ARM 810

Preliminary Data Sheet

ENGLAND
Advanced RISC Machines Limited
90 Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
UK
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
Email: info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone: +49 89 608 75545
Facsimile: +49 89 608 75599
Email: info@armltd.co.uk

JAPAN
Advanced RISC Machines K.K.
KSP West Bldg, 3F 300D, 3-2-1 Sakado
Takatsu-ku, Kawasaki-shi
Kanagawa
213 Japan
Telephone: +81 44 850 1301
Facsimile: +81 44 850 1308
Email: info@armltd.co.uk

USA
ARM USA Incorporated
Suite 5
985 University Avenue
Los Gatos
CA 95030 USA
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

World Wide Web address: http://www.arm.com

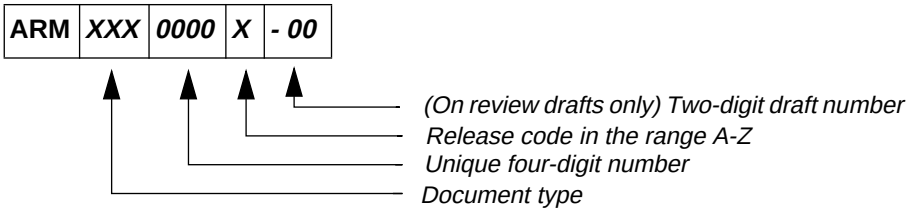
ARM
Advanced RISC Machines

Proprietary Notice

Key

Document Number

This document has a number which identifies it uniquely. The number is displayed on the front page and at the foot of each subsequent page.



Document Status

The document's status is displayed in a banner at the bottom of each page. This describes the document's confidentiality and its information status.

Confidentiality status is one of:

ARM Confidential	Distributable to ARM staff and NDA signatories only
Named Partner Confidential	Distributable to the above and to the staff of named partner companies only
Partner Confidential	Distributable within ARM and to staff of all partner companies
Open Access	No restriction on distribution

Information status is one of:

Advance	Information on a potential product
Preliminary	Current information on a product under development
Final	Complete information on a developed product

Change Log

Issue	Date	By	Change
A (Draft)	Mar 1995	EH	Created.
B (Draft)		Aug 1995	SFKEdits, in particular Bus Interface.
C	Oct 1995	AP	Clocking additions.
D	Mar 1996	EH	Appendix F removed. Status changed to Open Access.
E	August 1996	KTB	Corrections and amendments

Contents

Contents

Contents

Contents

1 Overview

This chapter provides an overview of the ARM810.

Overview

1.1 Introduction

The ARM810 is a general purpose 32-bit microprocessor with 8KB unified cache, write buffer and Memory Management Unit (MMU) combined in a single chip. The CPU within ARM810 is the ARM8 core designed to ARM architecture version 4.0. The ARM810 is software compatible with earlier ARM architectures and can be used with ARM Ltd's support chips.

The architecture of the ARM810 is based on Reduced Instruction Set Computer (RISC) principles, and its instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers.

1.2 The ARM810 Architecture

The block diagram for the ARM810 core is shown in ***Figure 1-1: ARM810 block diagram*** on page 1-3. The major blocks of the ARM810 are the ARM8 CPU, Memory Management Unit (MMU), Write Buffer (WB), Coprocessor15 (CP15), 8KB cache and external Bus Interface Unit (BIU).

The ARM8 CPU core combines an instruction Prefetch Unit (PU) with a four stage instruction and data pipeline to make a CPU with a five stage pipeline. This ensures that all parts of the processing and memory systems can operate continuously and the performance of each stage can be maximised allowing the use of very high clock rates. The processor implements static branch prediction in the PU which predicts whether or not a branch will be taken depending on the branch destination address. If the branch instruction is predicted as being taken then further instructions are fetched from the branch destination and the branch is removed. If it is predicted that a branch is not taken then the instruction is removed from the instruction pipeline.

Double-bandwidth reads to the on-chip memory reduces the average Load multiple CPI value by 1.5 when compared to a single bandwidth architecture running the same code. The single register Load and Store instructions require only one cycle for the normal cases. (This assumes cache hit and write buffer not full.)

The processor has an on-chip unified instruction and data cache supporting write-back and write-through operation. Together with the write buffer, this substantially raises the average execution speed and reduces the average amount of memory bandwidth required by the processor when compared to a processor without these features. This allows the external memory to support additional processors or Direct Memory Access (DMA) channels with minimal performance loss and results in low external memory power consumption. In addition the cache also has a lock down capability that can ensure that critical code such as interrupt routines can be locked into the cache to ensure low execution latency.

The MMU supports a conventional two-level page-table structure and a number of extensions which make it ideal for embedded control, UNIX and Object Oriented systems.

The Bus Interface has been designed to allow the performance potential to be realised without incurring high costs in the memory system. Speed critical control signals are pipelined to allow system control functions to be implemented in standard low power logic. These control signals allow the exploitation of page mode accesses offered by industry standard DRAMs.

The ARM810 is a fully static device designed to minimise power requirements. This makes it ideal for portable applications where both of these features are essential.

1.3 The Instruction Set

The ARM810 uses the ARM architecture version 4.0 instruction set, which comprises eleven basic instruction types:

- Two perform high-speed operations on data in a bank of thirty one 32-bit registers, using the on-chip arithmetic logic unit, shifter and multiplier.
- Three control data transfer between the registers and memory, one optimised for flexibility of addressing, another for rapid context switching and the third for swapping data.
- Three adjust the flow, privilege level and system control parameters of execution.
- Three are dedicated to the control of coprocessors. However ARM810 implements only MCR and MRC, which allow access to the ARM810 system control coprocessor, CP15.

The ARM instruction set is a good target for compilers of many different high-level languages, and is also straightforward to use when programming in assembly language - unlike the instruction sets of some RISC processors which rely on sophisticated compiler technology to manage complicated instruction interdependencies.

All existing code compiled for previous ARM processors will run on ARM810 with only rare exceptions.

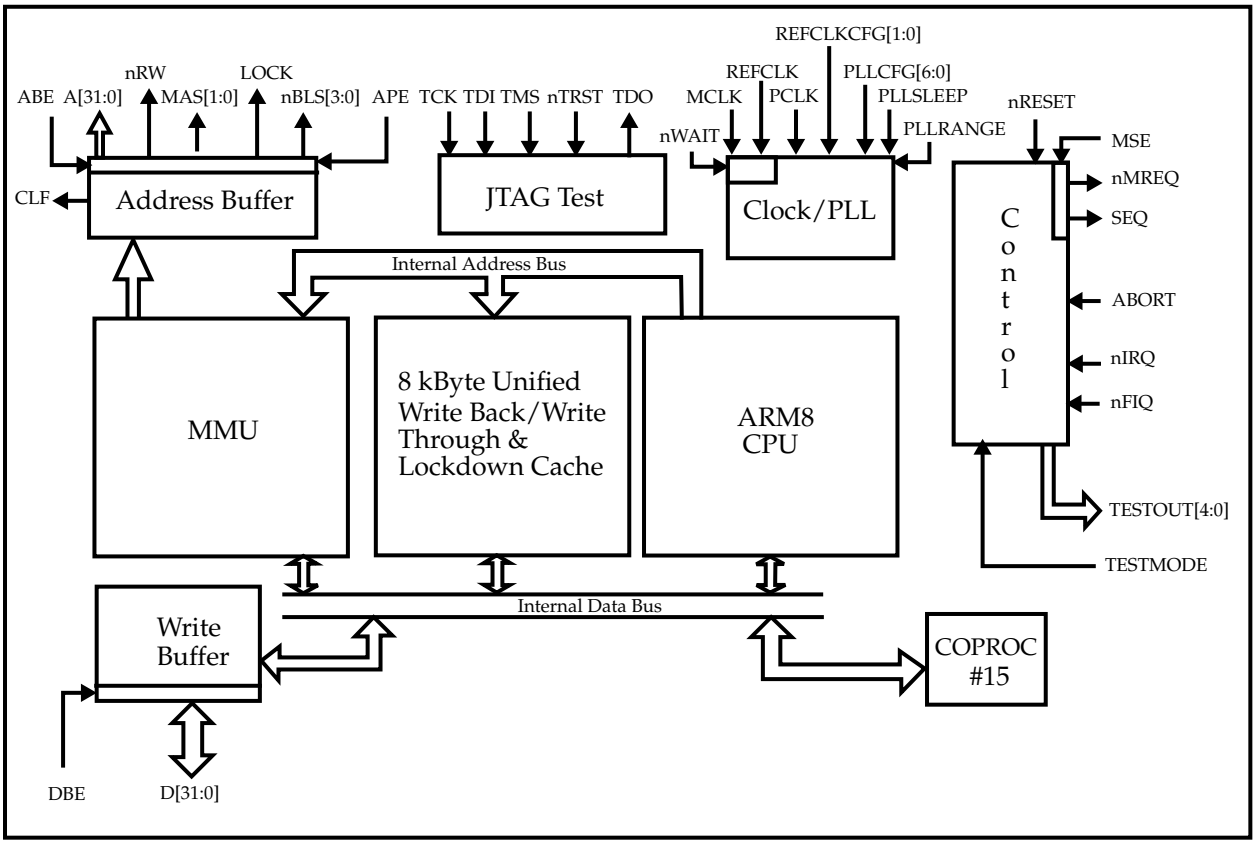


Figure 1-1: ARM810 block diagram

2 Signal Description

This chapter documents the ARM810 signals.

Signal Description

2.1 Functional Diagram

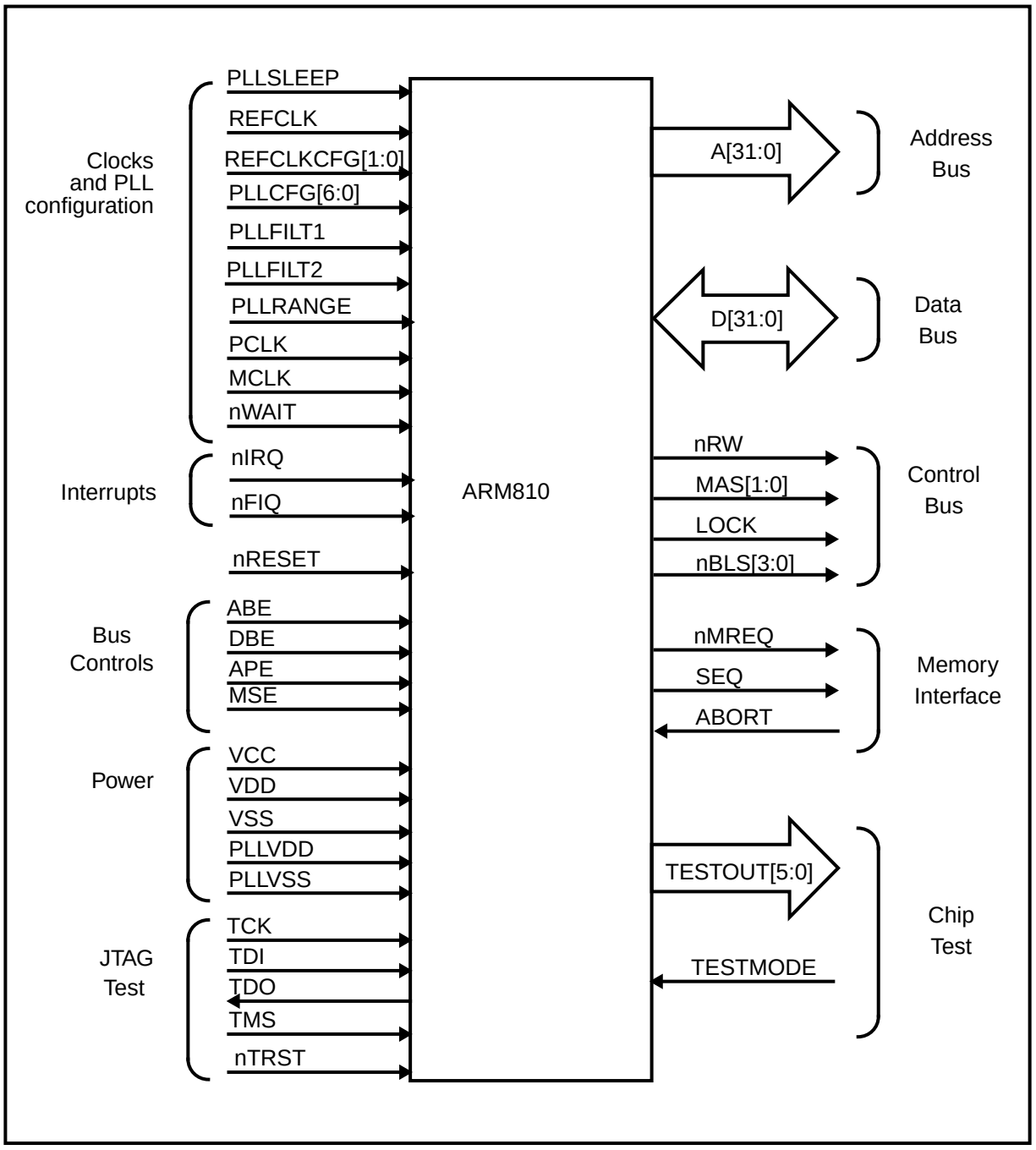


Figure 2-1: ARM810 functional diagram

2.2 Signals

Name	Type	Description:
A[31:0]	OCZ	Address Bus. This bus signals the address requested for memory accesses. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
ABE	I	Address bus enable. When this input is LOW, the address bus **A[31:0]**, **MAS[1:0]**, **CLF**, **nBLS[3:0]**, **nRW** and **LOCK** are put into a high impedance state (Note 1).
ABORT	I	External abort. Allows the memory system to tell the processor that a requested access has failed. Only monitored when ARM810 is accessing external memory.
APE	I	Address pipeline enable control input. When **APE** is HIGH, address and address-timed outputs are generated with normal pipeliined timing, where a new address is generated in the second phase of the bus clock (**MCLK** HIGH or **PCLK** LOW). Taking **APE** LOW delays these signals by one clock phase so they change in the first phase of the following bus cycle (**MCLK** LOW or **PCLK** HIGH). See the descriptions for **MCLK**/**PCLK** and ***Chapter 11, ARM810 Clocking*** for bus clock information. The address-timed signals are **A[31:0]**, **MAS[1:0]**, **nBLS[3:0**], **CLF**, **LOCK** and **nRW**.
CLF	O	Cache line fill. **CLF** HIGH indicates that the current read cycle is cacheable. **CLF** is always HIGH for writes. This signal may be used to indicate to a second level cache controller that a read is cacheable in the second level cache (if present).
D[31:0]	IOCZ	Data bus. These are bi-directional signal paths used for data transfers between the processor and external memory. For read operations (when **nRW** is LOW), the input data must be valid before the falling edge of **MCLK**. For write operations (when **nRW** is HIGH), the output data will become valid while **MCLK** is LOW. At high clock frequencies the data may not become valid until just after the **MCLK** rising edge.
DBE	I	Data bus enable. When this input is LOW, the data bus, **D[31:0]** is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and **DBE** must be driven HIGH in systems which do not require the data bus for DMA or similar activities.
LOCK	OCZ	Locked operation. **LOCK** is driven HIGH, to signal a “locked” memory access sequence, and the memory manager should wait until **LOCK** goes LOW before allowing another device to access the memory. **LOCK** remains HIGH during the locked memory sequence. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
MCLK	I	This is a bus clock input. Bus cycles start and end with falling edges of **MCLK**. Hold **PCLK** HIGH to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further details. This signal is provided for backwards compatibility with previous processors, see **PCLK** for the preferred bus clock input.

Table 2-1: Signal descriptions

Name	Type	Description:
MSE	I	Memory request/sequential enable. When this input is LOW, the **nMREQ** and **SEQ** outputs are put into a high impedance state (Note 1).
MAS[1:0]	OCZ	Memory Access Size. An output bus used by the processor to indicate the size of the next data transfer to the external memory system as being a byte, half word or full 32 bit word in length. MAS[1:0] is valid for both read and write operations**.** Normally it changes during phase 2 of the bus clock.The timing can be changed using **APE**.
nBLS[3:0]	OCZ	Not Byte Lane Selects. These signify which bytes of the memory are being accessed. For a word access all will be LOW. Normally they change during phase 2 of the bus clock. The timing can be changed using **APE**.
nFIQ	I	Not fast interrupt request. If FIQs are enabled, the processor will respond to a LOW level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input to guarantee that the interrupt has been taken.,
nIRQ	I	Not interrupt request. As **nFIQ**, but with lower priority. If IRQs are enabled, the processor will respond to a low level on this signal by taking the IRQ interrupt exception.
nMREQ	OCZ	Not memory request. A pipelined signal that changes while **MCLK** is LOW to indicate whether or not in the following cycle, the processor will be accessing external memory. When **nMREQ** is LOW, the processor will be accessing external memory in the next bus cycle.
nRESET	I	Not reset. This is a level sensitive input which is used to start the processor from a known address. A LOW level will cause the current instruction to terminate abnormally, and the on-chip cache, MMU, and write buffer to be disabled. When **nRESET** is driven HIGH, the processor will re-start from address 0. **nRESET** must remain LOW for at least 5 full fast clock cycles or 5 full bus clock cycles whichever is greater. While **nRESET** is LOW the processor will perform idle cycles and **nWAIT** must be HIGH.
nRW	OCZ	Not read/write. When HIGH this signal indicates a processor write operation; when LOW, a read. Normally it changes during phase 2 of the bus clock. The timing can be changed using **APE**.
nTRST	I	Test interface reset. Note this signal does NOT have an internal pullup resistor. This signal must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (**nRESET**).
nWAIT	I	Not wait. When LOW this allows extra **MCLK** cycles to be inserted in memory accesses. It must change during the LOW phase of the **MCLK** cycle to be extended.

Table 2-1: Signal descriptions (Continued)

Name	Type	Description:
PCLK	I	This is an inverted bus clock input. Bus cycles start and end with rising edges of **PCLK**. Hold **MCLK** LOW to use this clock input. See ***11.1.1 External input clock: MCLK or PCLK*** on page 11-3 for further information. We recommend using this bus clock input for compatibility with the new generations of synchronous memory systems (SSRAM, SDRAM) and future ARM microprocessors. The **MCLK** input is provided for compatibility with earlier ARM processors.
PLLCFG[6:0]	I	Phase locked loop configuration input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details.
PLLFILT1		Analog filter pin for PLL.
PLLFILT2		Analog filter fast start pin for PLL.
PLLRANGE	IOCZ	In normal operation, an input which selects the **PLL** output frequency range. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. This pin is also used as an output when the device is in some test modes. The output driver is guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
PLLSLEEP	I	When HIGH, this puts the **PLL** into low power sleep mode. Please refer to ***11.5 Low Power Idle and Sleep*** on page 11-10 for further details.
PLLVDD		VDD supply for analog components in PLL. 1 pin. Should be appropriately isolated from digital noise on supply.
PLLVSS		Ground supply for analog components in PLL. 1 pin.
REFCLK	I	Clock input which is divided by the prescaler to provide the **PLL** reference clock. **REFCLK** can also be configured to a direct source of the internal fast clock, bypassing the **PLL**. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 and ***11.3.3 Fast clock direct (bypassing the PLL)*** on page 11-8 for further details.
REFCLKCFG[1:0]	IOCZ	In normal operation, an input which selects the divide ratio for the **PLL** reference clock prescaler on the **REFCLK** input. Please refer to ***11.3.2 Fast clock from the output of the PLL*** on page 11-7 for further details. These pins are also used as an output when the device is in some test modes. The output drivers are guaranteed to be high-impedance if the **TESTMODE** pin is LOW.
SEQ	OCZ	Sequential address. This signal is the inverse of **nMREQ**, and is provided for compatibility with existing ARM memory systems.
TESTMODE	I	This signal must be tied LOW.
TESTOUT[4:0]	O	This bus should be left unconnected. These outputs will be driven LOW except when device test features are enabled. They will not be tri-stated, except via the JTAG test port.
TCK	I	Test interface reference Clock. This times all the transfers on the JTAG test interface.

Table 2-1: Signal descriptions (Continued)

Name	Type	Description:
TDI	I	Test interface data input. Note this signal does *not* have an internal pullup resistor.
TDO	OCZ	Test interface data output. Note this signal does *not* have an internal pullup resistor.
TMS	I	Test interface mode select. Note this signal does *not* have an internal pullup resistor.
VCC		Pad voltage reference. 1 pin is allocated to VCC. This should be tied to the system power supply, ie. 5V in a TLL system or 3.3V in a 3.3V system. See ***Appendix A, Use of the ARM810 in a 5V TTL System***.
VDD		Positive supply. 15 pins are allocated to **VDD** in the 144 TQFP package.
VSS		Ground supply. 15 pins are allocated to **VSS** in the 144 TQFP package.

Table 2-1: Signal descriptions (Continued)

Notes

1 When output pads are placed in the high impedance state for long periods, care must be taken to ensure that they do not float to an undefined logic level, as this can dissipate power, especially in the pads.

2 The input pads on this device are compatible with both CMOS and TTL signals. The thresholds can be found in ***Chapter 14, ARM810 DC Parameters***.

Key to signal types:

I	Input
OCZ	Output, CMOS levels, tristateable
IOCZ	Input/output tristateable, CMOS levels
ICK	Clock input

3 Programmer's Model

This chapter describes the programmer's model.

Programmer's Model

3.1 Introduction

The ARM810 supports a variety of operating conditions. Some are controlled by register bits and are known as the register configurations and others are controlled by software which are known as operating modes.

The ARM810 programmers model can be split into two distinct segments. The ARM8 CPU core can be configured to a number of hardware configurations and a number of operating modes. In adidtion the cache, MMU and branch prediction blocks exterior to the ARM8 core can also be configured to operate in different ways by software access to the Coprocessor 15 (CP15) configuration registers. This section concentrates on the programmer's model for the ARM8 CPU core. The details of accessing CP15 for the configuration of the ARM810 blocks external to the ARM8 core can be found in ***Chapter 5, Configuration***.

3.2 ARM810 Configuration

The ARM810 Microprocessor incorporates advanced cache, memory management and branch prediction features.

The ARM810 Cache features are described in detail in ***Chapter 7, Instruction and Data Cache (IDC)***. The Memory Management Unit features are described in ***Chapter 8, Memory Management Unit***. The Write Buffer configuration is decribed in detail in ***Chapter 9, Write Buffer***. The Prefetch Unit features including branch prediction configuration is documented in ***Chapter 6, The Prefetch Unit***.

3.3 ARM8 CPU Core Configuration

The ARM8 CPU core provides 1 register configuration setting which may be changed while the processor is running and which is discussed below. Please refer to ***Chapter 5, Configuration*** for details of how to configure the ARM8.

3.3.1 Big and little-endian memory formats (the BIGEND signal)

Memory is viewed as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second and so on. ARM810 can treat words in memory as being stored either in Big Endian or Little Endian format, depending on the state of the **BIGEND** input.

The Load/Store instructions are the only ones affected by the endianness.

Little-endian format

In Little-endian format (**BIGEND** LOW) the lowest numbered byte in a word is considered the least significant byte of the word, and the highest numbered byte the most significant. Byte 0 of the memory system should therefore be connected to data lines 7 through 0.

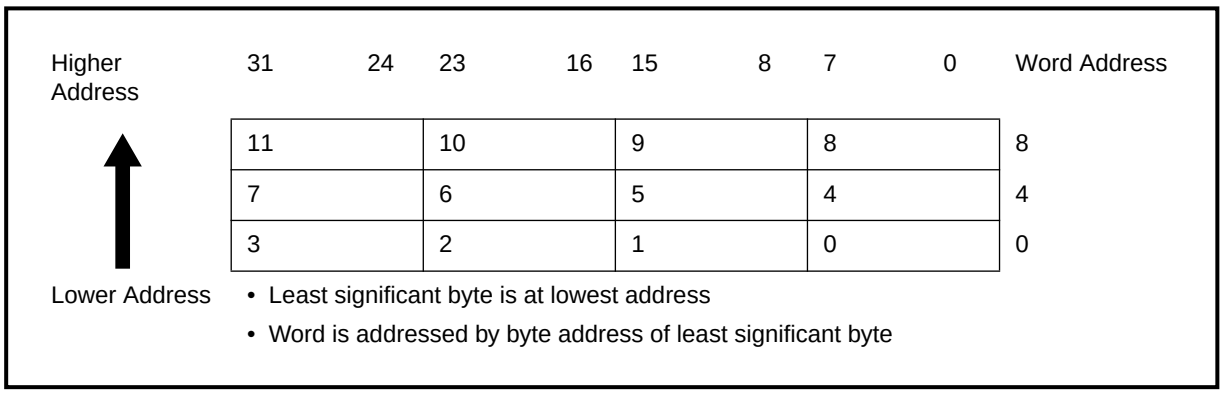


Figure 3-1: Little-endian addresses of bytes within words

Big-endian format

In Big-endian format (**BIGEND** HIGH) the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24.

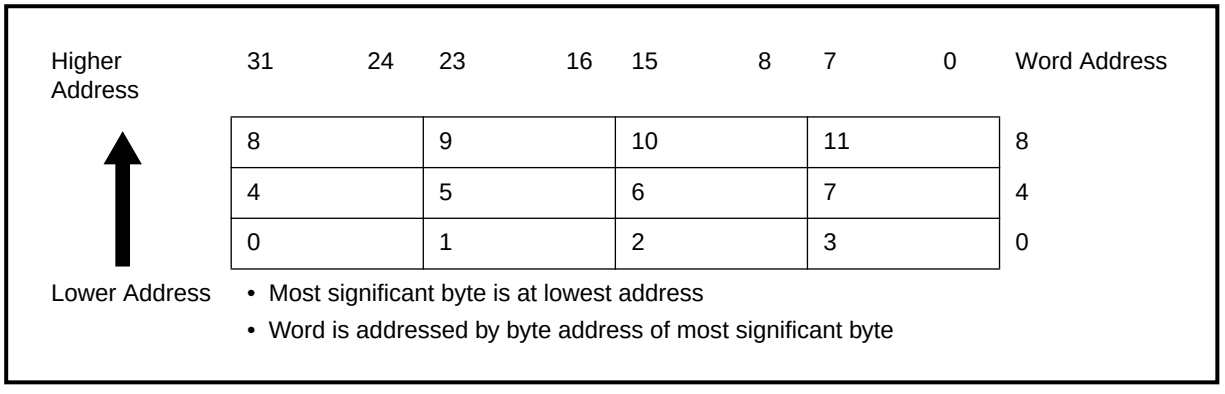


Figure 3-2: Big-endian addresses of bytes within words

3.4 Operating Modes

ARM810 supports *byte* (8-bit), *half-word* (16-bit), and *word* (32-bit) data types. Instructions are exactly one word long, and must be aligned to four-byte boundaries. Data operations, such as ADD, are only performed on word quantities. Load and Store operations are able to transfer bytes, half-words or words.

ARM810 supports seven modes of operation:

Mode	Description
User mode (usr)	normal program execution state
FIQ mode (fiq)	used for fast or higher priority interrupt handling
IRQ mode (irq)	used for general-purpose interrupt handling
Supervisor mode (svc)	a protected mode for the operating system
System mode (sys)	a privileged user mode for the operating system
Abort mode (abt)	entered after a data or instruction prefetch abort
Undefined mode (und)	entered when an undefined instruction is executed

Mode changes may be made under software control, or may be brought about by external interrupts or exception processing. Most application programs will execute in User mode. The other modes, known as *privileged modes*, are entered in order to service interrupts or exceptions, or to access protected resources.

3.5 Registers

ARM8 has a total of 37 registers - 31 general-purpose 32-bit registers and six status registers - but these cannot all be seen at once. The processor mode dictates which registers are available to the programmer. At any one time, 16 general registers and one or two status registers are visible. In privileged (non-User) modes, mode-specific *banked* registers are switched in. ***Figure 3-3: Register organisation*** on page 3-6 shows which registers are available in each processor mode: each of the banked registers is marked with a shaded triangle.

In all modes there are 16 directly accessible registers: R0 to R15. All of these except R15 are general-purpose registers which may be used to hold either data or address values. Register R15 holds the Program Counter (PC). When read, bits [1:0] of R15 are zero and bits [31:2] contain the PC. A seventeenth register, the CPSR (Current Program Status Register), is also accessible. This contains condition code flags and the current mode bits, and may be thought of as an extension to the PC.

R14 is used as the subroutine link register (LR). This receives a copy of R15 when a Branch and Link (BL) instruction is executed. At all other times it may be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within exception routines.

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ handlers will not need to save any registers. User, IRQ, Supervisor, Abort and Undefined each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer (SP) and link register (LR).

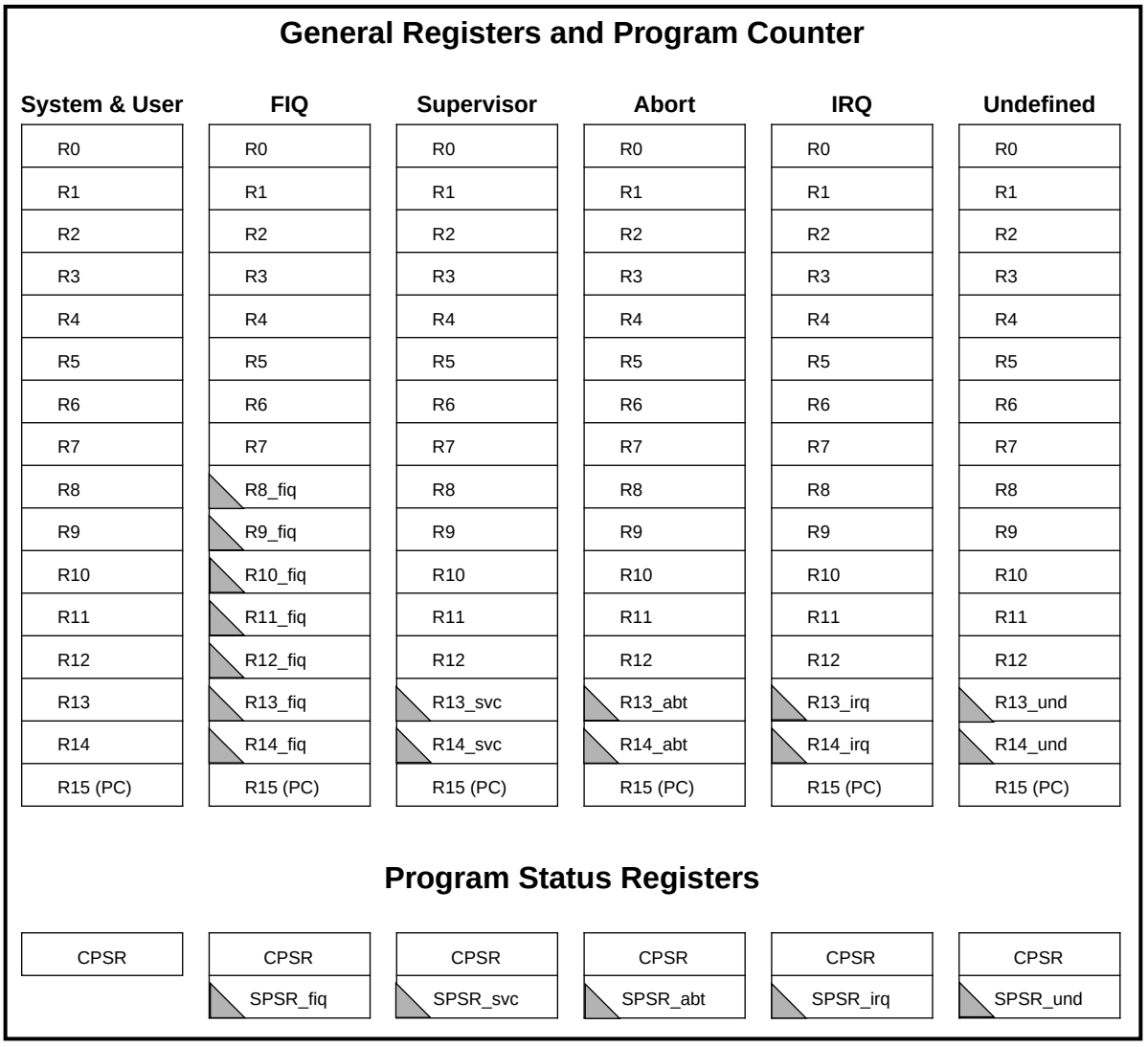


Figure 3-3: Register organisation

Supervisor, IRQ, Abort and Undefined mode programs which require more than these two banked registers are expected to save some or all of the caller's registers (R0 to R12) on their respective stacks. They are then free to use these registers, which they will restore before returning to the caller.

In addition there are five SPSRs (Saved Program Status Registers) which are loaded with the CPSR whenever an exception occurs. There is one SPSR for each privileged (non-User) mode, except System mode.

Note No SPSR exists for User or System modes because no exceptions enter these modes. Instructions that attempt to access this SPSR should not be executed in User or System mode.

3.5.1 Program Status Register format

Figure 3-4: Program Status Register (PSR) format shows the format of the Program Status Registers.

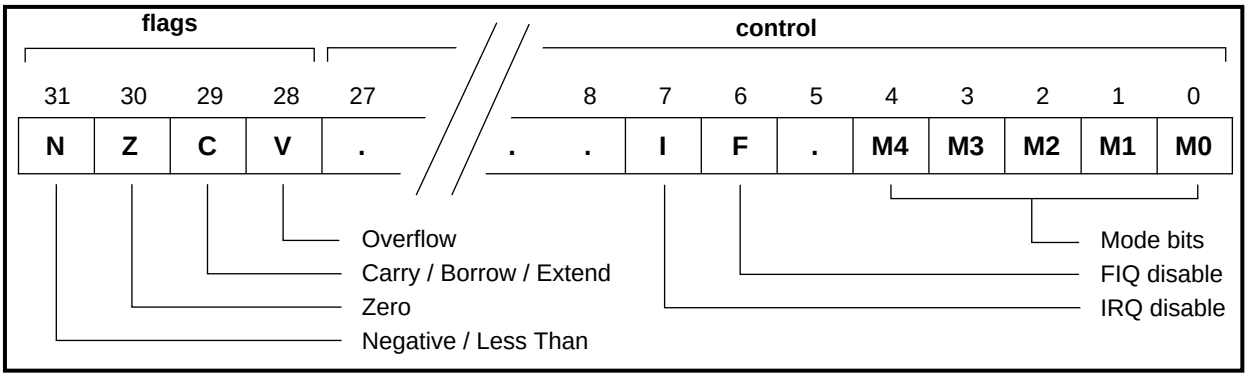


Figure 3-4: Program Status Register (PSR) format

Condition code flags

The N, Z, C and V bits are the *condition code flags*. These may be changed as a result of arithmetic and logical operations in the processor, and may be tested by any instruction to determine whether the instruction is to be executed.

Interrupt disable bits

The I and F bits are the *interrupt disable bits.* When set, these disable the IRQ and FIQ interrupts respectively.

Mode bits

The M4, M3, M2, M1 and M0 bits (M[4:0]) are the *mode bits*. These determine the mode in which the processor operates. The interpretation of the mode bits is shown in ***Table 3-1: The mode bits*** on page 3-8. Not all mode bit combinations define a valid processor mode: you should only use those which are explicitly described.

Control bits

The bottom 8 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. These will change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

Reserved bits

The remaining bits in the PSRs are *reserved*. When changing a PSR's flag or control bits, you must ensure that these unused bits are not changed by using a read-modify-write scheme. Also, your program should not rely on them containing specific values, since in future processors they may read as one or zero.

M[4:0]	Mode	Accessible Registers	
10000	User	PC, R14..R0	CPSR
10001	FIQ	PC, R14_fiq..R8_fiq, R7..R0	CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq..R13_irq, R12..R0	CPSR, SPSR_irq
10011	Supervisor	PC, R14_svc..R13_svc, R12..R0	CPSR, SPSR_svc
10111	Abort	PC, R14_abt..R13_abt, R12..R0	CPSR, SPSR_abt
11011	Undefined	PC, R14_und..R13_und, R12..R0	CPSR, SPSR_und
11111	System	PC, R14..R0	CPSR

Table 3-1: The mode bits

3.6 Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that the processor can be diverted to handle an interrupt from a peripheral, for example. The processor state immediately prior to handling the exception must be preserved, to ensure that the original program can be resumed when the exception routine has completed. It is possible for more than one exception to arise at the same time.

When handling an exception, ARM810 makes use of the banked registers to save state. The old PC and CPSR contents are copied into the appropriate R14 and SPSR, and the PC and the CPSR mode bits are forced to a value which depends on the exception. Where necessary, the interrupt disable flags are set to prevent otherwise unmanageable nestings of exceptions; this is detailed in the following sections.

In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before the interrupt is re-enabled. When transferring the SPSR register to and from a stack, it is important to transfer the whole 32-bit value and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled: see ***3.6.7 Exception priorities*** on page 3-11 for more information.

3.6.1 FIQ

The FIQ (fast interrupt request) exception is externally generated by taking the **nFIQ** input LOW. This input can accept asynchronous transitions because the ARM will always perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle.

FIQ is designed to support a fast or high priority interrupt, and has sufficient private registers to remove the need for register saving in such applications (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the CPSR's F flag (but note that this is not possible from User mode). If the F flag is clear, ARM810 checks for a LOW level on the FIQ logic output at the end of each instruction (including cancelled ones), and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting a FIQ, ARM810:

- saves the address of the next instruction to be executed plus 4 in R14_fiq
- saves the CPSR in SPSR_fiq
- forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR
- forces the PC to fetch the next instruction from the FIQ vector

To return normally from FIQ, use `SUBS PC,R14_fiq,#4`. This restores both the PC (from R14) and the CPSR (from SPSR_fiq), and resumes execution of the interrupted code.

3.6.2 IRQ

The IRQ (interrupt request) exception is externally generated by taking the **nIRQ** input LOW. This input can accept asynchronous transitions because the ARM will always perform the synchronisation. This synchronisation delays the effect of the input transition on the processor execution flow for one cycle.

IRQ has a lower priority that FIQ and is automatically masked out when a FIQ sequence is entered. The IRQ exception may be disabled by setting the CPSR's I flag (but note that this is not possible from User mode). If the I flag is clear, ARM810 checks for a LOW level on the IRQ logic output at the end of each instruction (including cancelled ones) and at the end of any coprocessor busy-wait cycle (allowing the busy-wait state to be interrupted).

On detecting an IRQ, ARM8:

- saves the address of the next instruction to be executed plus 4 in R14_irq
- saves the CPSR in SPSR_irq
- forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the IRQ vector

To return normally from IRQ, use `SUBS PC,R14_irq,#4`. This restores both the PC (from R14) and the CPSR (from SPSR_irq), and resumes execution of the interrupted code.

3.6.3 Aborts

Not all requests to the Memory System for Data or Instructions will result in a successful completion of the transaction. Such transactions result in an Abort. The rest of this section describes sources and types of aborts, and what happens once they occur.

Abort Sources

Aborts can be generated by Store instructions (STR, STM, SWP (for the write part)), by Data Read instructions (LDR, LDM, SWP (for the read part)) and by Instruction Prefetching. Please refer to ***8.12 MMU Faults and CPU Aborts*** on page 8-16 and ***8.16 External Aborts*** on page 8-23 for further details of Aborts.

Abort types

Aborts are classified as either Prefetch or Data Abort types depending upon the transaction taking place at the time. Each type has its own exception vector to allow branching to the relevant service routine to deal with them. These exception vectors are the Prefetch Abort Vector and the Data Abort Vector and their locations are summarised in ***3.6.6 Exception vector summary*** on page 3-11.

Prefetch Aborts

If the transaction taking place when the abort happened was an Instruction fetch, then a Prefetch Abort is indicated. The instruction is marked as invalid, but the abort exception vector is not taken immediately. Only if the instruction is about to get executed will the Prefetch Abort exception vector be taken.

For Prefetch Aborts, ARM8:

1. Saves the address of the aborted instruction plus 4 into R14_abt
2. Saves the CPSR into SPSR_abt

3 Forces M[4:0] to 10111 (Abort Mode) and sets the I bit in the CPSR
4 Forces the PC to fetch the next instruction from the Prefetch Abort vector.

Returning from a Prefetch Abort: After fixing the reason for the Prefetch Abort, use:

```
SUBS PC,R14_abt,#4
```

This restores both the PC (from R14) and the CPSR (from SPSR_abt), and retries the instruction.

Data aborts

If the transaction taking place when the abort happened was a Data Access (Read or Write), then a Data Abort is indicated, and the action depends upon the instruction type that caused it. In ALL cases, any base register is restored to the value it had before the instruction started whether or not writeback is specified. In addition:

- The LDR instruction does not overwrite the destination register.
- The SWP Instruction is aborted as though it had not executed, although externally the read access may have taken place.
- The LDM Instruction ensures that the PC is not overwritten and will restore the base register such that the instruction can be restarted. All registers up to the aborting one may have been overwritten, but no further ones will be.
- The STM Instruction will ensure that the base register is restored, and any stores up to the aborting one will have already been made - the details depending upon the Memory System itself.

For Data Aborts, ARM8:

1 Saves the address of the instruction which caused the abort plus 8 into R14_abt
2 Saves the CPSR into SPSR_abt
3 Forces M[4:0] to 10111 (Abort Mode) and sets the I bit in the CPSR
4 Forces the PC to fetch the next instruction from the Data Abort vector.

Returning from a Data Abort: After fixing the reason for the Data Abort, use:

```
SUBS PC,R14_abt,#8
```

This restores both the PC (from R14) and the CPSR (from SPSR_abt), and retries the instruction. Note that in the case of LDM, some registers may be re-loaded.

3.6.4 Software interrupt

The software interrupt instruction (SWI) is used for entering Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, ARM810:

- saves the address of the SWI instruction plus 4 in R14_svc
- saves the CPSR in SPSR_svc
- forces M[4:0]=10011 (Supervisor mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the SWI vector

To return from a SWI, use `MOVS PC,R14_svc`. This restores the PC (from R14) and CPSR (from SPSR_svc), and returns to the instruction following the SWI.

3.6.5 Undefined instruction trap

When the ARM810 decodes an instruction bit-pattern that it cannot process, it takes the undefined instruction trap.

Note Not all non-instruction bit patterns are detected, but such bit patterns will not halt or corrupt the processor and its state.

The trap may be used for software emulation of a coprocessor in a system which does not have the coprocessor hardware (and therefore cannot process), or for general-purpose instruction set extension by software emulation.

When ARM810 takes the undefined instruction trap, it:

- saves the address of the Undefined instruction plus 4 in R14_und
- saves the CPSR in SPSR_und
- forces M[4:0]=11011 (Undefined mode) and sets the I bit in the CPSR
- forces the PC to fetch the next instruction from the Undefined vector

To return from this trap after servicing or emulating the trapped instruction, use `MOVS PC,R14_und`. This restores the PC (from R14) and the CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

3.6.6 Exception vector summary

Address	Exception	Mode on Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	-- reserved --	--
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

Table 3-2: Exception vectors

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine.

To enhance FIQ response time, the FIQ routine might reside at 0x1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

3.6.7 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled.

1 Reset (highest priority)
2 Data Abort
3 FIQ
4 IRQ
5 Prefetch Abort
6 Undefined Instruction, Software interrupt (lowest priority)

Not all of the exceptions can occur at once: Undefined Instruction and Software Interrupt are mutually exclusive, since they each correspond to particular (non-overlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (ie the CPSR's F flag is clear), ARM810 enters the data abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume

execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry should be added to worst-case FIQ latency calculations.

3.7 Reset

nRESET can be asserted asynchronously. When the **nRESET** signal goes LOW, ARM810 abandons the executing instruction. When **nRESET** goes HIGH again, ARM810:

- overwrites R14_svc and SPSR_svc (by copying the current values of the PC and CPSR into them) with undefined values.
- forces M[4:0]=10011 (Supervisor mode) and sets the I and F bits in the CPSR
- forces the PC to fetch the next instruction from the Reset vector