variants of ARMv6 and above. On variants of ARMv4 and ARMv5, other than ARMv5TEJ, the J bit must be treated as a reserved bit, as described in *Types of PSR bits* on page A2-11.

Hardware acceleration for Jazelle opcode execution can be implemented on ARMv5TEJ and on ARMv6 and above. On these architecture variants, the BXJ instruction is used to switch from ARM state into Jazelle state when the hardware accelerator is present and enabled. If the hardware accelerator is disabled, or not present, the BXJ instruction behaves as a BX instruction, and the J bit remains clear. For more details, see *The Jazelle Extension* on page A2-53.

A2.5.9 Other bits

Other bits in the program status registers are reserved for future expansion. In general, programmers must take care to write code in such a way that these bits are never modified. Failure to do this might result in code that has unexpected side effects on future versions of the architecture. See *Types of PSR bits* on page A2-11, and the usage notes for the MSR instruction on page A4-76 for more details.

make sure
to read-
modify-
write

A2.6 Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an Undefined instruction. The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time.

The ARM architecture supports seven types of exception. Table A2-4 lists the types of exception and the processor mode that is used to process each type. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the *exception vectors*.

———— Note ————

The normal vector at address 0x00000014 and the high vector at address 0xFFFF0014 are reserved for future expansion.

no gap, implication?

Table A2-4 Exception processing modes

Exception type	Mode	VE ^a	Normal address	High vector address
Reset	Supervisor		0x00000000	0xFFFF0000
Undefined instructions	Undefined		0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor		0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort		0x0000000C	0xFFFF000C
Data Abort (data access memory abort)	Abort		0x00000010	0xFFFF0010
timer → IRQ (interrupt)	IRQ	0	0x00000018	0xFFFF0018
		1	IMPLEMENTATION DEFINED	
FIQ (fast interrupt)	FIQ	0	0x0000001C	0xFFFF001C
		1	IMPLEMENTATION DEFINED	

a. VE = vectored interrupt enable (CP15 control); RAZ when not implemented.

When an exception occurs, the banked versions of R14 and the SPSR for the exception mode are used to save state as follows:

```

R14_<exception_mode> = return link
SPSR_<exception_mode> = CPSR
CPSR[4:0] = exception mode number
CPSR[5] = 0 /* Execute in ARM state */
if <exception_mode> == Reset or FIQ then
    CPSR[6] = 1 /* Disable fast interrupts */
/* else CPSR[6] is unchanged */
CPSR[7] = 1 /* Disable normal interrupts */
if <exception_mode> != UNDEF or SWI then
    CPSR[8] = 1 /* Disable imprecise aborts (v6 only) */
/* else CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
PC = exception vector address
    
```

old state so we can use

To return after handling the exception, the SPSR is moved into the CPSR, and R14 is moved to the PC. This can be done atomically in two ways:

- using a data-processing instruction with the S bit set, and the PC as the destination
- using the Load Multiple with Restore CPSR instruction, as described in *LDM (3)* on page A4-40.

In addition, in ARMv6, the RFE instruction (see *RFE* on page A4-113) can be used to load the CPSR and PC from memory, so atomically returning from an exception to a PC and CPSR that was previously saved in memory.

Collectively these mechanisms define all of the mechanisms which perform a return from exception.

The following sections show what happens automatically when the exception occurs, and also show the recommended data-processing instruction to use to return from each exception. This instruction is always a MOVs or SUBS instruction with the PC as its destination.

Note

When the recommended data-processing instruction is a SUBS and a Load Multiple with Restore CPSR instruction is used to return from the exception handler, the subtraction must still be performed. This is usually done at the start of the exception handler, before the return link is stored to memory.

For example, an interrupt handler that wishes to store its return link on the stack might use instructions of the following form at its entry point:

```

SUB    R14, R14, #4
STMFD  SP!, {<other_registers>, R14}
    
```

and return using the instruction:

```

LDMFD  SP!, {<other_registers>, PC}^
    
```

A2.6.1 ARMv6 extensions to the exception model

In ARMv6 and above, the exception model is extended as follows:

- An imprecise data abort mechanism that allows some types of data abort to be treated asynchronously. The resulting exceptions behave like interrupts, except that they use Abort mode and its banked registers. This mechanism includes a mask bit (the A bit) in the PSRs, in order to ensure that imprecise data aborts do not occur while another abort is being handled. The mechanism is described in *Imprecise data aborts* on page A2-23.
- Support for vectored interrupts controlled by the VE bit in the system control coprocessor (see *Vectored interrupt support* on page A2-26). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.
- Support for a low interrupt latency configuration controlled by the FI bit in the system control coprocessor (see *Low interrupt latency configuration* on page A2-27). It is IMPLEMENTATION DEFINED whether support for this mechanism is included in earlier versions of the architecture.
- Three new instructions (CPS, SRS, RFE) to improve nested stack handling of different exceptions in a common mode. CPS can also be used to efficiently enable or disable the interrupt and imprecise abort masks, either within a mode, or while transitioning from a privileged mode to any other mode. See *New instructions to improve exception handling* on page A2-28 for a brief description.

A2.6.2 Reset

When the Reset input is asserted on the processor, the ARM processor immediately stops execution of the current instruction. When Reset is de-asserted, the following actions are performed:

```

R14_svc = UNPREDICTABLE value
SPSR_svc = UNPREDICTABLE value
CPSR[4:0] = 0b10011          /* Enter Supervisor mode */
CPSR[5]   = 0                /* Execute in ARM state */
CPSR[6]   = 1                /* Disable fast interrupts */
CPSR[7]   = 1                /* Disable normal interrupts */
CPSR[8]   = 1                /* Disable Imprecise Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit   /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0000
else
    PC = 0x00000000
    
```

After Reset, the ARM processor begins execution at address 0x00000000 or 0xFFFF0000 in Supervisor mode with interrupts disabled.

————— **Note** —————

There is no architecturally defined way of returning from a Reset.

A2.6.3 Undefined Instruction exception

If the ARM processor executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an Undefined Instruction exception occurs.

If an attempt is made to execute an instruction that is UNDEFINED, an Undefined Instruction exception occurs (see *Extending the instruction set* on page A3-32).

The Undefined Instruction exception can be used for software emulation of a coprocessor in a system that does not have the physical coprocessor (hardware), or for general-purpose instruction set extension by software emulation.

When an Undefined Instruction exception occurs, the following actions are performed:

```
R14_und = address of next instruction after the Undefined instruction
SPSR_und = CPSR
CPSR[4:0] = 0b11011 /* Enter Undefined Instruction mode */
CPSR[5] = 0 /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1 /* Disable normal interrupts */
/* CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0004
else
    PC = 0x00000004
```

To return after emulating the Undefined instruction use:

```
MOVSPC PC, R14
```

This restores the PC (from R14_und) and CPSR (from SPSR_und) and returns to the instruction following the Undefined instruction.

In some coprocessor designs, an internal exceptional condition caused by one coprocessor instruction is signaled *imprecisely* by refusing to respond to a later coprocessor instruction. In these circumstances, the Undefined Instruction handler takes whatever action is necessary to clear the exceptional condition, then returns to the second coprocessor instruction. To do this use:

```
SUBSPC PC, R14, #4
```

A2.6.4 Software Interrupt exception

The Software Interrupt instruction (SWI) enters Supervisor mode to request a particular supervisor (operating system) function. When a SWI is executed, the following actions are performed:

```

R14_svc = address of next instruction after the SWI instruction
SPSR_svc = CPSR
CPSR[4:0] = 0b10011 /* Enter Supervisor mode */
CPSR[5] = 0 /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1 /* Disable normal interrupts */
/* CPSR[8] is unchanged */
CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF0008
else
    PC = 0x00000008
    
```

To return after performing the SWI operation, use the following instruction to restore the PC (from R14_svc) and CPSR (from SPSR_svc) and return to the instruction following the SWI:

```

MOVN PC,R14
    
```

A2.6.5 Prefetch Abort (instruction fetch memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. A Prefetch Abort exception is generated if the processor tries to execute the invalid instruction. If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no Prefetch Abort occurs.

In ARMv5 and above, a Prefetch Abort exception can also be generated as the result of executing a BKPT instruction. For details, see *BKPT* on page A4-14 (ARM instruction) and *BKPT* on page A7-24 (Thumb instruction).

When an attempt is made to execute an aborted instruction, the following actions are performed:

```

R14_abt = address of the aborted instruction + 4
SPSR_abt = CPSR
CPSR[4:0] = 0b10111 /* Enter Abort mode */
CPSR[5] = 0 /* Execute in ARM state */
/* CPSR[6] is unchanged */
CPSR[7] = 1 /* Disable normal interrupts */
CPSR[8] = 1 /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit /* Endianness on exception entry */
if high vectors configured then
    PC = 0xFFFF000C
else
    PC = 0x0000000C
    
```

To return after fixing the reason for the abort, use:

```
SUBS PC,R14,#4
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to the aborted instruction.

A2.6.6 Data Abort (data access memory abort)

A memory abort is signaled by the memory system. Activating an abort in response to a data access (load or store) marks the data as invalid. A Data Abort exception occurs before any following instructions or exceptions have altered the state of the CPU. The following actions are performed:

```
R14_abt  = address of the aborted instruction + 8
SPSR_abt = CPSR
CPSR[4:0] = 0b10111          /* Enter Abort mode */
CPSR[5]   = 0                /* Execute in ARM state */
                                /* CPSR[6] is unchanged */
CPSR[7]   = 1                /* Disable normal interrupts */
CPSR[8]   = 1                /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit  /* Endianness on exception entry */
if high vectors configured then
    PC     = 0xFFFF0010
else
    PC     = 0x00000010
```

To return after fixing the reason for the abort use:

```
SUBS PC,R14,#8
```

This restores both the PC (from R14_abt) and CPSR (from SPSR_abt), and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed use:

```
SUBS PC,R14,#4
```

Effects of data-aborted instructions

Instructions that access data memory can modify memory by storing one or more values. If a Data Abort occurs in such an instruction, the value of each memory location that the instruction stores to is:

- unchanged if the memory system does not permit write access to the memory location
- UNPREDICTABLE otherwise.

Instructions that access data memory can modify registers in the following ways:

- By loading values into one or more of the general-purpose registers, that can include the PC.
- By specifying *base register write-back*, in which the base register used in the address calculation has a modified value written to it. All instructions that allow this to be specified have UNPREDICTABLE results if base register write-back is specified and the base register is the PC, so only general-purpose registers other than the PC can legitimately be modified in this way.
- By loading values into coprocessor registers.
- By modifying the CPSR.

If a Data Abort occurs, the values left in these registers are determined by the following rules:

1. The PC value on entry to the Data Abort handler is 0x00000010 or 0xFFFF0010, and the R14_abt value is determined from the address of the aborted instruction. Neither is affected in any way by the results of any PC load specified by the instruction.
2. If base register write-back is not specified, the base register value is unchanged. This applies even if the instruction loaded its own base register and the memory access to load the base register occurred earlier than the aborting access.

For example, suppose the instruction is:

```
LDMIA R0, {R0,R1,R2}
```

and the implementation loads the new R0 value, then the new R1 value and finally the new R2 value. If a Data Abort occurs on any of the accesses, the value in the base register R0 of the instruction is unchanged. This applies even if it was the load of R1 or R2 that aborted, rather than the load of R0.

3. If base register write-back is specified, the value left in the base register is determined by the *abort model* of the implementation, as described in *Abort models* on page A2-23.
4. If the instruction only loads one general-purpose register, the value in that register is unchanged.
5. If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in destination registers that are neither the PC nor the base register of the instruction.
6. If the instruction loads coprocessor registers, UNPREDICTABLE values are left in the destination coprocessor registers, unless otherwise specified in the instruction set description of the specific coprocessor.
7. CPSR bits not defined as updated on exception entry maintain their current value.

Abort models

The abort model used by an ARM implementation is IMPLEMENTATION DEFINED, and is one of the following:

Base Restored Abort Model

If a precise Data Abort occurs in an instruction that specifies base register write-back, the value in the base register is unchanged. This is the only abort model permitted in ARMv6 and above.

Base Updated Abort Model

If a precise Data Abort occurs in an instruction that specifies base register write-back, the base register write-back still occurs. This model is prohibited in ARMv6 and above.

In either case, the abort model applies uniformly across all instructions. An implementation does not use the Base Restored Abort Model for some instructions and the Base Updated Abort Model for others.

A2.6.7 Imprecise data aborts

An imprecise data abort, caused, for example, by an external error on a write that has been held in a Write Buffer, is asynchronous to the execution of the causing instruction and might in reality occur many cycles after the instruction that caused the memory access has retired. For this reason, the imprecise data abort might occur at a time that the processor is in abort mode because of a precise abort, or might have live state in abort mode, but be handling an interrupt.

To avoid the loss of the Abort mode state (R14 and SPSR_abt) in these cases, that would lead to the processor entering an unrecoverable state, the existence of a pending imprecise data abort must be held by the system until such time as the abort mode can safely be entered.

From ARMv6, a mask is added into the CPSR (CPSR[8]) to control when an imprecise abort cannot be accepted. This bit is referred to as the A bit. The imprecise data abort causes a Data Abort to be taken when imprecise data aborts are not masked. When imprecise data aborts are masked, the implementation is responsible for holding the presence of a pending imprecise abort until the mask is cleared and the abort is taken. It is IMPLEMENTATION DEFINED whether more than one imprecise abort can be pending.

The A bit is set automatically on taking a Prefetch Abort, a Data Abort, an IRQ or FIQ interrupt, and on reset.

The A bit can only be changed from a privileged mode.

A2.6.8 Interrupt request (IRQ) exception

The IRQ exception is generated externally by asserting the IRQ input on the processor. It has a lower priority than FIQ (see Table A2-1 on page A2-25), and is masked out when an FIQ sequence is entered.

Interrupts are disabled when the I bit in the CPSR is set. If the I bit is clear, ARM checks for an IRQ at instruction boundaries.

————— **Note** —————

The I bit can only be changed from a privileged mode.

When an IRQ is detected, the following actions are performed:

```

R14_irq = address of next instruction to be executed + 4
SPSR_irq = CPSR
CPSR[4:0] = 0b10010          /* Enter IRQ mode */
CPSR[5] = 0                  /* Execute in ARM state */
                                /* CPSR[6] is unchanged */
CPSR[7] = 1                  /* Disable normal interrupts */
CPSR[8] = 1                  /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9] = CP15_reg1_EEbit    /* Endianness on exception entry */
if VE==0 then
    if high vectors configured then
        PC = 0xFFFFF0018
    else
        PC = 0x00000018
else
    PC = IMPLEMENTATION DEFINED /* see page A2-26 */
    
```

To return after servicing the interrupt, use:

```
SUBS PC,R14,#4
```

This restores both the PC (from R14_irq) and CPSR (from SPSR_irq), and resumes execution of the interrupted code.

A2.6.9 Fast interrupt request (FIQ) exception

The FIQ exception is generated externally by asserting the FIQ input on the processor. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching.

Fast interrupts are disabled when the F bit in the CPSR is set. If the F bit is clear, ARM checks for an FIQ at instruction boundaries.

————— **Note** —————

The F bit can only be changed from a privileged mode.

When an FIQ is detected, the following actions are performed:


```

R14_fiq  = address of next instruction to be executed + 4
SPSR_fiq = CPSR
CPSR[4:0] = 0b10001          /* Enter FIQ mode */
CPSR[5]   = 0                /* Execute in ARM state */
CPSR[6]   = 1                /* Disable fast interrupts */
CPSR[7]   = 1                /* Disable normal interrupts */
CPSR[8]   = 1                /* Disable Imprecise Data Aborts (v6 only) */
CPSR[9]   = CP15_reg1_EEbit  /* Endianness on exception entry */
if VE==0 then
    if high vectors configured then
        PC = 0xFFFF001C
    else
        PC = 0x0000001C
else
    PC = IMPLEMENTATION DEFINED /* see page A2-26 */

```

To return after servicing the interrupt, use:

```

SUBS PC, R14,#4

```

This restores both the PC (from R14_fiq) and CPSR (from SPSR_fiq), and resumes execution of the interrupted code.

The FIQ vector is deliberately the last vector to allow the FIQ exception-handler software to be placed directly at address 0x0000001C or 0xFFFF001C, without requiring a branch instruction from the vector.

A2.6.10 Exception priorities

Table A2-1 shows the exception priorities:

Table A2-1 Exception priorities

Priority		Exception
Highest	1	Reset
	2	Data Abort (including data TLB miss)
	3	FIQ
	4	IRQ
	5	Imprecise Abort (external abort) - ARMv6
	6	Prefetch Abort (including prefetch TLB miss)
Lowest	7	Undefined instruction SWI

Undefined instruction and software interrupt cannot occur at the same time, because they each correspond to particular (non-overlapping) decodings of the current instruction. Both must be lower priority than Prefetch Abort, because a Prefetch Abort indicates that no valid instruction was fetched.

The priority of a Data Abort exception is higher than FIQ, which ensures that the Data Abort handler is entered before the FIQ handler is entered (so that the Data Abort is resolved after the FIQ handler has completed).

A2.6.11 High vectors

High vectors were introduced into some implementations of ARMv4 and are required in ARMv6 implementations. High vectors allow the exception vector locations to be moved from their normal address range 0x00000000-0x0000001C at the bottom of the 32-bit address space, to an alternative address range 0xFFFF0000-0xFFFF001C near the top of the address space. These alternative locations are known as the *high vectors*.

Prior to ARMv6, it is IMPLEMENTATION DEFINED whether the high vectors are supported. When they are, a hardware configuration input selects whether the normal vectors or the high vectors are to be used from reset.

The ARM instruction set does not contain any instructions that can directly change whether normal or high vectors are configured. However, if the standard System Control coprocessor is attached to an ARM processor that supports the high vectors, bit[13] of coprocessor 15 register 1 can be used to switch between using the normal vectors and the high vectors (see *Register 1: Control registers* on page B3-12).

A2.6.12 Vectored interrupt support

Historically, the IRQ and FIQ exception vectors are affected by whether high vectors are enabled, and are otherwise fixed. The result is that interrupt handlers typically have to start with an instruction sequence to determine the cause of the interrupt and branch to a routine to handle it. Support of vectored interrupts allows an interrupt controller to prioritize interrupts, and provide the required interrupt handler address directly to the core. The vectored interrupt behavior is explicitly enabled by the setting of a bit, the VE bit, in the system coprocessor CP15 register 1. See *Register 1: Control registers* on page B3-12. For backwards compatibility, the vectored interrupt mechanism is disabled on reset. The details of the hardware to support vectored interrupts is IMPLEMENTATION DEFINED.

A vectored interrupt controller (VIC) can reduce effective interrupt latency considerably, by eliminating the need for an interrupt handler to identify the source of an interrupt and acknowledge it before re-enabling the interrupts. Furthermore, if the VIC and core implement an appropriate handshake as the interrupt handler routine is entered, the VIC can automatically mask out the interrupt source associated with that handler and any lower priority sources. This allows the interrupts concerned to be re-enabled by the processor core as soon as their return information (that is, R14 and SPSR values) have been saved, reducing the period during which higher priority interrupts are disabled.

A2.6.13 Low interrupt latency configuration

The FI bit (bit[21]) in the system control register (CP15 register 1) enables the interrupt latency configuration logic in an implementation. See *Register 1: Control registers* on page B3-12. The purpose of this configuration is to reduce the interrupt latency of the processor. The exact mechanisms that are used to perform this are IMPLEMENTATION DEFINED.

In order to ensure that a change between normal and low interrupt latency configurations is synchronized correctly, the FI bit must only be changed in IMPLEMENTATION DEFINED circumstances. It is recommended that software systems should only change the FI bit shortly after reset, while interrupts are disabled.

When interrupt latency is reduced, this may result in reduced performance overall. Examples of the mechanisms which may be used are disabling Hit-Under-Miss functionality within a core, and the abandoning of restartable external accesses, allowing the core to react to a pending interrupt faster than would otherwise be the case. Low interrupt latency configuration may have IMPLEMENTATION DEFINED effects in the memory system or elsewhere outside the processor core. It is legal for the interrupt to be seen as being taken before a store to a restartable memory location, but for the memory to have been updated when in low interrupt latency configuration.

In low interrupt latency configuration, software must only use multi-word load/store instructions in ways that are fully restartable. This allows (but does not require) implementations to make multi-word instructions interruptible when in low interrupt latency configuration. The multi-access instructions to which this rule currently applies are:

ARM	LDC, all forms of LDM, LDRD, STC, all forms of STM, STRD
Thumb	LDMIA, PUSH, POP, STMIA

————— **Note** —————

If the instruction is interrupted before it is complete, the result may be that one or more of the words are accessed twice. Idempotent memory (multiple reads or writes of the same information exhibit identical system results) is a requirement of system correctness.

In ARMv6, memory with the normal attribute is guaranteed to behave this way, however, memory marked as Device or Strongly Ordered is not (for example, a FIFO). It is IMPLEMENTATION DEFINED whether multi-word accesses are supported for Device and Strongly Ordered memory types in the low interrupt latency configuration.

A similar situation exists with regard to multi-word load/store instructions that access memory locations that can abort in a recoverable way, since an abort on one of the words accessed may cause a previously-accessed word to be accessed twice – once before the abort, and a second time after the abort handler has returned. The requirement in this case is either that all side-effects are idempotent, or that the abort must either occur on the first word accessed or not at all.

A2.6.14 New instructions to improve exception handling

ARMv6 adds an instruction to simplify changes of processor mode and the disabling and enabling of interrupts. New instructions are also added to reduce the processing cost of handling exceptions in a different mode to the exception entry mode, by removing any need to use the original mode's stack. Two examples are:

- IRQ routines may wish to execute in System or Supervisor mode, so that they can both re-enable IRQs and use BL instructions. This is not possible in IRQ mode, because a nested IRQ could corrupt the BL's return link at any time. Using the new instructions, the system can store the return state (R14 link register and SPSR_irq) to the System/User or Supervisor mode stack, switch to System or Supervisor mode and re-enable IRQs efficiently, without making any use of R13_irq or the IRQ stack.
- FIQ mode is designed for efficient use by a single owner, using R8_fiq – R13_fiq as global variables. In addition, unlike IRQs, FIQs are not disabled by other exceptions (apart from reset), making them the preferred type for real time interrupts, when other exceptions are being used routinely, such as virtual memory or instruction emulation. IRQs may be disabled for unacceptably long periods of time while these needs are being serviced.

However, if more than one real-time interrupt source is required, there is a conflict of interest. The new mechanism allows multiple FIQ sources and minimizes the period with FIQs disabled, greatly reducing the interrupt latency penalty. The FIQ mode registers can be allocated to the highest priority FIQ as a single owner.

SRS – Store Return State

This instruction stores R14_<current_mode> and SPSR_<current_mode> to sequential addresses, using the banked version of R13 for a specified mode to supply the base address (and to be written back to if base register writeback is specified). This allows an exception handler to store its return state on a stack other than the one automatically selected by its exception entry sequence.

The addressing mode used is a version of ARM addressing mode 4 (see *Addressing Mode 4 - Load and Store Multiple* on page A5-41), modified so as to assume a {R14,SPSR} register list rather than using a list specified by a bit mask in the instruction. This allows the SRS instruction to access stacks in a manner compatible with the normal use of STM instructions for stack accesses. See SRS on page A4-174 for the instruction details.

RFE – Return From Exception

This instruction loads the PC and CPSR from sequential addresses. This is used to return from an exception which has had its return state saved using the SRS instruction, and again uses a version of ARM addressing mode 4, modified this time to assume a {PC,CPSR} register list. See RFE on page A4-113 for the instruction details.

CPS – Change Processor State

This instruction provides new values for the CPSR interrupt masks, mode bits, or both, and is designed to shorten and speed up the read/modify/write instruction sequence used in earlier architecture variants to perform such tasks. Together with the SRS instruction, it allows an exception handler to save its return information on the stack of another mode and then switch to that other mode, without modifying the stack belonging to the original mode or any registers other than the stack pointer of the new mode.

The instruction also streamlines interrupt mask handling and mode switches in other code, and in particular allows short, efficient, atomic code sequences in a uniprocessor system by disabling interrupts at their start and re-enabling interrupts at their end. See *CPS* on page A4-29 for the instruction details.

A CPS Thumb instruction that allows mask updates within the current mode is also provided, see section *CPS* on page A7-39.

Note

The Thumb instruction cannot change the mode due to instruction space usage constraints.

A2.7 Endian support

This section discusses memory and memory-mapped I/O, with regard to the assumptions ARM processor implementations make about endianness.

ARMv6 introduces several architectural extensions to support mixed-endian access in hardware:

- Byte reverse instructions that operate on general-purpose register contents to support word, and signed and unsigned halfword data quantities.
- Separate instruction and data endianness, with instructions fixed as little-endian format, naturally aligned, but with legacy support for 32-bit word-invariant binary images/ROM.
- A PSR Endian control flag, the E bit, which dictates the byte order used for the entire load and store instruction space when data is loaded into, and stored back out of the register file. In previous architectures this PSR bit was specified as 0 and is never set in legacy code written to conform to architectures prior to ARMv6.
- ARM and Thumb instructions to set and clear the E bit explicitly.
- A byte-invariant addressing scheme to support fine-grain big-endian and little-endian shared data structures, to conform to the *IEEE Standard for Shared-Data Formats Optimized for Scalable Coherent Interface (SCI) Processors*, IEEE Std 1596.5-1993 (ISBN 1-55937-354-7, IEEE).
- Bus interface endianness is IMPLEMENTATION DEFINED. However, it must support byte lane controls for unaligned word and halfword data access.

A2.7.1 Address space

The ARM architecture uses a single, flat address space of 2^{32} 8-bit bytes. Byte addresses are treated as unsigned numbers, running from 0 to $2^{32} - 1$.

This address space is regarded as consisting of 2^{30} 32-bit words, each of whose addresses is word-aligned, which means that the address is divisible by 4. The word whose word-aligned address is A consists of the four bytes with addresses A, A+1, A+2 and A+3.

In ARMv4 and above, the address space is also regarded as consisting of 2^{31} 16-bit halfwords, each of whose addresses is halfword-aligned (divisible by 2). The halfword whose halfword-aligned address is A consists of the two bytes with addresses A and A+1.

In ARMv5E and above, the address space supports 64-bit doubleword operations. Doubleword operations can be considered as two-word load/store operations, each word addressed as follows:

- A, A+1, A+2, and A+3 for the first word
- A+4, A+5, A+6, and A+7 for the second word.

Prior to ARMv6, word-aligned doubleword operations are UNPREDICTABLE with doubleword-aligned addresses always supported. ARMv6 mandates support of both modulo4 and modulo8 alignment of doublewords, and introduces support for unaligned word and halfword data accesses, all controlled through the standard System Control coprocessor.