1. Introduction

The **ARM60** is part of the Advanced RISC Machine (ARM) family of general purpose 32-bit single-chip microprocessors. The ARM architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are greatly simplified compared with microprogrammed Complex Instruction Set Computers. This simplification results in a high instruction throughput and a good real-time interrupt response from a small and cost-effective chip.

The instruction set of **ARM60** consists of ten basic instruction types. Two of these make use of the on-chip arithmetic logic unit (ALU), barrel shifter and multiplier to perform high-speed operations on the data in a bank of 27 registers, each 32 bits wide. Three instruction types control the transfer of data between main memory and the register bank, one optimised for flexibility of addressing, another for rapid context switching, and the third for indivisible semaphore operations. Two instructions control the flow and privilege level of execution, and the remaining three types are dedicated to the control of external coprocessors which allow the functionality of the instruction set to be extended off-chip in an open and uniform way.

The ARM instruction set has proved to be a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors which depend on sophisticated compiler technology to manage complicated instruction interdependencies.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The memory 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, and these control signals facilitate the exploitation of the fast local access modes offered by industry standard dynamic RAMs.

ARM60 has a 32 bit address bus, unlike earlier ARM processors which had a 26 bit address bus. All ARM processors share the same instruction set, and **ARM60** can be configured to use a 26 bit address bus for backwards compatibility with earlier processors.

ARM60 is a fully static CMOS implementation of the ARM which allows the clock to be stopped in any part of the cycle with extremely low residual power consumption and no loss of state.

FEATURES

- 32-bit data bus
- 32-bit address bus giving a uniform 4 gigabyte address space
- Big and Little Endian operating modes
- Support for virtual memory systems
- Fast interrupt response for real-time applications
- Simple but powerful instruction set with good high-level language compiler support
- Coprocessor interface for instruction set extension
- Fully static operation with very low standby power consumption
- 100 pin Plastic Quad Flat Pack (PQFP) package
- JTAG Boundary Scan for device and system testing
- Low power consumption with a single 5V supply

2. Block Diagram

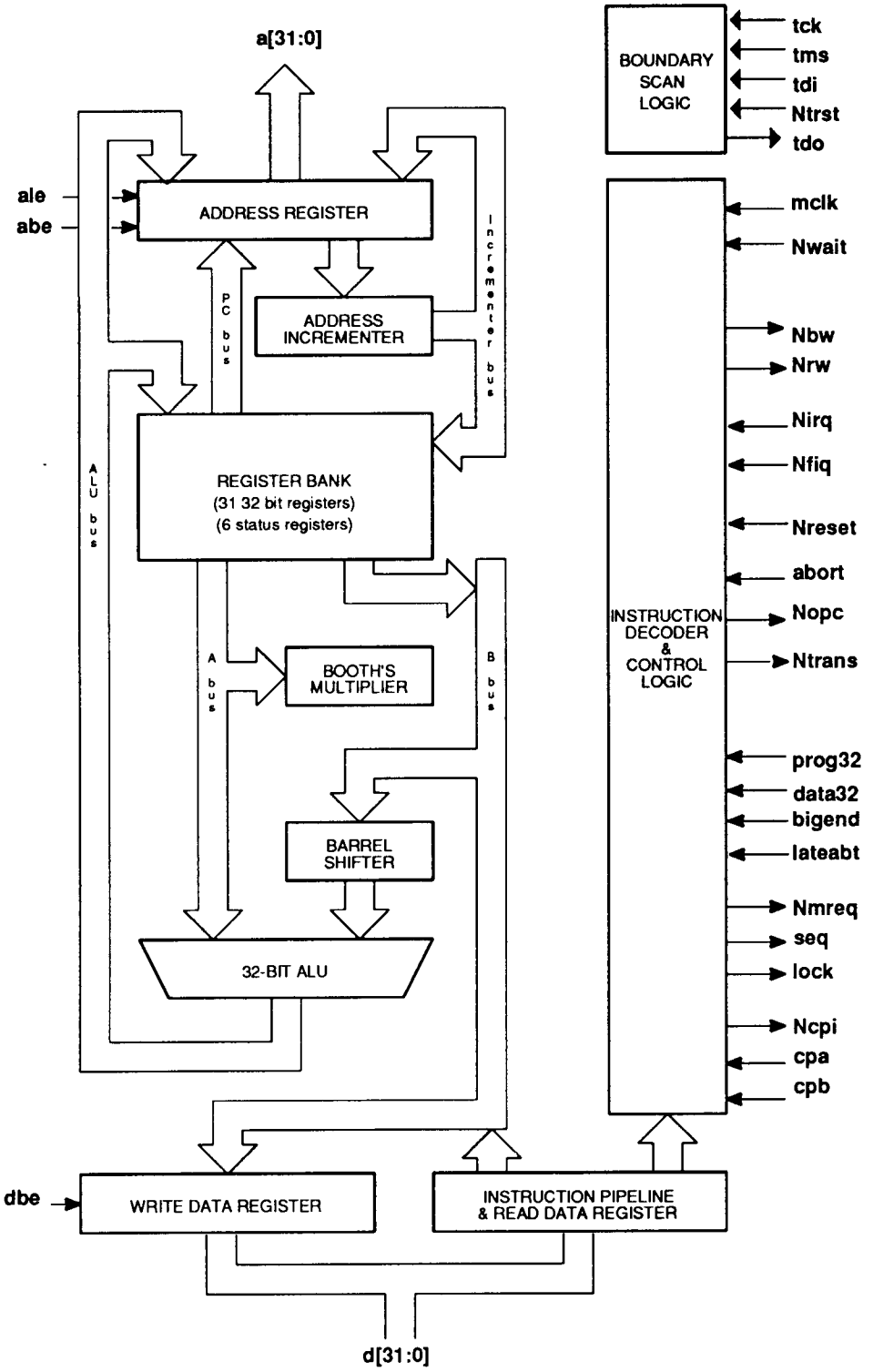


Figure 1: **ARM60** Block Diagram

3. Functional Diagram

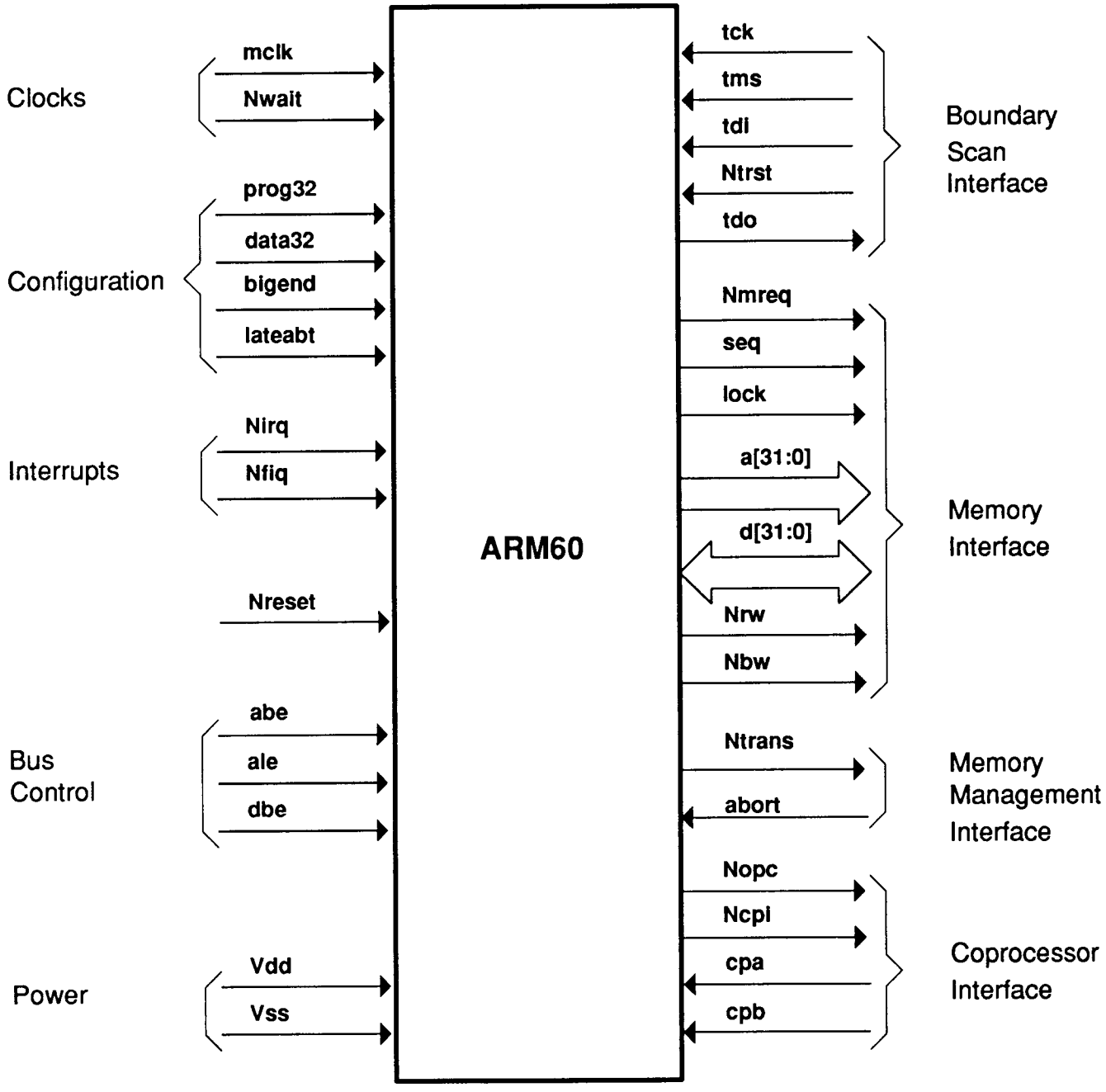


Figure 2: **ARM60** Functional Diagram

4. Description of Signals

Name	Pin	Type	Description
mclk	14	IT	Memory clock input. This clock times all **ARM60** memory accesses and internal operations. The clock has two distinct phases - *phase 1* in which **mclk** is LOW and *phase 2* in which **mclk** is HIGH. The clock may be stretched indefinitely in either phase to allow access to slow peripherals or memory. Alternatively, the **Nwait** input may be used with a free-running **mclk** to achieve the same effect.
Nwait	15	IT	NOT wait. When accessing slow peripherals, **ARM60** can be made to wait for an integer number of **mclk** cycles by driving **Nwait** LOW. Internally, **Nwait** is ANDed with the **mclk** clock, and must only change when **mclk** is LOW. If **Nwait** is not used in a system, it may be tied HIGH.
Nreset	26	IT	NOT reset. This is a level sensitive input signal which is used to start the processor from a known address. A LOW level will cause the instruction being executed to terminate abnormally. When **Nreset** becomes HIGH for at least one clock cycle, the processor will re-start from address 0. **Nreset** must remain LOW (and **Nwait** must remain HIGH) for at least two clock cycles, and during the LOW period the processor will perform dummy instruction fetches with the address incrementing from the point where reset was activated. The address value will overflow to zero if **Nreset** is held beyond the maximum address limit.
a[31:0]	30-50,53-63	OS8	Addresses. This is the processor address bus. If **ale** (address latch enable) is HIGH, the addresses become valid during phase 2 of the cycle before the one to which they refer and remain so during phase 1 of the referenced cycle. Their stable period may be controlled by **ale** as described below.
ale	27	IT	Address latch enable. This input to the processor is used to control transparent latches on the address outputs. Normally the addresses change during phase 2 to the value required during the next cycle, but for direct interfacing to ROMs they are required to be stable to the end of phase 2. Taking **ale** LOW until the end of phase 2 will ensure that this happens. If the system does not require address lines to be held in this way, **ale** may be held permanently HIGH. The **ale** latch is static, so **ale** may be held LOW indefinitely.
abe	66	IT	Address bus enable. This is an input signal which, when LOW, puts the address bus drivers into a high impedance state. **abe** may be tied HIGH when there is no system requirement to turn off the address drivers.
d[31:0]	5-1,100-82,79-72	IT/OS8	Data Bus. These are bidirectional signal paths which are used for data transfers between the processor and external memory. During read cycles (when **Nrw** is LOW), the input data must be valid before the end of phase 2 of the transfer cycle. During write cycles (when **Nrw** is HIGH), the output data will become valid during phase 1 and remain valid throughout phase 2 of the transfer cycle.

dbe	12	IT	Data bus enable. When this input is LOW, the data bus drivers (**d[31:0]**) are put into a high impedance state. The drivers will always be high impedance except during write operations, and **dbe** may be tied HIGH in systems which do not require the data bus for DMA or similar activities.
Nrw	19	OS8	NOT read/write. When HIGH this signal indicates a processor write cycle; when LOW, a read cycle. It becomes valid during phase 2 of the cycle before that to which it refers, and remains valid to the end of phase 1 of the referenced cycle.
Nrw	13	OS8	NOT read/write. This is an output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. The signal is HIGH for word transfers and LOW for byte transfers and is valid for both read and write cycles. The signal will become valid during phase 2 of the cycle before the one in which the transfer will take place. It will remain stable throughout phase 1 of the transfer cycle.
lock	9	OS8	Locked operation. When **lock** is HIGH, the processor is performing a "locked" memory access, and the memory controller should wait until **lock** goes LOW before allowing another device to access the memory. **lock** changes while **mclk** is HIGH, and remains HIGH for the duration of the locked memory accesses. It is active only during the data swap (SWP) instruction.
Nmreq	21	O4	NOT memory request. This signal, when LOW, indicates that the processor requires memory access during the following cycle. The signal becomes valid during phase 1, remaining valid through phase 2 of the cycle preceding that to which it refers.
seq	22	O4	Sequential address. This output signal will become HIGH when the address of the next memory cycle will be related to that of the last memory access. The new address will either be the same as or 4 greater than the old one. The signal becomes valid during phase 1 and remains so through phase 2 of the cycle before the cycle whose address it anticipates. It may be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) and/or to by-pass the address translation system.
Nopc	20	O4	NOT op-code fetch. When LOW this signal indicates that the processor is fetching an instruction from memory; when HIGH data (if anything) is being transferred. The signal becomes valid during phase 2 of the previous cycle, remaining valid through phase 1 of the referenced cycle.
Ntrans	29	OS8	NOT memory translate. When this signal is LOW it indicates that the processor is in user mode. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator of non-user mode activity.
abort	23	IT	Memory abort. This is an input which allows the memory system to tell the processor that a requested access is not allowed. The signal must be valid before the end of phase 1 of the cycle during which the memory transfer is attempted.

Nirq	24	IT	NOT interrupt request. This is an asynchronous interrupt request to the processor which causes it to be interrupted if taken LOW when the appropriate enable in the processor is active. The signal is level sensitive and must be held LOW until a suitable response is received from the processor.
Nfiq	25	IT	NOT fast interrupt request. As **Nirq**, but with higher priority. May be taken LOW asynchronously to interrupt the processor when the appropriate enable is active.
Ncpi	11	O4	NOT Coprocessor instruction. When **ARM60** executes a coprocessor instruction, it will take this output LOW and wait for a response from the coprocessor. The action taken will depend on this response, which the coprocessor signals on the **cpa** and **cpb** inputs.
cpb	28	IT	Coprocessor busy. A coprocessor which is capable of performing the operation which **ARM60** is requesting (by asserting **Ncpi**), but cannot commit to starting it immediately, should indicate this by driving **cpb** HIGH. When the coprocessor is ready to start it should take **cpb** LOW. **ARM60** samples **cpb** at the end of phase 1 of each cycle in which **Ncpi** is LOW.
cpa	6	IT	Coprocessor absent. A coprocessor which is capable of performing the operation which **ARM60** is requesting (by asserting **Ncpi**) should take **cpa** LOW immediately. If **cpa** is HIGH at the end of phase 1 of the cycle in which **Ncpi** went LOW, **ARM60** will abort the coprocessor handshake and take the undefined instruction trap. If **cpa** is LOW and remains LOW, **ARM60** will busy-wait until **cpb** is LOW and then complete the coprocessor instruction.
prog32	17	IT	32 bit Program configuration. When this signal is HIGH the processor can fetch instructions from a 32 bit address space using address lines **a[31:0]**. When it is LOW the processor fetches instructions from a 26 bit address space using **a[25:0]**. In this latter configuration the address lines **a[31:26]** are not used for instruction fetches. Before changing **prog32**, ensure that the processor is in a 26 bit mode, and is not about to write to an address in the range 0 to &1F (inclusive) in the next cycle.
data32	18	IT	32 bit Data configuration. When this signal is HIGH the processor can access data in a 32 bit address space using address lines **a[31:0]**. When it is LOW the processor can access data from a 26 bit address space using **a[25:0]**. In this latter configuration the address lines **a[31:26]** are not used. Before changing **data32**, ensure that the processor is not about to access an address greater that &3FFFFFF in the next cycle.
bigend	10	IT	Big Endian configuration. When this signal is HIGH the processor treats bytes in memory as being in Big Endian format. When it is LOW memory is treated as Little Endian. ARM processors which do not have selectable Endianism (ARM2, ARM2aS, ARM3, ARM61) are Little Endian.
lateabt	16	I	Late Abort. This signal controls the action of the processor on an Abort exception. When it is HIGH (Late Abort) the modified base register of an aborted LDR or STR instruction is written back. When it is LOW (Early Abort) the modified base register is not written back. **lateabt** must not be changed during the execution of a data access instruction where **abort** is active. It is recommended that the Late Abort scheme be used where possible as this scheme will be used in

			future ARM processors. ARM2, ARM2aS, ARM3 and ARM61 support the Early Abort mechanism.
Ntrst	69	IP	NOT Test Reset. Active-low reset signal for the boundary scan logic. This input has an on-chip pullup resistor to **VDD**. The action of this and the following four boundary scan signals are described in more detail later in this document.
tms	68	IP	Test Mode Select. This input to the boundary scan logic has an on-chip pullup resistor to **VDD**.
tdi	68	IP	Test Data Input. This input to the boundary scan logic has an on-chip pullup resistor to **VDD**.
tck	67	IP	Test Clock. This input to the boundary scan logic has an on-chip pullup resistor to **VDD**.
tdo	71	OS8	Test Data Output. Output from the boundary scan logic.
VSS	7,52,64,80	P	Ground. These pins are the ground reference for all signals.
VDD	8,51,65,81	P	Power supply. These pins provide power to the device. A nominal voltage of +5V relative to ground is required.

Key to Signal Types

IT	Input with TTL thresholds
IP	Input with TTL thresholds and pull-up resistor
O4	CMOS Output (4mA drive)
OS8	CMOS slew-limited output (8mA drive)
P	Power

5. Configuration and Mode Selection

ARM60 supports a variety of operating configurations. Some are controlled by signal inputs and are known as the *hardware configurations*. Others may be controlled by software and these are known as *operating modes*.

5.1 Hardware Configuration

The **ARM60** provides 4 hardware configuration inputs which may be changed while the processor is running. The inputs may only change during phase 2 of the clock cycle and there are restrictions which were described in the previous chapter.

Two of the inputs (**data32** and **prog32**) allow one of three processor configurations to be selected as follows.

(1) **26 bit program and data space** - (**prog32** LOW, **data32** LOW). This configuration forces **ARM60** to operate like the earlier ARM processors with 26 bit address space. The programmer's model for these processors applies, but the new instructions to access the CPSR and SPSR registers operate as detailed later in this document. In this configuration it is impossible to select a 32 bit operating mode, and all exceptions (including address exceptions) enter the exception handler in the appropriate 26 bit mode.

(2) **26 bit program space and 32 bit data space** - (**prog32** LOW, **data32** HIGH). This is the same as the 26 bit program and data space configuration, but with address exceptions disabled to allow data transfer operations to access the full 32 bit address space.

(3) **32 bit program and data space** - (**prog32** HIGH, **data32** HIGH). This configuration extends the address space to 32 bits, introduces major changes in the programmer's model as described below and provides support for running existing 26 bit programs in the 32 bit environment.

The fourth processor configuration which is possible (26 bit data space and 32 bit program space) should not be selected.

The **bigend** signal controls whether the **ARM60** treats words in memory as being stored in Big Endian or Little Endian format. 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.

In the Little Endian scheme the lowest numbered byte in a word is considered to be the least significant byte of the word and the highest numbered byte is the most significant. Byte 0 of the memory system should be connected to data lines 7 through 0 (**d[7:0]**) in this scheme.

In the Big Endian scheme the most significant byte of a word is stored at the lowest numbered byte and the least significant byte is stored at the highest numbered byte. Byte 0 of the memory system should therefore be connected to data lines 31 through 24 (**d[31:24]**)

The **lateabt** signal controls the processor's behaviour when a data abort exception occurs. It only affects the behaviour of LDR and STR instructions and is discussed more fully in the Programmer's Model and Instruction Set chapters.

5.2 Operating Mode Selection

When configured for 26 bit program space, **ARM60** is limited to operating in one of four modes known as the 26 bit modes. These modes correspond to the modes of the earlier ARM processors and are known as **User26, FIQ26, IRQ26** and **Supervisor26**.

When using a 32 bit program space there are a total of 10 modes available. These are the four 26 bit modes described above plus 6 more known as the 32 bit modes. These are **User32, FIQ32, IRQ32, Supervisor32, Abort32,** and **Undefined32**. These are the normal operating modes in this configuration and

the 26 bit modes are only provided for backwards compatibility to allow execution of programs originally written for earlier ARM processors.

The differences between **ARM60** and the earlier ARM processors are documented in an ARM Application Note - "Differences between ARM6 and earlier ARM Processors"

Important Note

(1) The remainder of this document describes **ARM60** when configured for 32 bit program and data space and operating in one of the 32 bit modes. It is recommended that all new designs using **ARM60** should configure the processor in this way by setting **prog32** and **data32** HIGH and that all new code should be written to use only the 32 bit operating modes. It is also recommended that the **lateabt** input be set HIGH so that the Late Abort exception mechanism is used.

(2) Because the original ARM instruction set has been modified to accommodate 32 bit operation there are certain additional restrictions which programmers must be aware of. These are indicated in the text by the words **shall** and **shall not**. Reference should also be made to the ARM Application Notes "Rules for ARM Code Writers" and "Notes for ARM Code Writers".

6. Programmer's Model

6.1 Introduction

ARM60 has a 32 bit data bus and a 32 bit address bus. The data types the processor supports are Bytes (8 bits) and Words (32 bits), where words must be aligned to four byte boundaries. Instructions are exactly one word, and data operations (e.g. ADD) are only performed on word quantities. Load and store operations can transfer either bytes or words.

ARM60 supports six modes of operation:-

(1) User mode: the normal program execution state

(2) FIQ mode (fiq): designed to support a data transfer or channel process

(3) IRQ mode (irq): used for general purpose interrupt handling

(4) Supervisor mode (svc): a protected mode for the operating system

(5) Abort mode (abt): entered after a data or instruction prefetch abort

(6) 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*, will be entered to service interrupts or exceptions or to access protected resources.

6.2 Registers

The processor has a total of 37 registers made up of 31 general purpose 32 bit registers and 6 status registers. At any one time 16 general purpose registers (R0 to R15) and one or two status registers are visible to the programmer. The visible registers depend on the processor mode and the other registers (the *banked registers*) are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing. The register bank organisation is shown in Figure 3. The banked registers are shaded in the diagram.

In all modes 16 registers, R0 to R15, are directly accessible. All registers except R15 are general purpose and may be used to hold data or address values. Register R15 holds the Program Counter (PC). When R15 is read, bits [1:0] are zero and bits [31:2] contain the PC. A seventeenth register (the CPSR - Current Program Status Register) is also accessible. It 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 and receives a copy of R15 when a Branch and Link instruction is executed. It may be treated as a general purpose register at all other times. R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are used similarly to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). Many FIQ programs will not need to save any registers. User mode, IRQ mode, Supervisor mode, Abort mode and Undefined mode each have two banked registers mapped to R13 and R14. The two banked registers allow the these modes to each have a private stack pointer and link register. 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 also five SPSRs (Saved Program Status Registers) which are loaded with the CPSR when an exception occurs. There is one SPSR for each privileged mode.

The format of the Program Status Registers is shown in Figure 4. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction is to be executed.

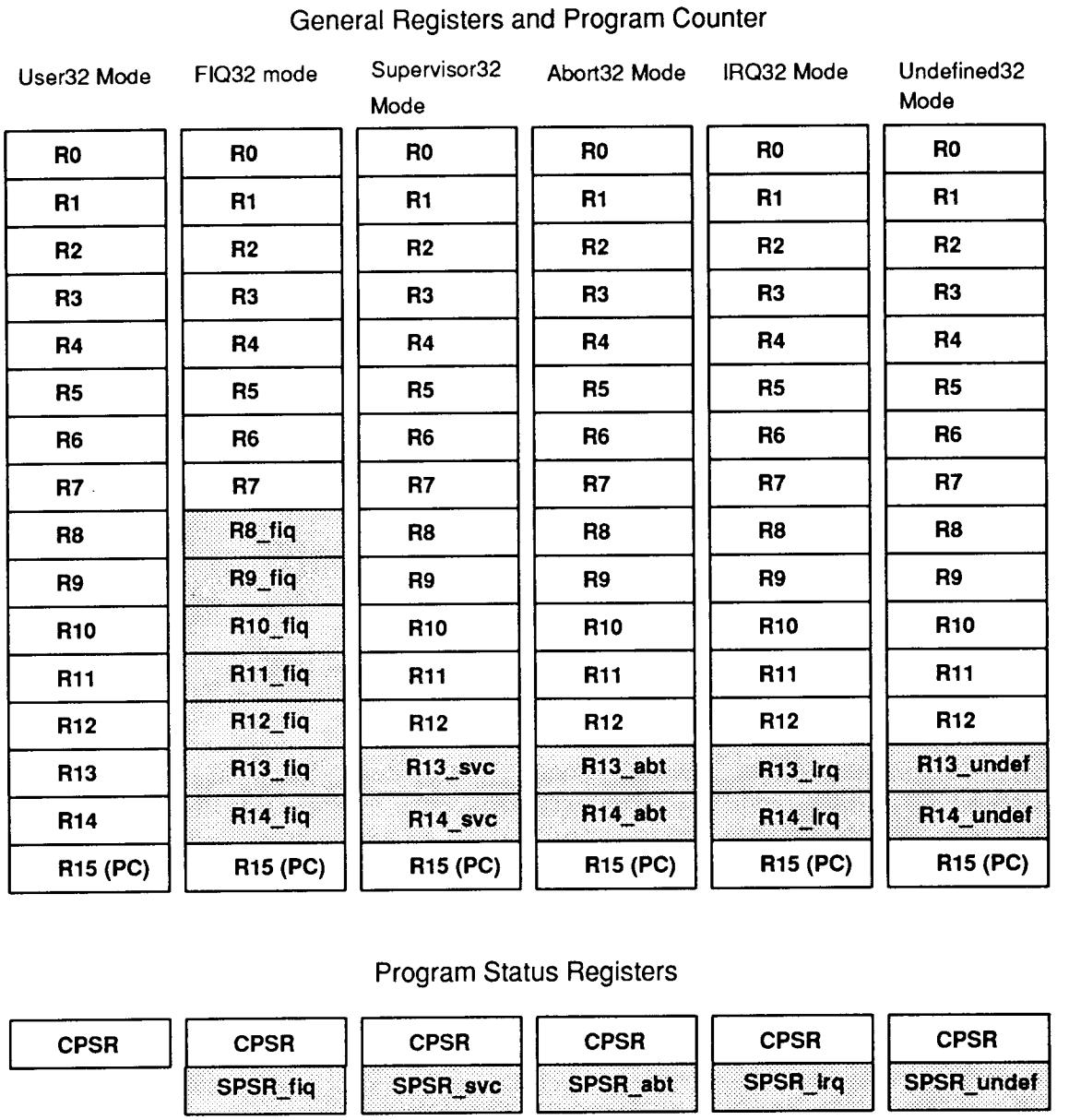


Figure 3: Register Organisation

The I and F bits are the *interrupt disable bits*. The I bit disables IRQ interrupts when it is set and the F bit disables FIQ interrupts when it is set. The M0, M1, M2, M3 and M4 bits (M[4:0]) are the *mode bits*, and these determine the mode in which the processor operates. The interpretation of the mode bits is shown below in Table 1. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described shall be used.

The bottom 28 bits of a PSR (incorporating I, F and M[4:0]) are known collectively as the *control bits*. The control bits will change when an exception arises and in addition can be manipulated by software when the processor is in a privileged mode. Unused bits in the PSRs are reserved and their state shall be preserved when changing the flag or control bits. Programs shall not rely on specific values from the

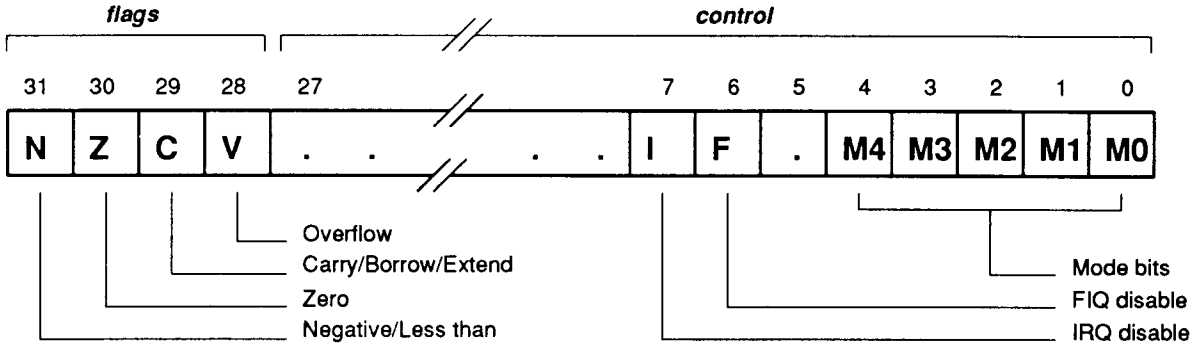


Figure 4: Format of the Program Status Registers (PSRs)

reserved bits when checking the PSR status, since they may read as one or zero in future processors.

M[4:0]	Mode	Accessible register set		
10000	usr	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	svc	PC, R14_svc..R13_svc,	R12..R0	CPSR, SPSR_svc
10111	abt	PC, R14_abt..R13_abt,	R12..R0	CPSR, SPSR_abt
11011	und	PC, R14_und..R13_und,	R12..R0	CPSR, SPSR_und

Table 1: The Mode Bits

6.3 Exceptions

Exceptions arise whenever there is a need for the normal flow of program execution to be broken, so that (for example) the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. Many exceptions may arise at the same time.

ARM60 handles exceptions by making 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 mode bits in the CPSR bits are forced to a value which depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a re-entrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt; 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. The priorities are listed later in this chapter.

6.3.1 FIQ

The FIQ (Fast Interrupt reQuest) exception is externally generated by taking the **Nfiq** input LOW. This input can accept asynchronous transitions, and is delayed by one clock cycle for synchronisation before it can affect the processor execution flow. It 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 (thus minimising the overhead of context switching). The FIQ exception may be disabled by setting the F flag in the CPSR (but note that this is not possible from User mode). If the F flag is clear, **ARM60** checks for a LOW level on the output of the FIQ synchroniser at the end of each instruction.

When a FIQ is detected, **ARM60** performs the following:

(1) Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq

(2) Forces M[4:0]=%10001 (FIQ mode) and sets the F and I bits in the CPSR

(3) Forces the PC to fetch the next instruction from address &1C

To return normally from FIQ, use SUBS PC, R14_fiq,#4 which will restore both the PC (from R14) and the CPSR (from SPSR_fiq) and resume execution of the interrupted code.

6.3.2 IRQ

The IRQ (Interrupt ReQuest) exception is a normal interrupt caused by a LOW level on the **Nirq** input. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the CPSR (but note that this is not possible from User mode). If the I flag is clear, **ARM60** checks for a LOW level on the output of the IRQ synchroniser at the end of each instruction.

When an IRQ is detected, **ARM60** performs the following:

(1) Saves the address of the next instruction to be executed plus 4 in R14_irq; saves CPSR in SPSR_irq

(2) Forces M[4:0]=%10010 (IRQ mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &18

To return normally from IRQ, use SUBS PC,R14_irq,#4 which will restore both the PC and the CPSR and resume execution of the interrupted code.

6.3.3 Abort

The **abort** input comes from an external Memory Management system, and indicates that the current memory access cannot be completed. For instance, in a virtual memory system the data corresponding to the current address may have been moved out of memory onto a disc, and considerable processor activity may be required to recover the data before the access can be performed successfully. **ARM60** checks for **abort** during memory access (N and S) cycles. When successfully aborted **ARM60** will respond in one of two ways:

(i) If the abort occurred during an instruction prefetch (a *Prefetch Abort*), the prefetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example as a result of a branch being taken while it is in the pipeline, no abort will occur. An abort will take place if the instruction reaches the head of the pipeline and is about to be executed.

(ii) If the abort occurred during a data access (a *Data Abort*), the action depends on the instruction type.

  (a) Single data transfer instructions (LDR, STR) are aborted as though the instruction had not executed if the processor is configured for Early Abort. When configured for Late Abort, these instructions are able to write back modified base registers and the Abort handler must be aware of this.

  (b) The swap instruction (SWP) is aborted as though it had not executed.

  (c) Block data transfer instructions (LDM, STM) complete, and if write-back is set, the base is updated. If the instruction would normally have overwritten the base with data (i.e. LDM with the base in the transfer list), this overwriting is prevented. All register overwriting is prevented after the Abort is indicated, which means in particular that R15 (which is always last to be transferred) is preserved in an aborted LDM instruction.

When either a prefetch or data abort occurs, **ARM60** performs the following:

(1) Saves the address of the aborted instruction plus 4 (for prefetch aborts) or 8 (for data aborts) in R14_abt; saves CPSR in SPSR_abt

(2) Forces M[4:0]=%10111 (Abort mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from either address &0C (prefetch abort) or address &10 (data abort)

To return after fixing the reason for the abort, use SUBS PC,R14_abt,#4 (for a prefetch abort) or SUBS PC,R14_abt,#8 (for a data abort). This will restore both the PC and the CPSR and retry the aborted instruction.

The abort mechanism allows a *demand paged virtual memory system* to be implemented when a suitable memory manager (such as MEMC) is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable the memory manager signals an abort. The processor traps into system software which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

6.3.4 Software interrupt

The software interrupt instruction (SWI) is used for getting into Supervisor mode, usually to request a particular supervisor function. When a SWI is executed, **ARM60** performs the following:

(1) Saves the address of the SWI instruction plus 4 in R14_svc; saves CPSR in SPSR_svc

(2) Forces M[4:0]=%10011 (Supervisor mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &08

To return from a SWI, use MOVS PC,R14_svc. This will restore the PC and CPSR and return to the instruction following the SWI.

6.3.5 Undefined instruction trap

When **ARM60** executes a coprocessor instruction or the Undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor can perform this instruction but is busy at that moment, **ARM60** will wait until the coprocessor is ready. If no coprocessor can handle the instruction **ARM60** will take the undefined instruction trap.

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

When **ARM60** takes the undefined instruction trap it performs the following:

(1) Saves the address of the Undefined or coprocessor instruction plus 4 in R14_und; saves CPSR in SPSR_und

(2) Forces M[4:0]=%11011 (Undefined mode) and sets the I bit in the CPSR

(3) Forces the PC to fetch the next instruction from address &04

To return from this trap after emulating the failed instruction, use MOVS PC,R14_und. This will restore the CPSR and return to the instruction following the undefined instruction.

6.3.6 Reset

When the **Nreset** signal goes LOW, **ARM60** abandons the currently executing instruction and then continues to fetch instructions from memory which it interprets as NOPs.

When **Nreset** goes HIGH again, **ARM60** does the following:

(1) Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and CPSR is not defined.

(2) Forces M[4:0]=%10011 (Supervisor mode) and sets the I and F bits in the CPSR

(3) Forces the PC to fetch the next instruction from address &00

6.3.7 Vector Summary

Address	Exception	Mode on entry
&00000000	Reset	Supervisor
&00000004	Undefined instruction	Undefined
&00000008	Software interrupt	Supervisor
&0000000C	Abort (prefetch)	Abort
&00000010	Abort (data)	Abort
&00000014	-- reserved --	--
&00000018	IRQ	IRQ
&0000001C	FIQ	FIQ

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine. The FIQ routine might reside at &1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

The reserved entry is for an Address Exception vector which is only operative when the processor is configured for a 26 bit program space.

6.3.8 Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they will be handled:

(1) Reset (highest priority)

(2) Data abort

(3) FIQ

(4) IRQ

(5) Prefetch abort

(6) Undefined Instruction, Software interrupt (lowest priority)

Note that not all 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 F flag in the CPSR is clear), **ARM60** will enter the data abort handler and then immediately proceed 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.

6.3.9 Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax*), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). At the end of this time **ARM60** will be executing the instruction at &1C.

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchroniser (*Tsyncmax*), plus the time for the longest instruction to complete (*Tldm*, the longest instruction is load multiple registers), plus the time for address exception or data abort entry (*Texc*), plus the time for FIQ entry (*Tfiq*). At the end of this time **ARM60** will be executing the instruction at 1CH.

Tsyncmax is 3 processor cycles, *Tldm* is 20 cycles, *Texc* is 3 cycles, and *Tfiq* is 2 cycles. The total time is therefore 28 processor cycles. This is just over 1 microsecond in a system which uses a continuous 25

MHz processor clock.

The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time.

The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchroniser (*Tsyncmin*) plus *Tfiq*. This is 4 processor cycles.

7. Instruction Set

7.1 The condition field

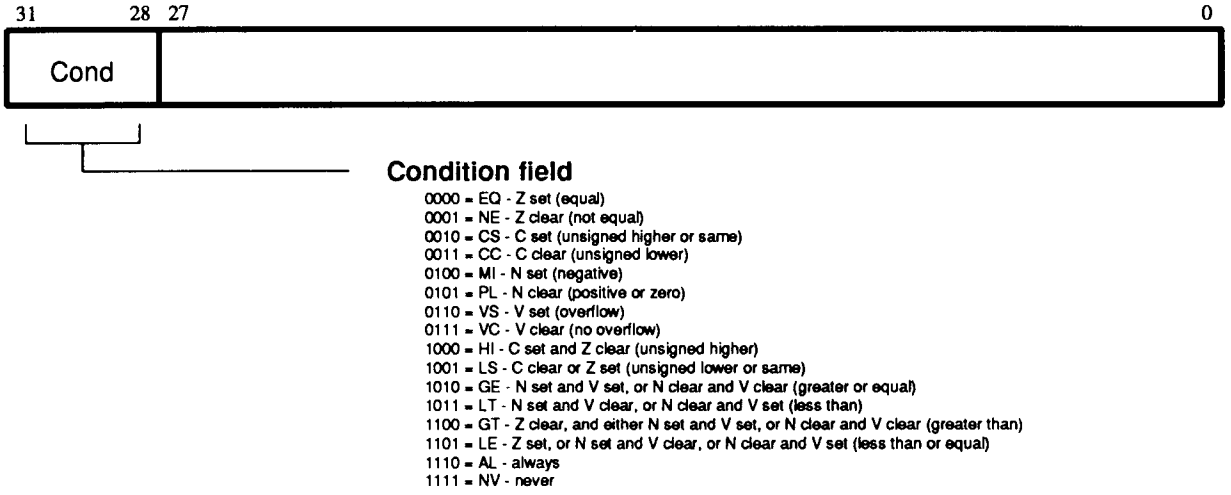


Figure 5: ARM Condition Codes

All **ARM60** instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR. The instruction encoding is shown in Figure 5.

If the *always* (AL) condition is specified, the instruction will be executed irrespective of the flags. The *never* (NV) class of condition codes shall not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though *always* had been specified.

The other condition codes have meanings as detailed in Figure 5, for instance code 0000 (EQual) causes the instruction to be executed only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction will not be executed.

7.2 Branch and branch with link (B, BL)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 6.

Branch instructions contain a signed 2's complement 24 bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction.

Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a Branch with Link type operation is required.

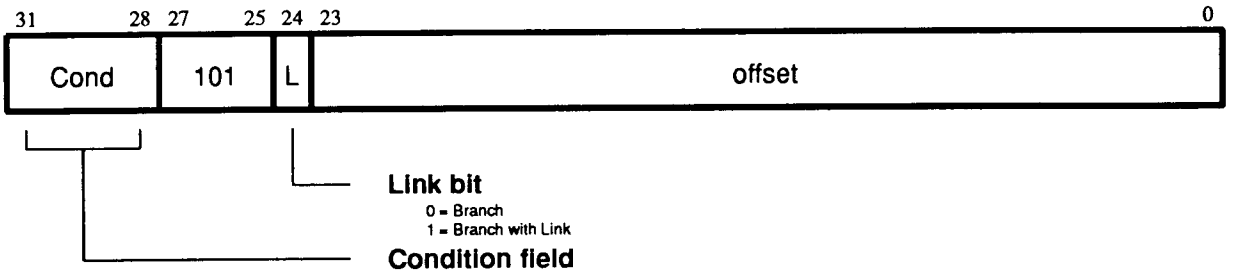


Figure 6: Branch Instructions

7.2.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.

7.2.2 Assembler syntax

B{L}{cond} <expression>

{L} is used to request the Branch with Link form of the instruction. If absent, R14 will not be affected by the instruction.

{cond} is a two-char mnemonic as shown in Figure 5 (EQ, NE, VS etc). If absent then AL (ALways) will be used.

<expression> is the destination. The assembler calculates the offset.

Items in {} are optional. Items in <> must be present.

7.2.3 Examples

```
here BAL here     ; assembles to &EAFFFFFE (note effect of PC offset)

    B there       ; ALways condition used as default

    CMP R1,#0     ; compare R1 with zero and branch to fred if R1
    BEQ fred      ;  was zero otherwise continue to next instruction

    BL sub + ROM ; call subroutine at computed address

    ADDS R1,#1    ; add 1 to register 1, setting CPSR flags on the
    BLCC sub      ; result then call subroutine if the C flag is clear,
                  ; which will be the case unless R1 held &FFFFFFFF
```

7.3 Data processing

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 7.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to

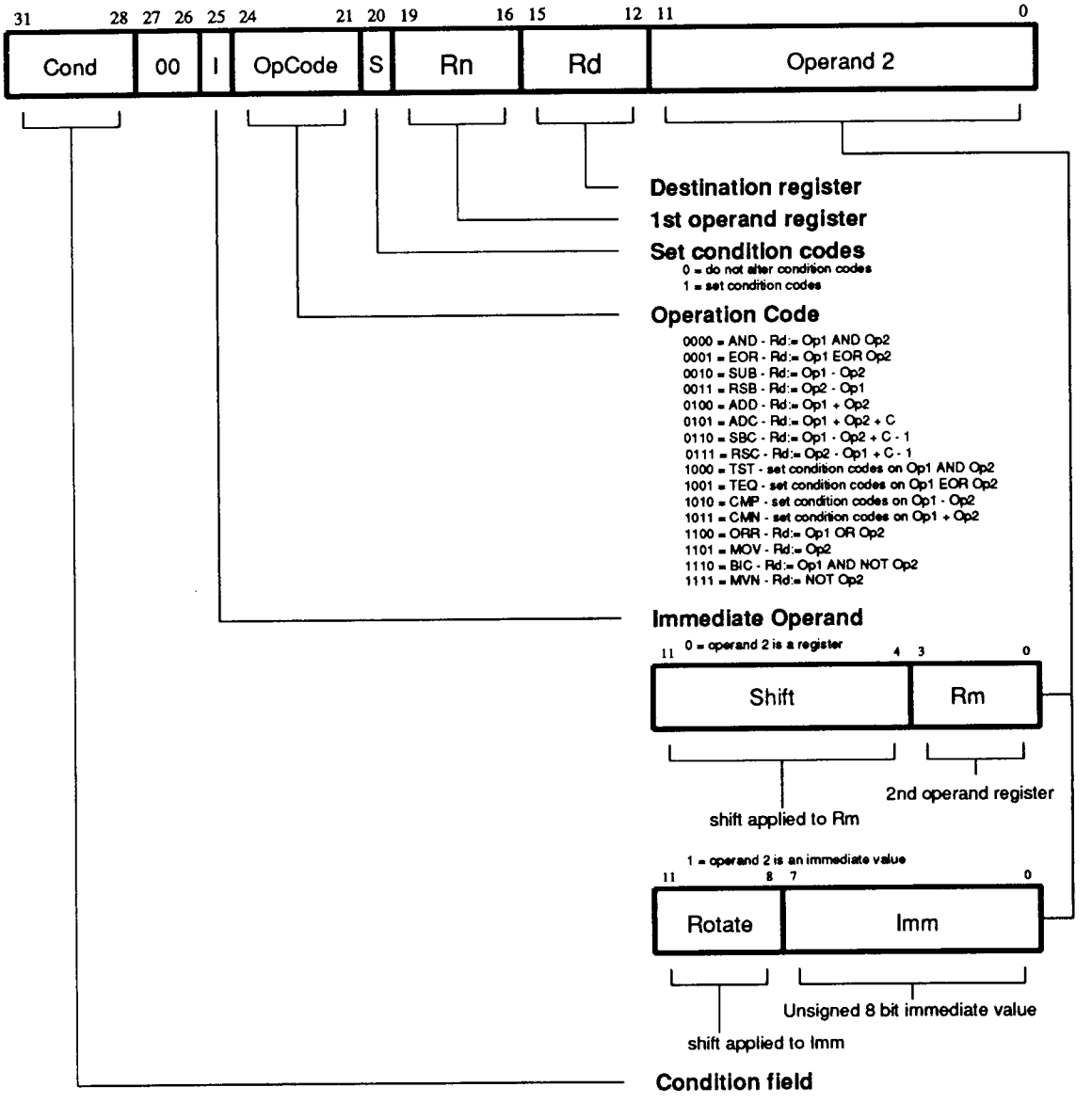


Figure 7: Data Processing Instructions

Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set. The instructions and their effects are listed in Table 2.

7.3.1 CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the CPSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeros, and the N

Assembler Mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry (CPSR C flag)
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Table 2: ARM Data Processing Instructions

flag will be set to the logical value of bit 31 of the result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

7.3.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15). The encoding for the different shift types is shown in Figure 8.

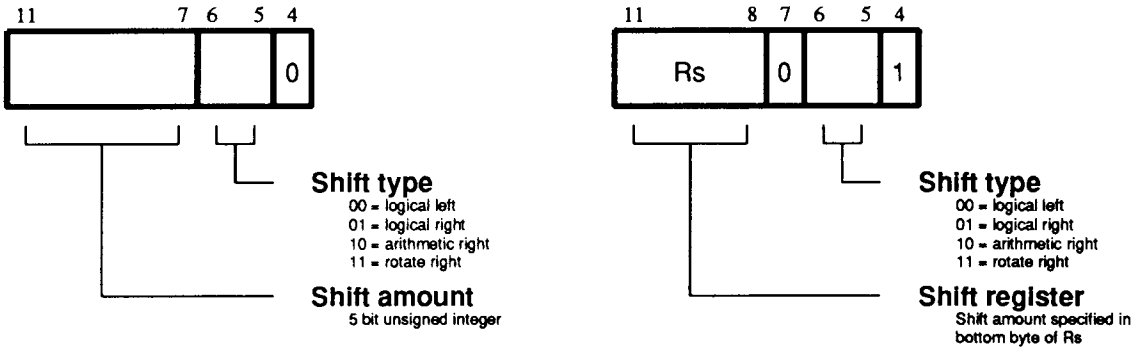


Figure 8: ARM Shift Operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in Figure 9.

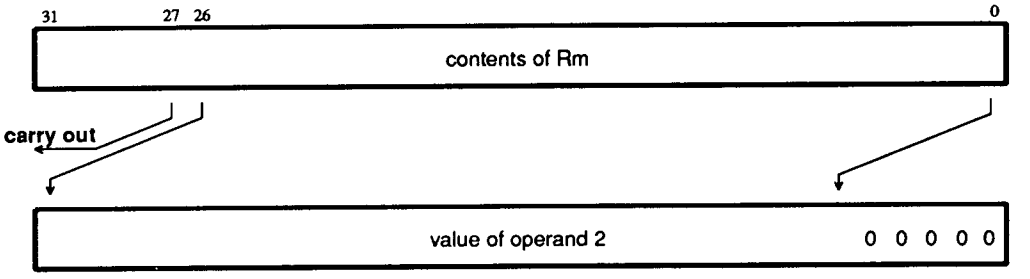


Figure 9: Logical Shift Left

Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in Figure 10.

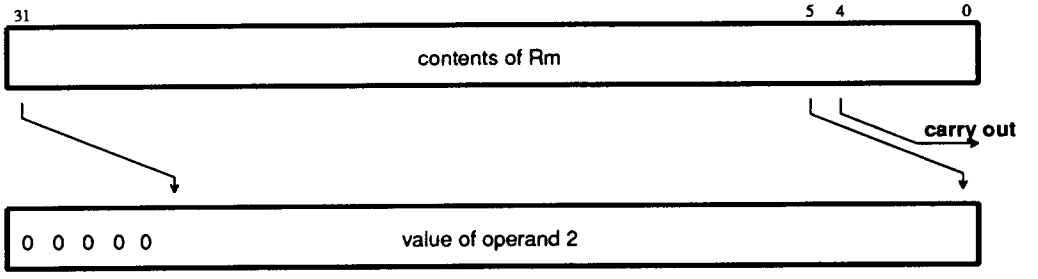


Figure 10: Logical Shift Right

The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in Figure 11.

The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in Figure 12.

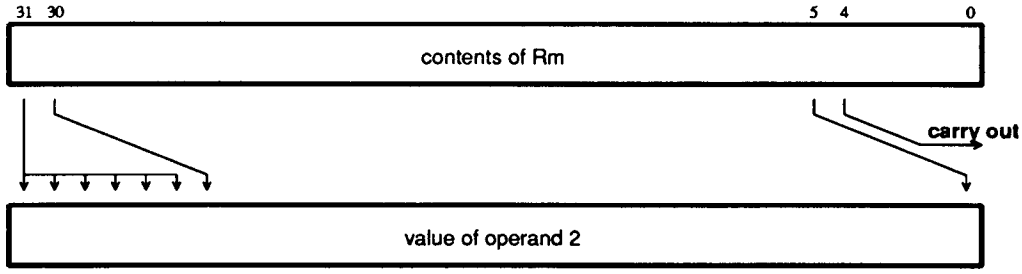


Figure 11: Arithmetic Shift Right

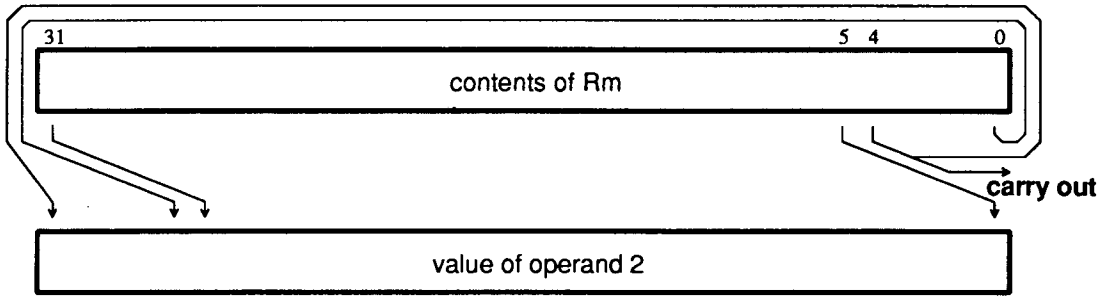


Figure 12: Rotate Right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in Figure 13.

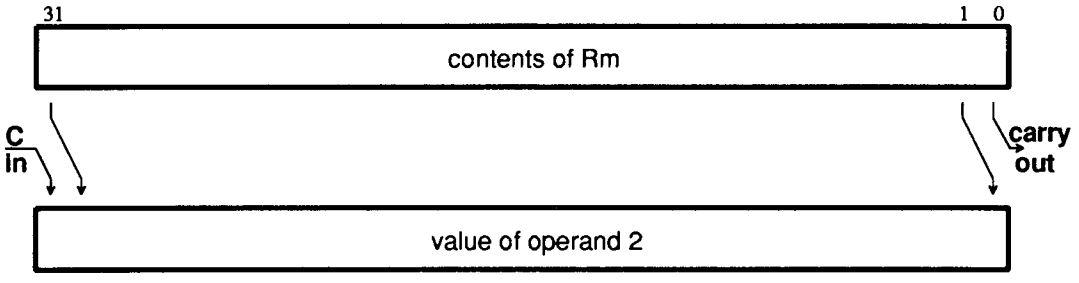


Figure 13: Rotate Right Extended

Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shifting processes described above:

(i) LSL by 32 has result zero, carry out equal to bit 0 of Rm.

(ii) LSL by more than 32 has result zero, carry out zero.

(iii) LSR by 32 has result zero, carry out equal to bit 31 of Rm.

(iv) LSR by more than 32 has result zero, carry out zero.

(v) ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.

(vi) ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.

(vii) ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

7.3.3 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

7.3.4 Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR. This form of instruction shall not be used in User mode.

7.3.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

7.3.6 The TEQ, TST, CMP and CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32 bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

7.3.7 Assembler syntax

(i) MOV,MVN - single operand instructions

<opcode>{cond}{S} Rd,<Op2>

(ii) CMP,CMN,TEQ,TST - instructions which do not produce a result.

<opcode>{cond} Rn,<Op2>

(iii) AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

<opcode>{cond}{S} Rd,Rn,<Op2>

where <Op2> is **Rm{,<shift>}** or **,<#expression>**

{cond} - two-character condition mnemonic, see Figure 5.

{S} - set condition codes if S present (implied for CMP, CMN, TEQ, TST).

Rd, Rn and Rm are expressions evaluating to a register number.

If <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

<shift> is <shiftname> <register> or <shiftname> #expression, or RRX (rotate right one bit with extend).

<shiftname>s are: ASL, LSL, LSR, ASR, ROR.

(ASL is a synonym for LSL, the two assemble to the same code.)

7.3.8 Examples

```
ADDEQ R2,R4,R5        ; if the Z flag is set make R2:=R4+R5

TEQS R4,#3            ; test R4 for equality with 3
                      ; (the S is in fact redundant as the
                      ;  assembler inserts it automatically)

SUB R4,R5,R7,LSR R2   ; logical right shift R7 by the number in
                      ; the bottom byte of R2, subtract the result
                      ; from R5, and put the answer into R4

MOV PC,R14            ; return from subroutine

MOVS PC,R14           ; return from exception & restore CPSR_<mode>
```

7.4 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 14.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in Figure 14.

These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register. The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register. R15 shall not be specified as the source or destination register.

The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32 bit immediate value are written to the top four bits of the relevant PSR.

7.4.1 Operand restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

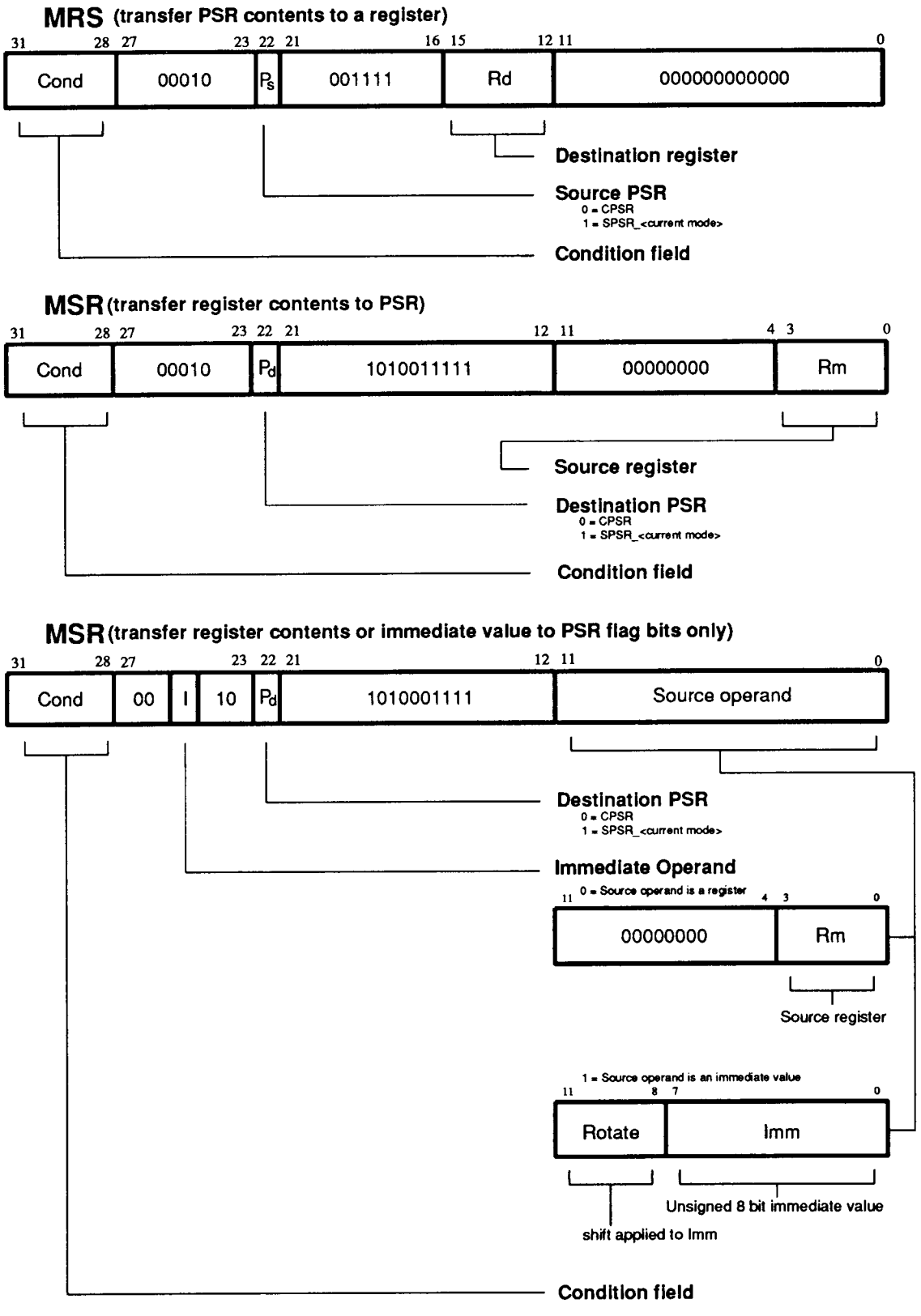


Figure 14: MRS and MSR Instructions

A further restriction is that no attempt shall be made to access an SPSR in User mode, since no such register exists.

7.4.2 Reserved bits

Only eleven bits of the PSR are defined in **ARM60** (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor. To ensure the maximum compatibility between **ARM60** programs and future processors, the following rules should be observed:

(a) The reserved bits shall be preserved when changing the value in a PSR.

(b) Programs shall not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.

e.g. The following sequence performs a mode change:

```
MRS Rtmp,CPSR           ; take a copy of the CPSR
BIC Rtmp,Rtmp,#&1F      ; clear the mode bits
ORR Rtmp,Rtmp,#new_mode ; select new mode
MSR CPSR,Rtmp           ; write back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, an immediate value can be written directly to the flag bits without disturbing the control bits.

e.g. The following instruction sets the N,Z,C & V flags:

```
MSR CPSR_flg,#&F0000000 ; set all the flags regardless of
                        ; their previous state (does not
                        ; affect any control bits)
```

No attempt shall be made to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.

7.4.3 Assembler syntax

(i) MRS - transfer PSR contents to a register

MRS{cond} Rd,<psr>

(ii) MSR - transfer register contents to PSR

MSR{cond} <psr>,Rm

(iii) MSR - transfer register contents to PSR flag bits only

MSR{cond} <psrf>,Rm

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

(iv) MSR - transfer immediate value to PSR flag bits only

MSR{cond} <psrf>,<#expression>

The expression should symbolise a 32 bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

{cond} - two-character condition mnemonic, see Figure 5

Rd and Rm are expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

Where <#expression> is used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.

7.4.4 Examples

In User mode the instructions behave as follows:

```
MSR   CPSR_all,Rm           ; CPSR[31:28] <- Rm[31:28]
MSR   CPSR_flg,Rm           ; CPSR[31:28] <- Rm[31:28]

MSR   CPSR_flg,#&A0000000   ; CPSR[31:28] <- &A
                            ;   (i.e. set N,C; clear Z,V)

MRS   Rd,CPSR               ; Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

```
MSR   CPSR_all,Rm           ; CPSR[31:0]  <- Rm[31:0]
MSR   CPSR_flg,Rm           ; CPSR[31:28] <- Rm[31:28]

MSR   CPSR_flg,#&50000000   ; CPSR[31:28] <- &5
                            ;   (i.e. set Z,V; clear N,C)

MRS   Rd,CPSR               ; Rd[31:0] <- CPSR[31:0]

MSR   SPSR_all,Rm           ; SPSR_<mode>[31:0]  <- Rm[31:0]
MSR   SPSR_flg,Rm           ; SPSR_<mode>[31:28] <- Rm[31:28]

MSR   SPSR_flg,#&C0000000   ; SPSR_<mode>[31:28] <- &C
                            ;   (i.e. set N,Z; clear C,V)

MRS   Rd,SPSR               ; Rd[31:0] <- SPSR_<mode>[31:0]
```

7.5 Multiply and multiply-accumulate (MUL, MLA)

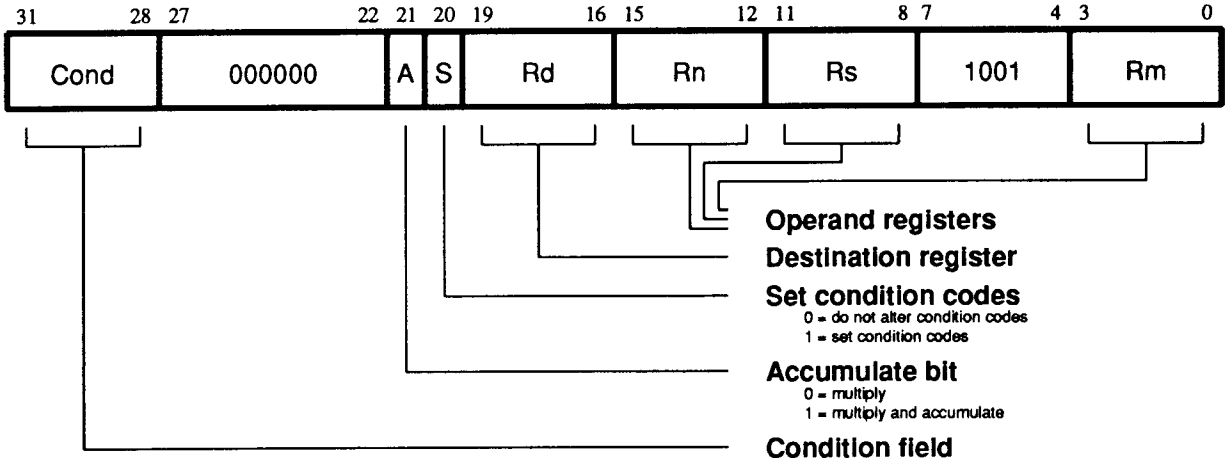


Figure 15: Multiply Instructions

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 15.

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2's complement) or unsigned integers.

7.5.1 Operand restrictions

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result. R15 shall not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

7.5.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to bit 31 of the result, Z is set if and only if the result is zero), the V flag is unaffected by the instruction (as for logical data processing instructions), and the C flag is set to a meaningless value.

7.5.3 Assembler syntax

MUL{cond}{S} Rd,Rm,Rs

MLA{cond}{S} Rd,Rm,Rs,Rn

{cond} - two-character condition mnemonic, see Figure 5

{S} - set condition codes if S present

Rd, Rm, Rs and Rn are expressions evaluating to a register number other than R15.

7.5.4 Examples

```
MUL R1,R2,R3          ; R1:=R2*R3

MLAEQS R1,R2,R3,R4    ; conditionally R1:=R2*R3+R4,
                      ;  setting condition codes
```

The multiply instruction may be used to synthesize higher precision multiplications, for instance to multiply two 32 bit integers and generate a 64 bit result:

```
mul64
    MOV   a1,A,LSR #16    ; a1:= top half of A
    MOV   D,B,LSR #16     ; D := top half of B
    BIC   A,A,a1,LSL #16  ; A := bottom half of A
    BIC   B,B,D,LSL #16   ; B := bottom half of B
    MUL   C,A,B           ; low section of result
    MUL   B,a1,B          ; ) middle sections
    MUL   A,D,A           ; )     of result
    MUL   D,a1,D          ; high section of result
    ADDS  A,B,A           ; add middle sections
                          ;  (couldn't use MLA as we need C correct)
    ADDCS D,D,#&10000     ; carry from above add
    ADDS  C,C,A,LSL #16   ; C is now bottom 32 bits of product
    ADC   D,D,A,LSR #16   ; D is top 32 bits
```

A, B are registers containing the 32 bit integers; C, D are registers for the 64 bit result; a1 is a temporary register. A and B are overwritten during the multiply.

7.6 Single data transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 16.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if 'auto-indexing' is required.

7.6.1 Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces the **Ntrans** signal to go LOW for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this signal.

7.6.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class.

7.6.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an **ARM60** register and memory.

The action of LDR(B) and STR(B) instructions is influenced by the **bigend** configuration signal to the processor. The two possible configurations are described below.

Little Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are

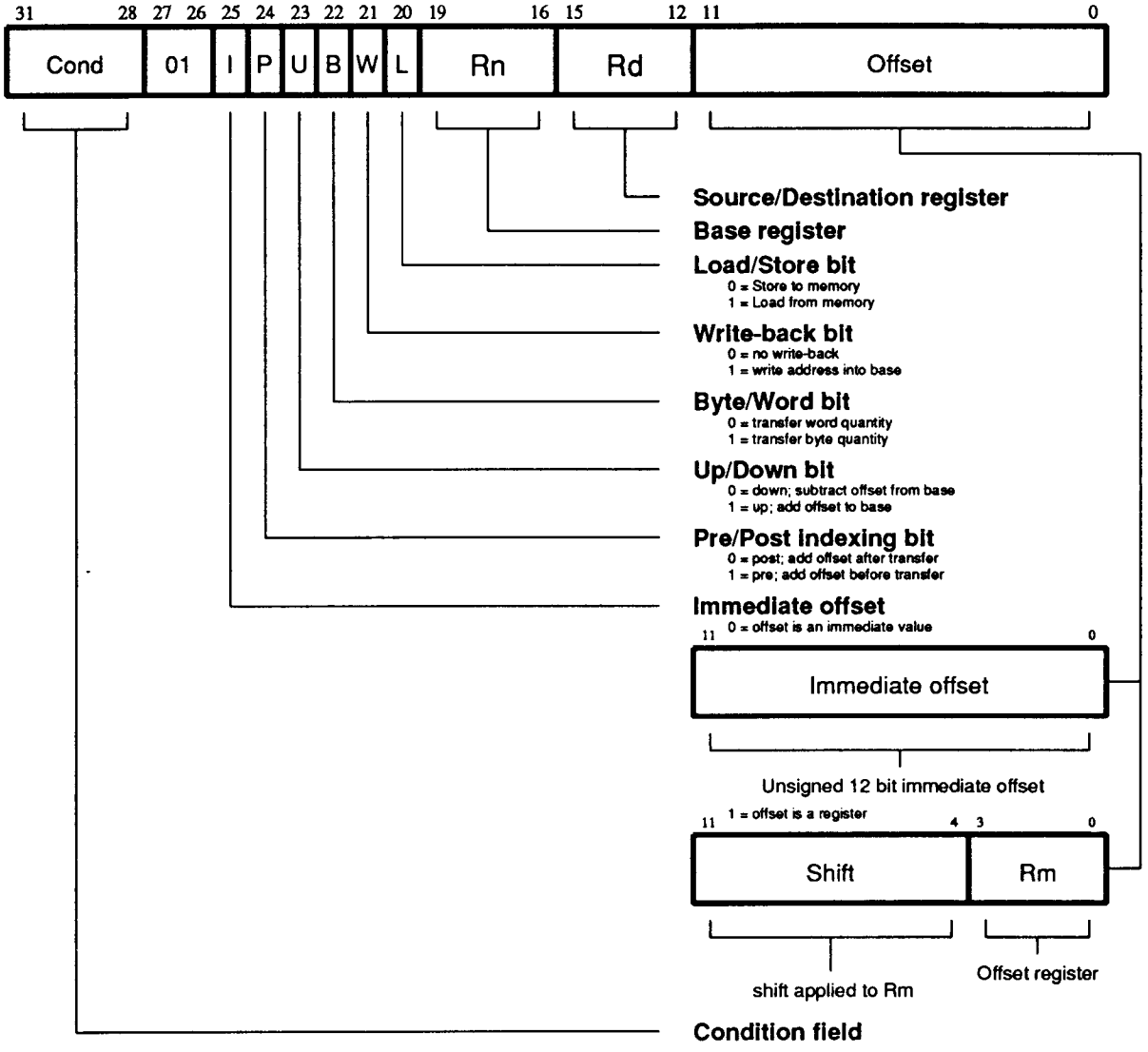


Figure 16: Single Data Transfer Instructions

filled with zeros.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

Big Endian Configuration

A byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0. The external memory system should activate the appropriate byte subsystem to store the data.

A word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

7.6.4 Use of R15

Write-back shall not be specified if R15 is specified as the base register (Rn). When using R15 as the base register one must remember that it contains an address 8 bytes on from the address of the current instruction.

R15 shall not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

7.6.5 Restriction on the use of base register

When configured for late aborts, the following code is very difficult to unwind as the base register, Rn, gets updated before the abort handler is entered. In certain circumstances it may be impossible to calculate the initial value.

```
<LDR|STR> Rd, [Rn],{+/-}Rn{,<shift>}
```

A post-indexed LDR|STR where Rm=Rn shall not be used.

7.6.6 Data Aborts

A transfer to or from a legal address may cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor **abort** input HIGH, whereupon the data transfer instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

ARM60 supports two types of Data Abort processing depending on the **lateabt** configuration input. When configured for Early Aborts, any base register write-back which would have occurred is prevented from happening in the event of an Abort. When configured for Late Aborts, this write-back is allowed to take place and the Abort handler must correct this before allowing the instruction to be re-executed.

7.6.7 Assembler syntax

<LDR|STR>{cond}{B}{T} Rd,<Address>

LDR - load from memory into a register

STR - store from a register into memory

{cond} - two-character condition mnemonic, see Figure 5

{B} - if B is present then byte transfer, otherwise word transfer

{T} - if T is present the W bit will be set in a post-indexed instruction, causing the **Ntrans** signal to go LOW for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

[Rn,{+/-}Rm{,<shift>}]{!} offset of +/- contents of index register, shifted by <shift>.

(iii) A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

[Rn],{+/-}Rm{,<shift>} offset of +/- contents of index register, shifted as by <shift>.

Rn and Rm are expressions evaluating to a valid register number. NOTE if Rn is R15 then the assembler will subtract 8 from the offset value to allow for **ARM60** pipelining. In this case base write-back shall not be specified.

<shift> is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.

{!} write back the base register (set the W bit) if ! is present.

7.6.8 Examples

```
    STR R1,[BASE,INDEX]!     ; store R1 at BASE+INDEX (both of which are
                             ; registers) and write back address to BASE

    STR R1,[BASE],INDEX      ; store R1 at BASE and write back
                             ;  BASE+INDEX to BASE

    LDR R1,[BASE,#16]        ; load R1 from contents of BASE+16.
                             ;  Don't write back

    LDR R1,[BASE,INDEX,LSL #2] ; load R1 from contents of BASE+INDEX*4

    LDREQB R1,[BASE,#5]      ; conditionally load byte at BASE+5 into
                             ;  R1 bits 0 to 7, filling bits 8 to 31
                             ;  with zeros

    STR R1,PLACE             ; generate PC relative offset to address
       .                     ;  PLACE
       .
PLACE
```

7.7 Block data transfer (LDM, STM)

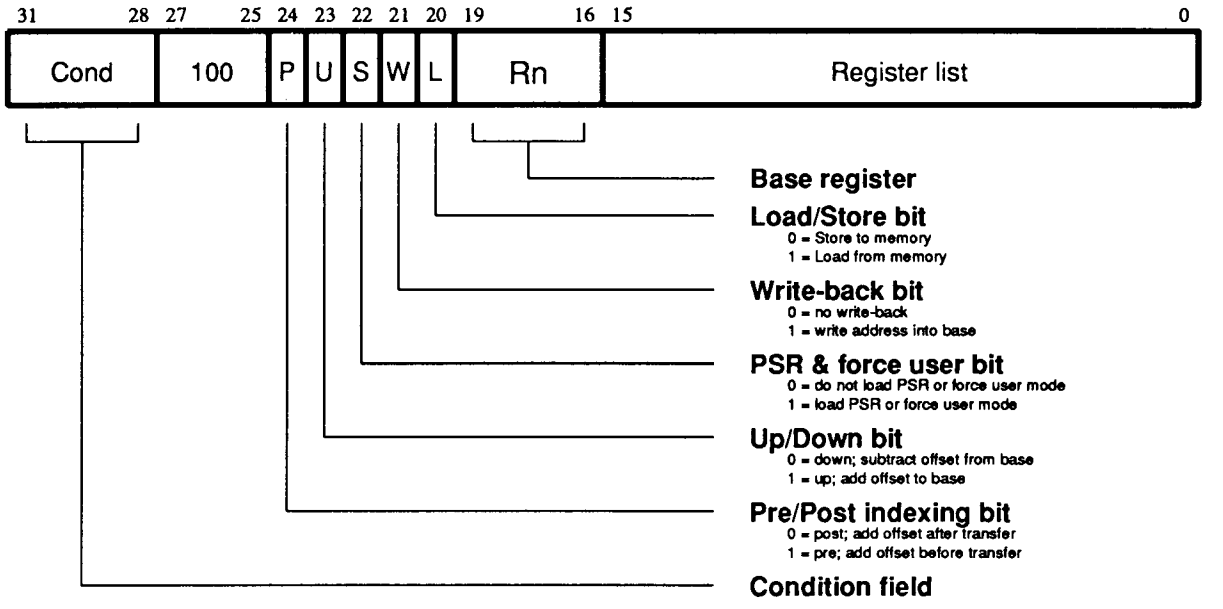


Figure 17: Block Data Transfer Instructions

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 17.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

7.7.1 The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

7.7.2 Addressing modes

The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=1000H and write back of the modified base is required (W=1). The following figures show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 1000H unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.

7.7.3 Address Alignment

The address should normally be a word aligned quantity and non-word aligned addresses do not affect the instruction. However, the bottom 2 bits of the address will appear on **a[1:0]** and might be interpreted by the memory system.

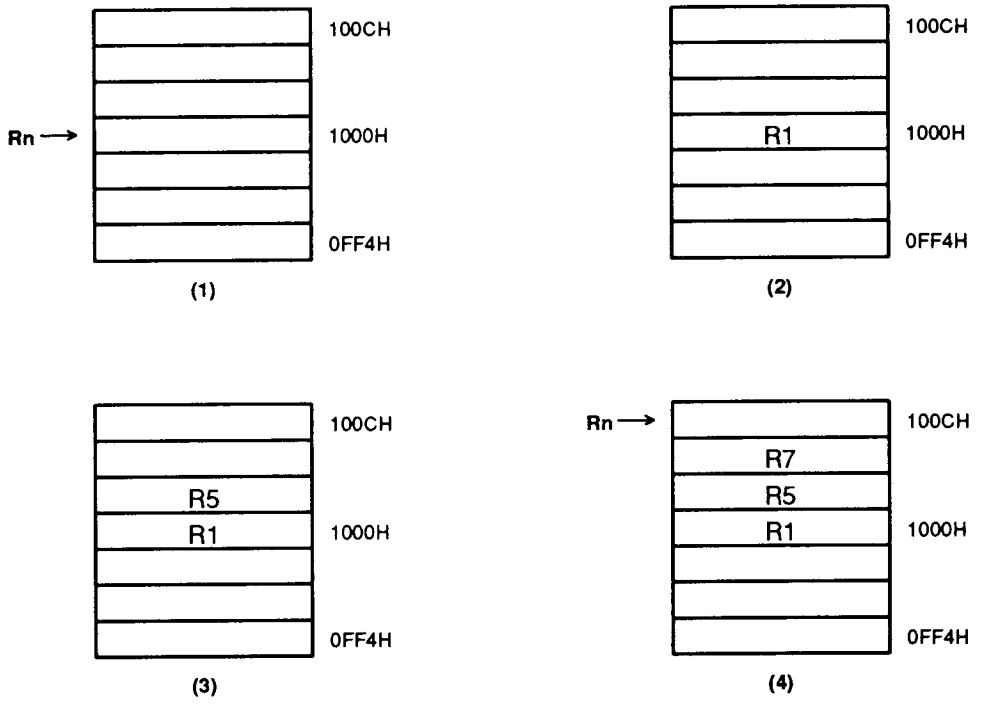


Figure 18: Post-increment addressing

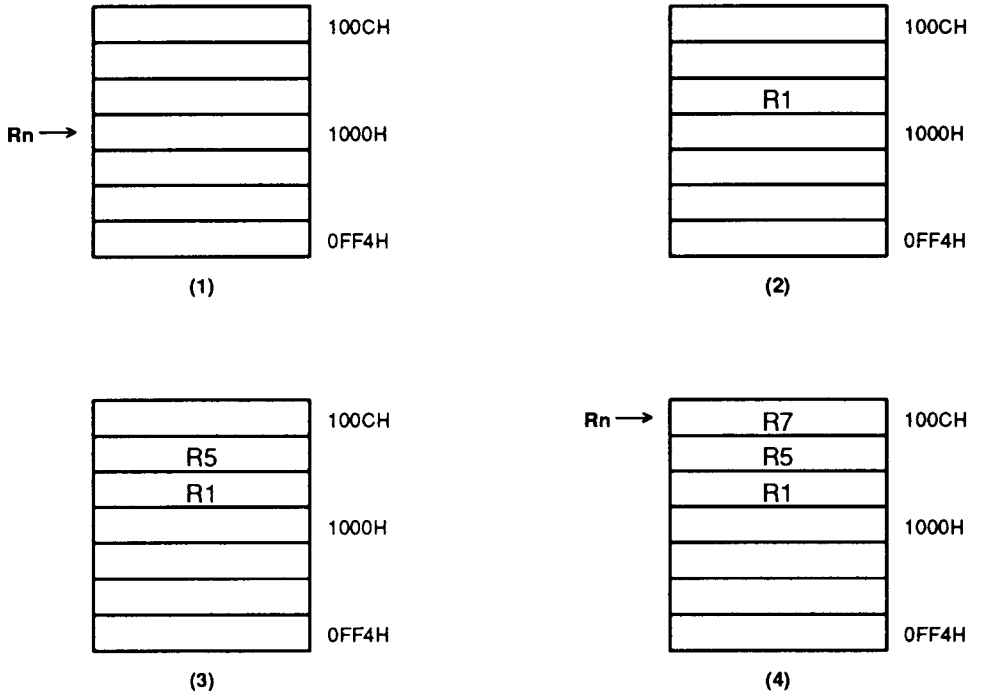


Figure 19: Pre-increment addressing

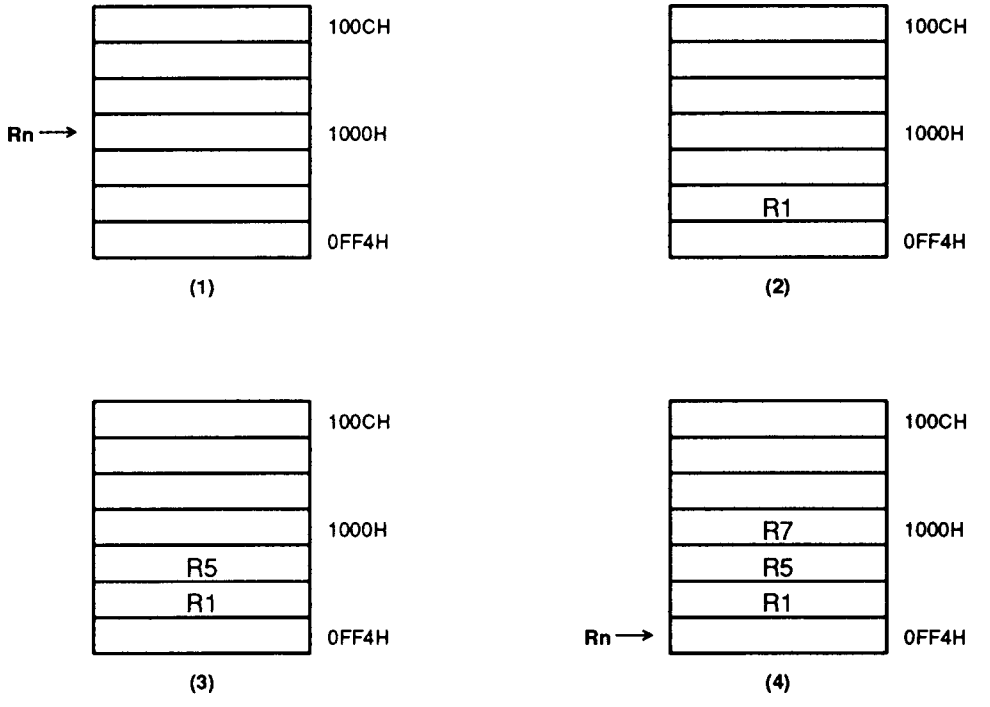


Figure 20: Post-decrement addressing

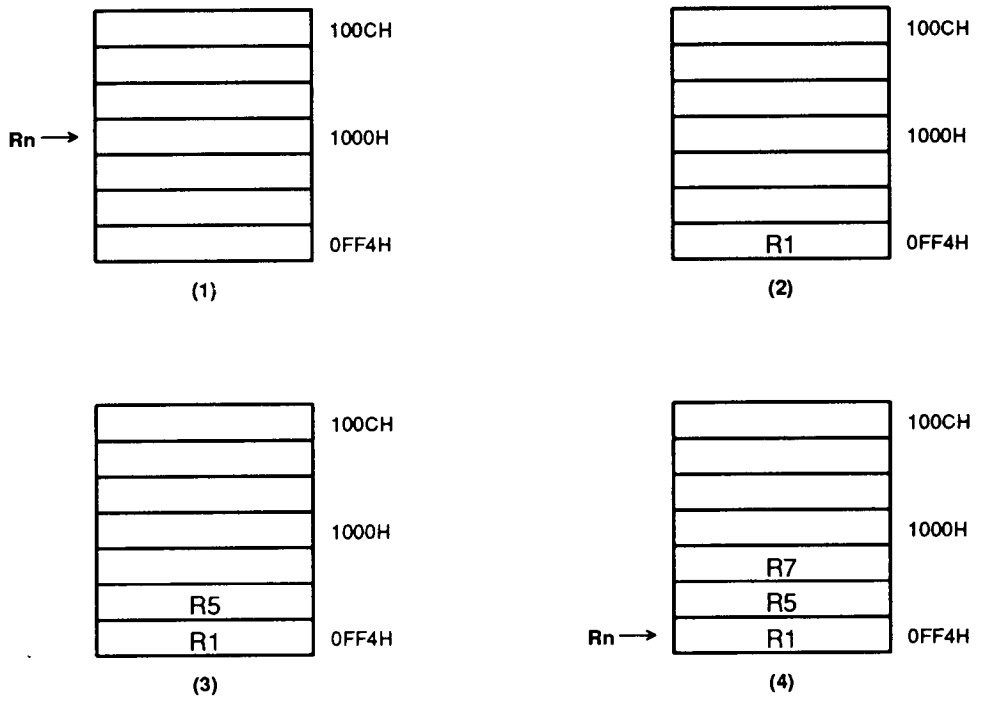


Figure 21: Pre-decrement addressing

7.7.4 Use of the S bit

When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

LDM with R15 in transfer list and S bit set (Mode changes)

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

STM with R15 in transfer list and S bit set (User bank transfer)

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

R15 not in list and S bit set (User bank transfer)

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a NOP after the LDM will ensure safety).

7.7.5 Use of R15 as the base

R15 shall not be used as the base register in any LDM or STM instruction.

7.7.6 Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes

storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. A LDM will always overwrite the updated base if the base is in the list.

7.7.7 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **abort** signal HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if **ARM60** is to be used in a virtual memory system.

The state of the **lateabt** configuration input does not affect the behaviour of LDM and STM instructions in the event of an Abort exception.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, **ARM60** takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When **ARM60** detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

(i) Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.

(ii) The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

7.7.8 Assembler syntax

<LDM|STM>{cond}<FD|ED|FA|EA|IA|IB|DA|DB> Rn{!},<Rlist>{^}

{cond} - two character condition mnemonic, see Figure 5

Rn is an expression evaluating to a valid register number

<Rlist> is a list of registers and register ranges enclosed in {} (eg {R0,R2-R7,R10})

{!} if present requests write-back (W=1), otherwise W=0

{^} if present set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalences between the names and the values of the bits in the instruction are:

name	stack	other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a "full" or "empty" stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

IA, IB, DA, DB allow control when LDM/STM are not being used for stacks and simply mean Increment After, Increment Before, Decrement After, Decrement Before.

7.7.9 Examples

```
LDMFD SP!,{R0,R1,R2}  ; unstack 3 registers

STMIA BASE,{R0-R15}   ; save all registers

LDMFD SP!,{R15}       ; R15 <- (SP), CPSR unchanged
LDMFD SP!,{R15}^      ; R15 <- (SP), CPSR <- SPSR_mode (allowed only
                      ;  in privileged modes)

STMFD R13,{R0-R14}^   ; Save user mode regs on stack (allowed only
                      ;  in privileged modes)
```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```
STMED SP!,{R0-R3,R14}  ; save R0 to R3 to use as workspace
                       ;  and R14 for returning

BL somewhere           ; this nested call will overwrite R14

LDMED SP!,{R0-R3,R15}  ; restore workspace and return
```

7.8 Single data swap (SWP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 22.

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are "locked" together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register may be specified as both the source and destination.

The **lock** output goes HIGH for the duration of the read and write operations to signal to the external memory manager that they are locked together, and should be allowed to complete without interruption. This is important in multi-processor systems where the swap instruction is the only indivisible instruction which may be used to implement semaphores; control of the memory must not be removed from a

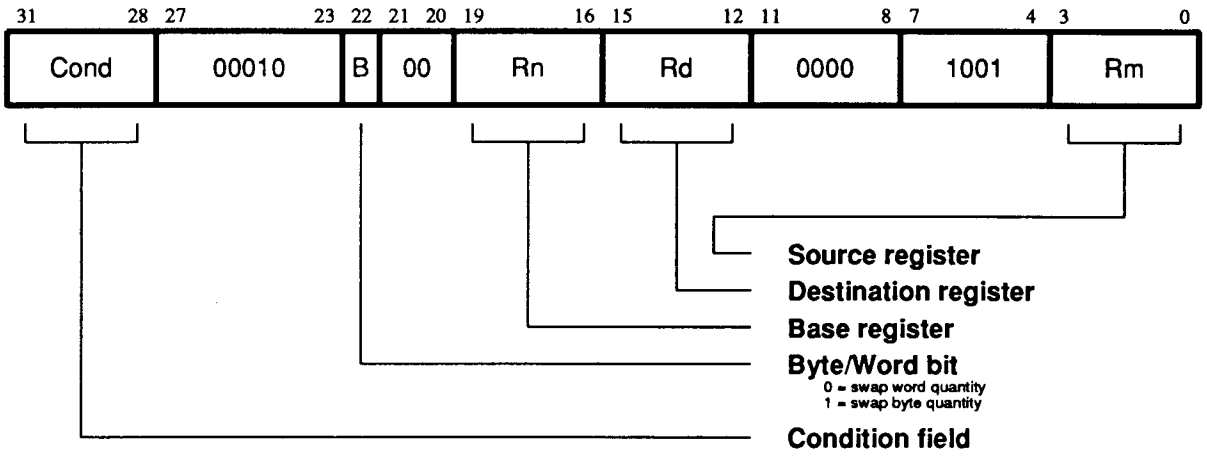


Figure 22: ARM Swap Instruction

processor while it is performing a locked operation.

7.8.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an **ARM60** register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

7.8.2 Use of R15

R15 shall not be used as an operand (Rd, Rn or Rs) in a SWP instruction.

7.8.3 Data Aborts

If the address used for the swap is unacceptable to a memory management system, the memory manager can flag the problem by driving **abort** HIGH. This can happen on either the read or the write cycle (or both), and in either case, the data swap instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

Because no base register write-back is allowed, the behaviour of an aborted SWP instruction is the same regardless of the state of the **lateabt** configuration input.

7.8.4 Assembler syntax

<SWP>{cond}{B} Rd,Rm,[Rn]

{cond} - two-character condition mnemonic, see Figure 5

{B} - if B is present then byte transfer, otherwise word transfer

Rd,Rm,Rn are expressions evaluating to valid register numbers

7.8.5 Examples

```
SWP R0,R1,[BASE]          ; load R0 with the contents of BASE, and
                          ; store R1 at BASE

SWPB R2,R3,[BASE]         ; load R2 with the byte at BASE, and
                          ; store bits 0 to 7 of R3 at BASE

SWPEQ R0,R0,[BASE]        ; conditionally swap the contents of BASE
                          ; with R0
```

7.9 Software interrupt (SWI)

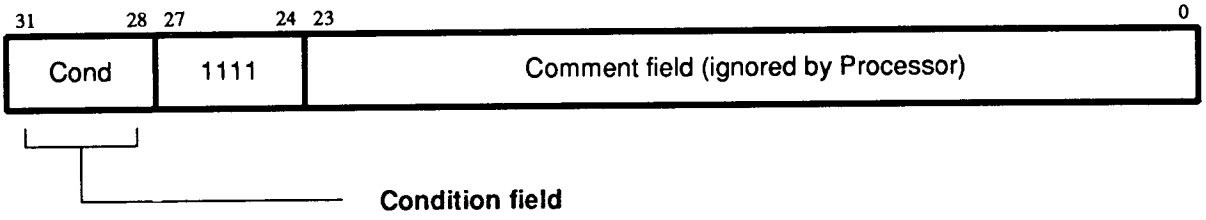


Figure 23: Software Interrupt Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 23.

The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change and the PC is then forced to a fixed value (&08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

7.9.1 Return from the supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

7.9.2 Comment field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

7.9.3 Assembler syntax

SWI{cond} <expression>

{cond} - two character condition mnemonic, see Figure 5

<expression> is evaluated and placed in the comment field (which is ignored by **ARM60**).

7.9.4 Examples

```
SWI    ReadC              ; get next character from read stream

SWI    WriteI+"k"         ; output a "k" to the write stream

SWINE 0                   ; conditionally call supervisor
                          ;  with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
&08 B Supervisor          ; SWI entry point

EntryTable                ; addresses of supervisor routines
    & ZeroRtn
    & ReadCRtn
    & WriteIRtn
      ...

Zero    * 0
ReadC   * 256
WriteI  * 512

Supervisor

; SWI has routine required in bits 8-23 and data (if any) in bits 0-7.
; Assumes R13_svc points to a suitable stack

    STM R13,{R0-R2,R14}     ; save work registers and return address
    LDR R0,[R14,#-4]        ; get SWI instruction
    BIC R0,R0,#&FF000000    ; clear top 8 bits
    MOV R1,R0,LSR #8        ; get routine offset
    ADR R2,EntryTable       ; get start address of entry table
    LDR R15,[R2,R1,LSL #2] ; branch to appropriate routine

WriteIRtn                 ; enter with character in R0 bits 0-7
 .......
 LDM R13,{R0-R2,R15}^     ; restore workspace and return
```

7.10 Coprocessor data operations (CDP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 24.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to **ARM60**, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other **ARM60** activity allowing the coprocessor and **ARM60** to perform independent tasks in parallel.

7.10.1 The Coprocessor fields

Only bit 4 and bits 24 to 31 are significant to **ARM60**; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

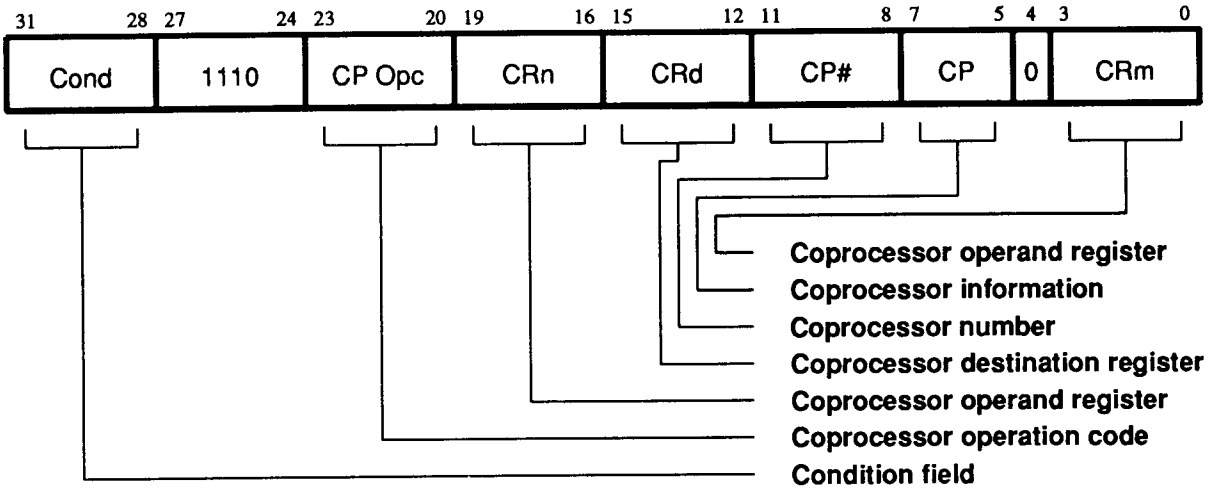


Figure 24: Coprocessor Data Operations

7.10.2 Assembler syntax

CDP{cond} CP#,<expression1>,CRd,CRn,CRm{,<expression2>}

{cond} - two character condition mnemonic, see Figure 5

CP# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

CRd, CRn and CRm are expressions evaluating to a valid coprocessor register number

<expression2> - where present is evaluated to a constant and placed in the CP field

7.10.3 Examples

```
CDP 1,10,CR1,CR2,CR3     ; request coproc 1 to do operation 10
                         ;  on CR2 and CR3, and put the result in CR1

CDPEQ 2,5,CR1,CR2,CR3,2  ; if Z flag is set request coproc 2 to do
                         ;  operation 5 (type 2) on CR2 and CR3,
                         ;  and put the result in CR1
```

7.10.4 ARM2 CDP instruction

The implementation of the CDP instruction on the ARM2 processor causes a Software Interrupt (SWI) to take the Undefined Instruction trap if the SWI was the next instruction after the CDP. The following code sequence should therefore be avoided.

```
CDP
SWI
```

7.11 Coprocessor data transfers (LDC, STC)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 25.

This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessor's registers directly to memory. **ARM60** is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

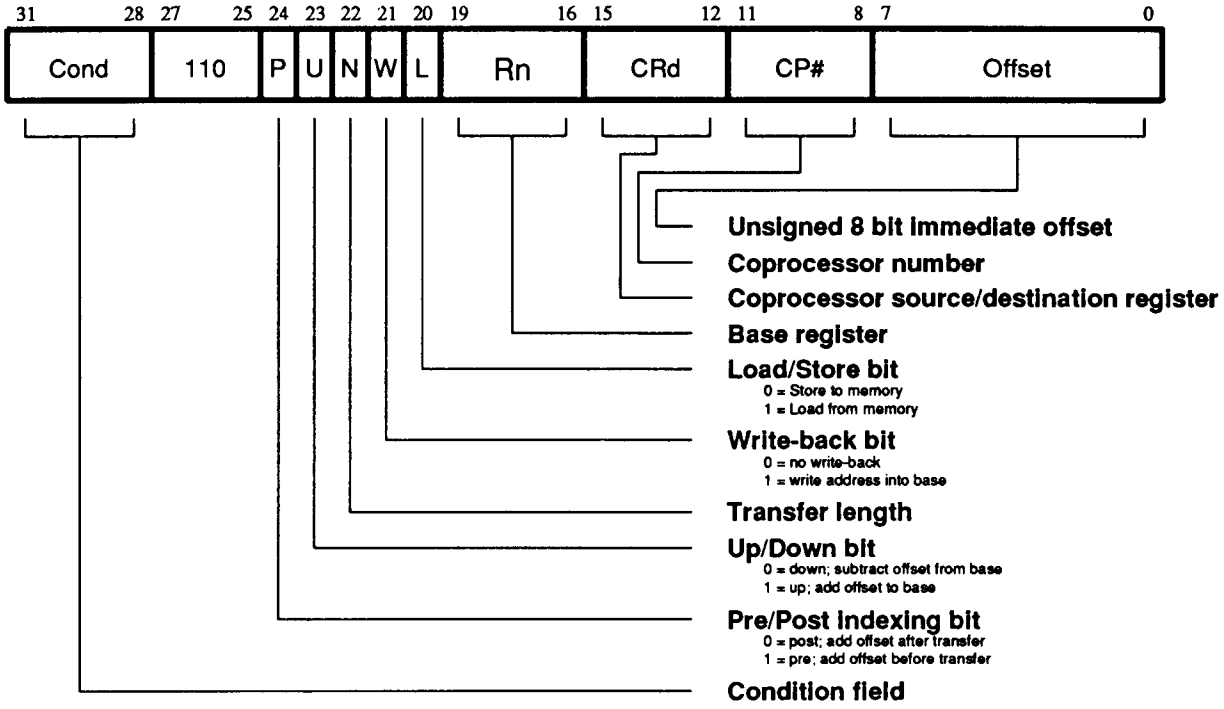


Figure 25: Coprocessor Data Transfer Instructions

7.11.1 The Coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options. For instance N=0 could select the transfer of a single register, and N=1 could select the transfer of all the registers for context switching.

7.11.2 Addressing modes

ARM60 is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0). Note that post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

7.11.3 Address Alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on **a[1:0]** and might be interpreted by the memory system.

7.11.4 Use of R15

If Rn is R15, the value used will be the address of the instruction plus 8 bytes. Base write-back to R15 shall not be specified.

7.11.5 Data aborts

If the address is legal but the memory manager generates an abort, the data abort trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

The state of the **lateabt** configuration input does not affect the behaviour of LDC and STC instructions in the event of an Abort exception.

7.11.6 Assembler syntax

<LDC|STC>{cond}{L} CP#,CRd,<Address>

LDC - load from memory to coprocessor

STC - store from coprocessor to memory

{L} - when present perform long transfer (N=1), otherwise perform short transfer (N=0)

{cond} - two character condition mnemonic, see Figure 5

CP# - the unique number of the required coprocessor

CRd is an expression evaluating to a valid coprocessor register number

<Address> can be:

(i) An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

(ii) A pre-indexed addressing specification:

[Rn] offset of zero

[Rn,<#expression>]{!} offset of <expression> bytes

(iii) A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

Rn is an expression evaluating to a valid **ARM60** register number. NOTE if Rn is R15 then the assembler will subtract 8 from the offset value to allow for **ARM60** pipelining.

{!} write back the base register (set the W bit) if ! is present

7.11.7 Examples

```
LDC 1,CR2,table          ; load CR2 of coproc 1 from address table,
                         ;  using a PC relative address.

STCEQL 2,CR3,[R5,#24]! ; conditionally store CR3 of coproc 2 into
                         ;  an address 24 bytes up from R5, write this
                         ;  address back into R5, and use long transfer
                         ;  option (probably to store multiple words)
```

Note that though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

7.12 Coprocessor register transfers (MRC, MCR)

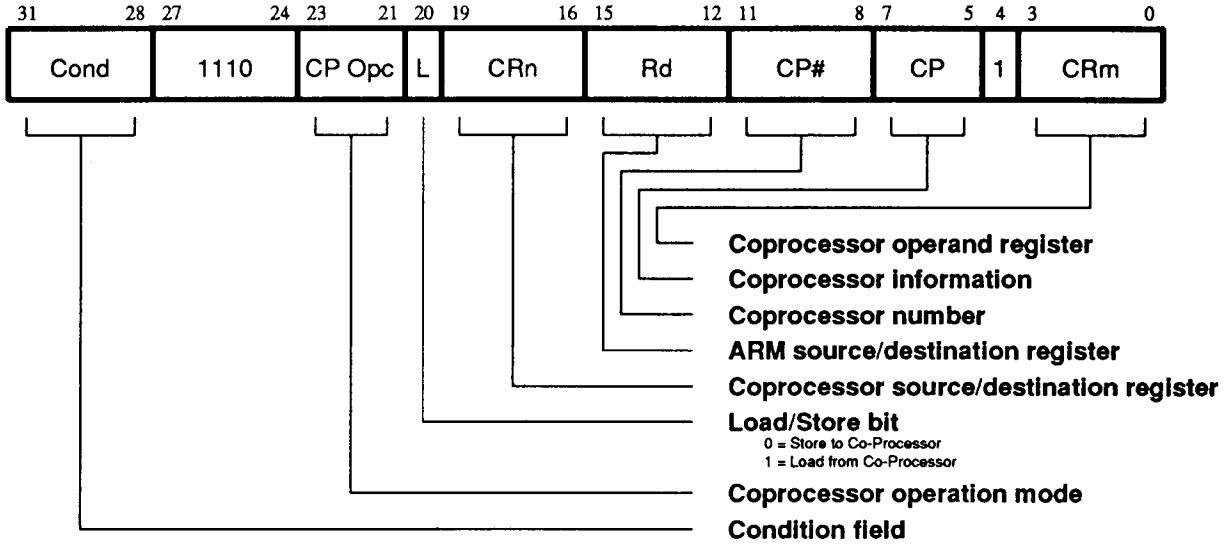


Figure 26: Coprocessor Register Transfer Instructions

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in Figure 26.

This class of instruction is used to communicate information directly between **ARM60** and a coprocessor. An example of a coprocessor to **ARM60** register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32 bit integer within the coprocessor, and the result is then transferred to an **ARM60** register. A FLOAT of a 32 bit value in an **ARM60** register into a floating point value within the coprocessor illustrates the use of an **ARM60** register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the **ARM60** CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

7.12.1 The Coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon to respond.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

7.12.2 Transfers to R15

When a coprocessor register transfer to **ARM60** has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

7.12.3 Transfers from R15

A coprocessor register transfer from **ARM60** with R15 as the source register will store the PC+12.

7.12.4 Assembler syntax

<MCR|MRC>{cond} CP#,<expression1>,Rd,CRn,CRm{,<expression2>}

MRC - move from coprocessor to **ARM60** register (L=1)

MCR - move from **ARM60** register to coprocessor (L=0)

{cond} - two character condition mnemonic, see Figure 5.

CP# - the unique number of the required coprocessor

<expression1> - evaluated to a constant and placed in the CP Opc field

Rd is an expression evaluating to a valid **ARM60** register number

CRn and CRm are expressions evaluating to a valid coprocessor register number

<expression2> - where present is evaluated to a constant and placed in the CP field

7.12.5 Examples

```
MRC 2,5,R3,CR5,CR6      ; request coproc 2 to perform operation 5
                        ;  on CR5 and CR6, and transfer the (single
                        ;  32 bit word) result back to R3

MCR 6,0,R4,CR6          ; request coproc 6 to perform operation 0
                        ;  on R4 and place the result in CR6

MRCEQ 3,9,R3,CR5,CR6,2  ; conditionally request coproc 2 to perform
                        ;  operation 9 (type 2) on CR5 and CR6, and
                        ;  transfer the result back to R3
```

7.13 Undefined instruction

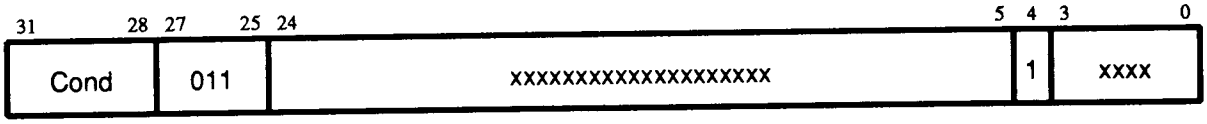


Figure 27: Undefined Instruction

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in Figure 27.

If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering this instruction to any coprocessors which may be present, and all coprocessors must refuse to accept it by driving **cpa** and **cpb** HIGH.

7.13.1 Assembler syntax

At present the assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction shall not be used.

7.14 Instruction Set Summary

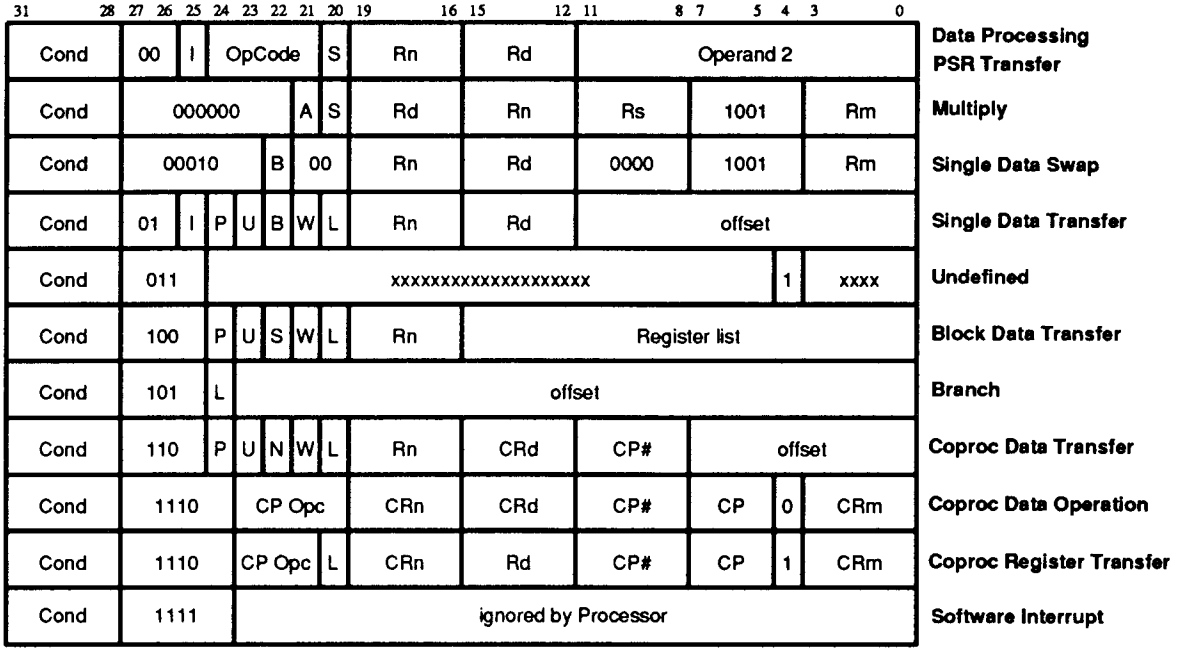


Figure 28: Instruction Set Summary

NOTE: some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a Multiply instruction with bit 5 or bit 6 changed to a 1. These instructions shall not be used, as their action may change in future ARM implementations. The summary is shown in Figure 28.

7.15 Instruction set examples

The following examples show ways in which the basic **ARM60** instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

7.15.1 Using the conditional instructions

(1) using conditionals for logical OR

```
        CMP    Rn,#p  ;IF Rn=p OR Rm=q THEN GOTO Label
        BEQ    Label
        CMP    Rm,#q
        BEQ    Label
```

can be replaced by

```
        CMP    Rn,#p
        CMPNE  Rm,#q  ;if condition not satisfied try other test
        BEQ    Label
```

(2) absolute value

```
        TEQ    Rn,#0            ;test sign
        RSBMI  Rn,Rn,#0         ;and 2's complement if necessary
```

(3) multiplication by 4, 5 or 6 (run time)

```
        MOV    Rc,Ra,LSL #2     ;multiply by 4
        CMP    Rb,#5            ;test value
        ADDCS  Rc,Rc,Ra         ;complete multiply by 5
        ADDHI  Rc,Rc,Ra         ;complete multiply by 6
```

(4) combining discrete and range tests

```
        TEQ    Rc,#127          ;discrete test
        CMPNE  Rc,#" "-1        ;range test
        MOVLS  Rc,#"."          ;IF   Rc<=" " OR Rc=CHR$127
                                ;THEN Rc:="."
```

(5) division and remainder

```
; enter with numbers in Ra and Rb
;
        MOV    Rcnt,#1          ;bit to control the division
Div1    CMP    Rb,#&80000000    ;move Rb until greater than Ra
        CMPCC  Rb,Ra
        MOVCC  Rb,Rb,ASL #1
        MOVCC  Rcnt,Rcnt,ASL #1
        BCC    Div1
        MOV    Rc,#0
Div2    CMP    Ra,Rb            ;test for possible subtraction
        SUBCS  Ra,Ra,Rb         ;subtract if ok
        ADDCS  Rc,Rc,Rcnt       ;put relevant bit into result
        MOVS   Rcnt,Rcnt,LSR #1;shift control bit
        MOVNE  Rb,Rb,LSR #1     ;halve unless finished
        BNE    Div2
;
; divide result in Rc
; remainder in Ra
```

7.15.2 Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition), so this example uses a 33 bit register with taps at bits 33 and 20. The basic algorithm is newbit:=bit33 eor bit20, shift left the 33 bit number and put in newbit at the bottom; this operation is performed for all the newbits needed (i.e. 32 bits). The entire operation can be done in 5S cycles:

```
; enter with seed in Ra (32 bits), Rb (1 bit in Rb lsb), uses Rc
;
     TST   Rb,Rb,LSR #1     ;top bit into carry
     MOVS  Rc,Ra,RRX        ;33 bit rotate right
     ADC   Rb,Rb,Rb         ;carry into lsb of Rb
     EOR   Rc,Rc,Ra,LSL#12  ;(involved!)
     EOR   Ra,Rc,Rc,LSR#20  ;(similarly involved!)
;
; new seed in Ra, Rb as before
```

7.15.3 Multiplication by constant using the barrel shifter

(1) Multiplication by 2^n (1,2,4,8,16,32..)

```
MOV   Ra,Ra,LSL #n
```

(2) Multiplication by 2^n+1 (3,5,9,17..)

```
ADD   Ra,Ra,Ra,LSL #n
```

(3) Multiplication by 2^n-1 (3,7,15..)

```
RSB   Ra,Ra,Ra,LSL #n
```

(4) Multiplication by 6

```
ADD   Ra,Ra,Ra,LSL #1 ;multiply by 3
MOV   Ra,Ra,LSL #1    ;and then by 2
```

(5) Multiply by 10 and add in extra number

```
ADD   Ra,Ra,Ra,LSL #2 ;multiply by 5
ADD   Ra,Rc,Ra,LSL #1 ;multiply by 2 and add in next digit
```

(6) General recursive method for Rb := Ra*C where C is a constant:

```
(a) If C even, say C = 2^n*D, D odd:

    D=1:  MOV   Rb,Ra,LSL #n
    D<>1: {Rb := Ra*D}
          MOV   Rb,Rb,LSL #n

(b) If C MOD 4 = 1, say C = 2^n*D+1, D odd, n>1:

    D=1:  ADD   Rb,Ra,Ra,LSL #n
    D<>1: {Rb := Ra*D}
          ADD   Rb,Ra,Rb,LSL #n

(c) If C MOD 4 = 3, say C = 2^n*D-1, D odd, n>1:

    D=1:  RSB   Rb,Ra,Ra,LSL #n
    D<>1: {Rb := Ra*D}
          RSB   Rb,Ra,Rb,LSL #n
```

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
RSB   Rb,Ra,Ra,LSL #2 ;multiply by 3
RSB   Rb,Ra,Rb,LSL #2 ;multiply by 4*3-1 = 11
ADD   Rb,Ra,Rb,LSL #2 ;multiply by 4*11+1 = 45
```

rather than by:

```
ADD   Rb,Ra,Ra,LSL #3 ;multiply by 9
ADD   Rb,Rb,Rb,LSL #2 ;multiply by 5*9 = 45
```

7.15.4 Loading a word from an unknown alignment

```
; enter with address in Ra (32 bits)
; uses Rb, Rc; result in Rd.
; Note d must be less than c e.g. 0,1
;
     BIC   Rb,Ra,#3          ;get word aligned address
     LDMIA Rb,{Rd,Rc}        ;get 64 bits containing answer
     AND   Rb,Ra,#3          ;correction factor in bytes
     MOVS  Rb,Rb,LSL #3      ;...now in bits and test if aligned
     MOVNE Rd,Rd,LSR Rb      ;produce bottom of result word
                             ;(if not aligned)
     RSBNE Rb,Rb,#32         ;get other shift amount
     ORRNE Rd,Rd,Rc,LSL Rb ;combine two halves to get result
```

7.15.5 Loading a halfword (Little Endian)

```
     LDR   Ra, [Rb,#2]       ; Get halfword to bits 15:0
     MOV   Ra,Ra,LSL #16     ;  move to top
     MOV   Ra,Ra,LSR #16     ;  and back to bottom
                             ;use ASR to get sign extended version
```

7.15.6 Loading a halfword (Big Endian)

```
     LDR   Ra, [Rb,#2]       ; Get halfword to bits 31:16
     MOV   Ra,Ra,LSR #16     ;  and back to bottom
                             ;  use ASR to get sign extended version
```

8. Memory Interface

ARM60 reads instructions and data from, and writes data to, its memory system via a 32 bit data bus (**d[31:0]**). A separate 32 bit address bus specifies the memory location to be used for the transfer, and the **Nrw** signal gives the direction of transfer (**ARM60** to memory or memory to **ARM60**). Control signals give additional information about the transfer cycle, and in particular they facilitate the use of DRAM page mode where applicable. Interfaces to static RAM based memories are not ruled out and, in general, they are much simpler than the DRAM interface described here.

8.1 Cycle types

All memory transfer cycles can be placed in one of four categories:

(1) Non-sequential cycle. **ARM60** requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.

(2) Sequential cycle. **ARM60** requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word after the preceding address.

(3) Internal cycle. **ARM60** does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.

(4) Coprocessor register transfer. **ARM60** wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

These four classes are distinguishable to the memory system by inspection of the **Nmreq** and **seq** control lines (see Table 3). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

Nmreq	seq	Cycle type	
0	0	Non-sequential cycle	(N-cycle)
0	1	Sequential cycle	(S-cycle)
1	0	Internal cycle	(I-cycle)
1	1	Coprocessor register transfer	(C-cycle)

Table 3: Memory cycle types

Figure 29 shows the pipelining of the control signals, and suggests how the DRAM address strobes (**NRAS** and **NCAS**) might be timed to use page mode for S-cycles. Note that the N-cycle is longer than the other cycles. This is to allow for the DRAM precharge and row access time, and is not an **ARM60** requirement.

When an S-cycle follows an N-cycle, the address will always be one word greater than the address used in the N-cycle. This address (marked "a" in the above diagram) should be checked to ensure that it is not the last in the DRAM page before the memory system commits to the S-cycle. If it is at the page end, the S-cycle cannot be performed in page mode and the memory system will have to perform a full access. The processor clock must be stretched to match the full access.

When an S-cycle follows an I- or C-cycle, the address will be the same as that used in the I- or C-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to

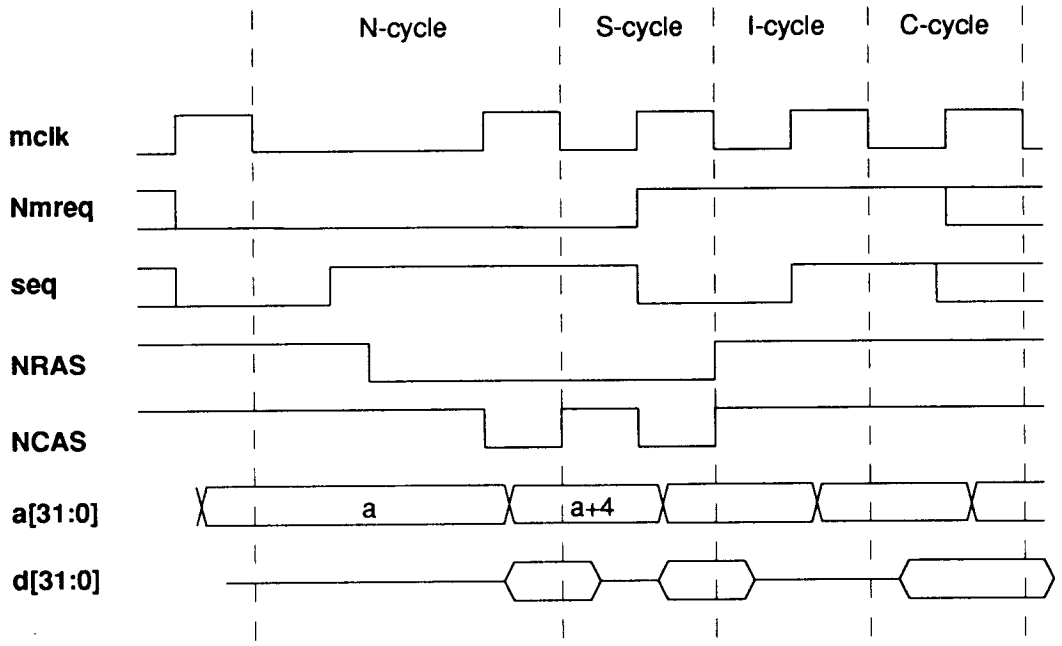


Figure 29: ARM Memory Cycle Timing

run at page mode speed whilst performing a full DRAM access. This is shown below in Figure 30.

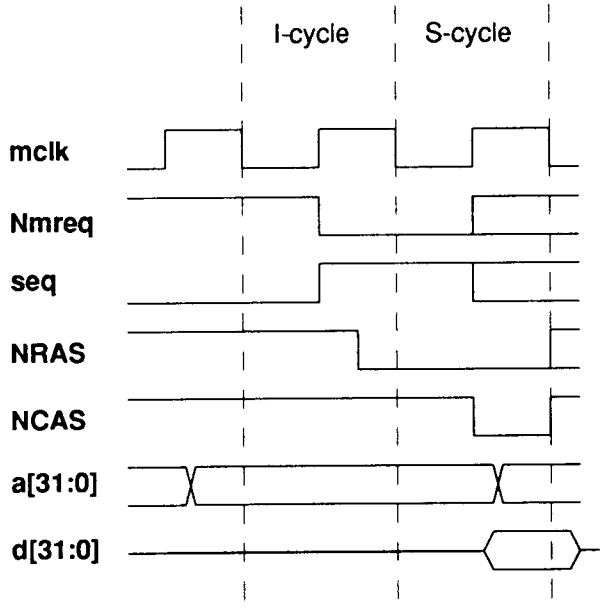


Figure 30: Memory Cycle Optimisation

8.2 Byte addressing

The processor address bus gives byte addresses, but instructions are always words (where a word is 4 bytes) and data quantities are usually words. Single data transfers (LDR and STR) can, however, specify that a byte quantity is required. The **Nbw** control line is used to request a byte from the memory system; normally it is HIGH, signifying a request for a word quantity, and it goes LOW during phase 2 of the preceding cycle to request a byte transfer.

When the processor is fetching an instruction from memory, the state of the bottom two address lines **a[1:0]** is undefined.

When a byte is requested in a read transfer (LDRB), the memory system can safely ignore that the request is for a byte quantity and present the whole word. **ARM60** will perform the byte extraction internally. Alternatively, the memory system may activate only the addressed byte of the memory. This may be desirable in order to save power, or to enable the use of a common decoding system for both read and write cycles.

If a byte write is requested (STRB), **ARM60** will broadcast the byte value across the data bus, presenting it at each byte location within the word. The memory system must decode **a[1:0]** to enable writing only to the addressed byte.

One way of implementing the byte decode in a DRAM system is to separate the 32 bit wide block of DRAM into four byte wide banks, and generate the column address strobes independently as shown in Figure 31 below.

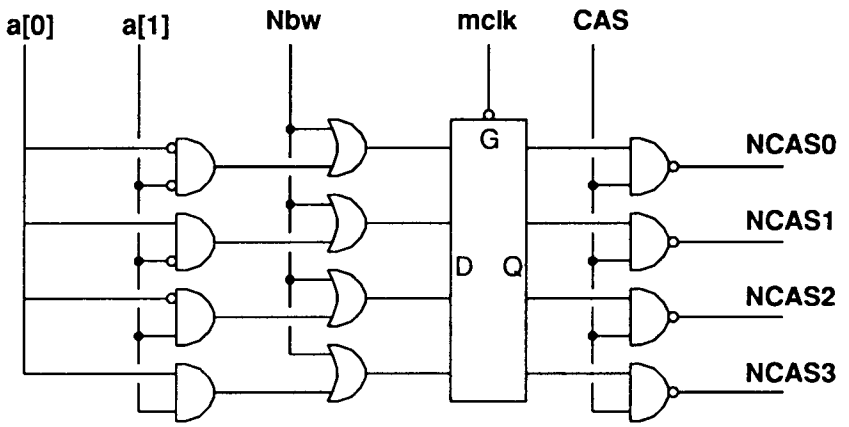


Figure 31: Decoding Byte Accesses to Memory

When the processor is configured for Little Endian operation byte 0 of the memory system should be connected to data lines 7 through 0 (**d[7:0]**) and strobed by **NCAS0**. **NCAS1** drives the bank connected to data lines 15 though 8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time critical signal.

In the Big Endian case, byte 0 of the memory system should be connected to data lines 31 through 24.

8.3 Address timing

Normally the processor address changes during phase 2 to the value which the memory system should use during the following cycle. This gives maximum time for driving the address to large memory arrays, and for address translation where required. Dynamic memories usually latch the address on chip, and if the latch is timed correctly they will work even though the address changes before the access has completed. Static RAMs and ROMs will not work under such circumstances, as they require the address to be stable

until after the access has completed. Therefore for use with such devices the address transition must be delayed until after the end of phase 2. An on-chip address latch, controlled by **ale**, allows the address timing to be modified in this way.

In a system with a mixture of static and dynamic memories (which for these purposes means a mixture of devices with and without address latches), the use of **ale** may change dynamically from one cycle to the next, at the discretion of the memory system.

8.4 Memory management

The **ARM60** address bus may be processed by an address translation unit before being presented to the memory, and **ARM60** is capable of running a virtual memory system. The **abort** input to the processor may be used by the memory manager to inform **ARM60** of page faults. Various other signals enable different page protection levels to be supported:

(i) **Nrw** can be used by the memory manager to protect pages from being written to.

(ii) **Ntrans** indicates whether the processor is in User or a non-user mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user.

(iii) **Nm[4:0]** can give the memory manager full information on the processor mode.

Address translation will normally only be necessary on an N-cycle, and this fact may be exploited to reduce power consumption in the memory manager and avoid the translation delay at other times. The times when translation is necessary can be deduced by keeping track of the cycle types that the processor uses.

If an N-cycle is matched to a full DRAM access, it will be longer than the minimum processor cycle time. Stretching phase 1 rather than phase 2 will give the translation system more time to generate an abort (which must be set up to the end of phase 1).

8.5 Locked operations

ARM60 includes a data swap (SWP) instruction that allows the contents of a memory location to be swapped with the contents of a processor register. This instruction is implemented as an uninterruptable pair of accesses; the first access reads the contents of the memory, and the second writes the register data to the memory. These accesses must be treated as a contiguous operation by the memory controller to prevent another device from changing the affected memory location before the swap is completed. **ARM60** drives the **lock** signal HIGH for the duration of the swap operation to warn the memory controller not to give the memory to another device.

8.6 Stretching access times

All memory timing is defined by **mclk**, and long access times can be accommodated by stretching this clock. It is usual to stretch the LOW period of **mclk**, as this allows the memory manager to abort the operation if the access is eventually unsuccessful (**abort** must be setup prior to the rising edge of **mclk**).

Either **mclk** can be stretched before it is applied to **ARM60**, or the **Nwait** input can be used together with a free-running **mclk**. Taking **Nwait** LOW has the same effect as stretching the LOW period of **mclk**, and **Nwait** must only change when **mclk** is LOW.

ARM60 does not contain any dynamic logic which relies upon regular clocking to maintain its internal state. Therefore there is no limit upon the maximum period for which **mclk** may be stretched, or **Nwait** held LOW.

9. Coprocessor Interface

The functionality of the **ARM60** instruction set may be extended by the addition of up to 16 external coprocessors. When the coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the coprocessor will then increase the system performance in a software compatible way. Note that some coprocessor numbers have already been assigned. Contact ARM Ltd for up to date information.

9.1 Interface signals

Three dedicated signals control the coprocessor interface, **Ncpi**, **cpa** and **cpb**. The **cpa** and **cpb** inputs should be driven high except when they are being used for handshaking.

9.1.1 Coprocessor present/absent

ARM60 takes **Ncpi** LOW whenever it starts to execute a coprocessor (or undefined) instruction. (This will not happen if the instruction fails to be executed because of the condition codes.) Each coprocessor will have a copy of the instruction, and can inspect the CP# field to see which coprocessor it is for. Every coprocessor in a system must have a unique number and if that number matches the contents of the CP# field the coprocessor should drive the **cpa** (coprocessor absent) line LOW. If no coprocessor has a number which matches the CP# field, **cpa** and **cpb** will remain HIGH, and **ARM60** will take the undefined instruction trap. Otherwise **ARM60** observes the **cpa** line going LOW, and waits until the coprocessor is not busy.

9.1.2 Busy-waiting

If **cpa** goes LOW, **ARM60** will watch the **cpb** (coprocessor busy) line. Only the coprocessor which is driving **cpa** LOW is allowed to drive **cpb** LOW, and it should do so when it is ready to complete the instruction. **ARM60** will busy-wait while **cpb** is HIGH, unless an enabled interrupt occurs, in which case it will break off from the coprocessor handshake to process the interrupt. Normally **ARM60** will return from processing the interrupt to retry the coprocessor instruction.

When **cpb** goes LOW, the instruction continues to completion. This will involve data transfers taking place between the coprocessor and either **ARM60** or memory, except in the case of coprocessor data operations which complete immediately the coprocessor ceases to be busy.

All three interface signals are sampled by both **ARM60** and the coprocessor(s) on the rising edge of **mclk**. If all three are LOW, the instruction is committed to execution, and if transfers are involved they will start on the next cycle. If **Ncpi** has gone HIGH after being LOW, and before the instruction is committed, **ARM60** has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other coprocessor instructions may come sooner, and the instruction should be discarded.

9.1.3 Pipeline following

In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. All **ARM60** instructions are fetched from memory via the main data bus, and coprocessors are connected to this bus, so they can keep copies of all instructions as they go into the **ARM60** pipeline. The **Nopc** signal indicates when an instruction fetch is taking place, and **mclk** gives the timing of the transfer, so these may be used together to load an instruction pipeline within the coprocessor.

9.2 Data transfer cycles

Once the coprocessor has gone not-busy in a data transfer instruction, it must supply or accept data at the **ARM60** bus rate (defined by **mclk**). It can deduce the direction of transfer by inspection of the L bit in the instruction, but must only drive the bus when permitted to by **dbe** being HIGH. The coprocessor is responsible for determining the number of words to be transferred; **ARM60** will continue to increment the

address by one word per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor driving **cpa** and **cpb** HIGH.

There is no limit in principle to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case **ARM60** interrupt latency, as the instruction is not interruptable once the transfers have commenced. At 16 words, this instruction is comparable with a block transfer of 16 registers, and therefore does not affect the worst case latency.

9.3 Register transfer cycle

The coprocessor register transfer cycle is the one case when **ARM60** requires the data bus without requiring the memory to be active. The memory system is informed that the bus is required by **ARM60** taking both **Nmreq** and **seq** HIGH. When the bus is free, **dbe** should be taken HIGH to allow **ARM60** or the coprocessor to drive the bus, and an **mclk** cycle times the transfer.

9.4 Privileged instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the **Ntrans** and/or **Nm[4:0]** outputs.

As an example of the use of this facility, consider the case of a floating point coprocessor (FPU) in a multi-tasking system. The operating system could save all the floating point registers on every task switch, but this is inefficient in a typical system where only one or two tasks will use floating point operations. Instead, there could be a privileged instruction which turns the FPU on or off. When a task switch happens, the operating system can turn the FPU off without saving its registers. If the new task attempts an FPU operation, the FPU will appear to be absent, causing an undefined instruction trap. The operating system will then realise that the new task requires the FPU, so it will re-enable it and save FPU registers. The task can then use the FPU as normal. If, however, the new task never attempts an FPU operation (as will be the case for most tasks), the state saving overhead will have been avoided.

9.5 Idempotency

A consequence of the implementation of the coprocessor interface, with the interruptable busy-wait state, is that all instructions may be interrupted at any point up to the time when the coprocessor goes not-busy. If so interrupted, the instruction will normally be restarted from the beginning after the interrupt has been processed. It is therefore essential that any action taken by the coprocessor before it goes not-busy must be idempotent, ie must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to an **ARM60** register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as **ARM60** will expect to receive the integer value on the cycle immediately following that where it goes not-busy. The coprocessor must therefore preserve the original floating point value and not corrupt it during the conversion, because it will be required again if an interrupt arises during the busy period.

The coprocessor data operation class of instruction is not generally subject to idempotency considerations, as the processing activity can take place after the coprocessor goes not-busy. There is no need for **ARM60** to be held up until the result is generated, because the result is confined to stay within the coprocessor.

9.6 Undefined instructions

Undefined instructions are treated by **ARM60** as coprocessor instructions. All coprocessors must be absent (ie **cpa** and **cpb** must be HIGH) when an undefined instruction is presented. **ARM60** will then take the undefined instruction trap. Note that the coprocessor need only look at bit 27 of the instruction to differentiate undefined instructions (which all have 0 in bit 27) from coprocessor instructions (which all have 1 in bit 27).

10. Instruction Cycle Operations

In the following tables **Nmreq** and **seq** (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the address of the next cycle. The address, **Nbw**, **Nrw**, and **Nopc** (which appear up to half a cycle ahead) are shown in the cycle to which they apply.

10.1 Branch and branch with link

A branch instruction calculates the branch destination in the first cycle, whilst performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination + 4, refilling the instruction pipeline, and if the branch is with link R14 is modified (4 is subtracted from it) to simplify return from SUB PC,R14,#4 to MOV PC,R14. This makes the STM ..{R14} LDM ..{PC} type of subroutine work correctly. The cycle timings are shown below in Table 4.

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
1	pc+8	1	0	(pc+8)	0	0	0
2	alu	1	0	(alu)	1	0	0
3	alu+4 alu+8	1	0	(alu+4)	1	0	0

pc is the address of the branch instruction
alu is an address calculated by **ARM60**
(alu) are the contents of that address, etc

Table 4: Branch Instruction Cycle Operations

10.2 Data operations

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus. The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are used.)

An instruction prefetch occurs at the same time as the above operation, and the program counter is incremented.

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (ie will not request memory). This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC may be one or more of the register operands. When it is the destination external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are locked out.

PSR Transfer operations exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register. The cycle timings are shown below in Table 5.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
normal	1	pc+8 pc+12	1	0	(pc+8)	1	0	0
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	1	0	0
	3	alu+4 alu+8	1	0	(alu+4)	1	0	0
shift(Rs)	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12 pc+12	1	0	-	1	0	1
shift(Rs), dest=pc	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	0	0	1
	3	alu	1	0	(alu)	1	0	0
	4	alu+4 alu+8	1	0	(alu+4)	1	0	0

Table 5: Data Operation Instruction Cycle Operations

10.3 Multiply and multiply accumulate

The multiply instructions make use of special hardware which implements a 2 bit Booth's algorithm with early termination. During the first cycle the accumulate Register is brought to the ALU, which either transmits it or produces zero (according to whether the instruction is MLA or MUL) to initialise the destination register. During the same cycle, the multiplier (Rs) is loaded into the Booth's shifter via the A bus.

The datapath then cycles, adding the multiplicand (Rm) to, subtracting it from, or just transmitting, the result register. The multiplicand is shifted in the Nth cycle by 2N or 2N+1 bits, under control of the Booth's logic. The multiplier is shifted right 2 bits per cycle, and when it is zero the instruction terminates (possibly after an additional cycle to clear a pending borrow).

All cycles except the first are internal. The cycle timings are shown below in Table 6.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
(Rs)=0,1	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12 pc+12	1	0	-	1	0	1
(Rs)>1	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	0	1	1
	.	pc+12	1	0	-	0	1	1
	m	pc+12	1	0	-	0	1	1
	m+1	pc+12 pc+12	1	0	-	1	0	1

m is the number cycles required by the Booth's algorithm; see the section on instruction speeds.

Table 6: Multiply Instruction Cycle Operations

10.4 Load register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in Table 7.

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case the destination modification is prevented. In addition, if the processor is configured for Early Abort, the base register write-back is also prevented.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
normal	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	b/w	0	(alu)	0	1	1
	3	pc+12	1	0	-	1	0	1
		pc+12						
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	b/w	0	pc'	0	1	1
	3	pc+12	1	0	-	0	0	1
	4	pc'	1	0	(pc')	1	0	0
	5	pc'+4	1	0	(pc'+4)	1	0	0
		pc'+8						

Table 7: Load Register Instruction Cycle Operations

10.5 Store register

The first cycle of a store register is similar to the first cycle of load register. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle. The cycle timings are shown below in Table 8. The base write-back is prevented during a Data Abort if the processor is configured for Early Abort. The write-back is not prevented if Late Abort is configured.

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ntrans
1	pc+8	1	0	(pc+8)	0	0	0	
2	alu	b/w	1	Rd	0	0	1	
	pc+12							

Table 8: Store Register Instruction Cycle Operations

10.6 Load multiple registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, whilst performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is moved to the ALU A bus input latch for holding in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown below in Table 9.

The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. The final cycle is altered to restore the modified base register (which may have been overwritten

by the load activity before the abort occurred).

When the PC is in the list of registers to be loaded, and assuming that no abort takes place, the current instruction pipeline must be invalidated.

Note that the PC is always the last register to be loaded, so an abort at any point will prevent the PC from being overwritten.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	0	1	1
	3	pc+12	1	0	-	1	0	1
		pc+12						
1 register	1	pc+8	1	0	(pc+8)	0	0	0
dest=pc	2	alu	1	0	pc'	0	1	1
	3	pc+12	1	0	-	0	0	1
	4	pc'	1	0	(pc')	1	0	0
	5	pc'+4	1	0	(pc'+4)	1	0	0
		pc'+8						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	0	(alu)	1	0	1
	.	alu+.	1	0	(alu+.)	1	0	1
	n	alu+.	1	0	(alu+.)	1	0	1
	n+1	alu+.	1	0	(alu+.)	0	1	1
	n+2	pc+12	1	0	-	1	0	1
		pc+12						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	0	(alu)	1	0	1
incl. pc	.	alu+.	1	0	(alu+.)	1	0	1
	n	alu+.	1	0	(alu+.)	1	0	1
	n+1	alu+.	1	0	pc'	0	1	1
	n+2	pc+12	1	0	-	0	0	1
	n+3	pc'	1	0	(pc')	1	0	0
	n+4	pc'+4	1	0	(pc'+4)	1	0	0
		pc'+8						

Table 9: Load Multiple Registers Instruction Cycle Operations

10.7 Store multiple registers

Store multiple proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers to contend with. The cycle timings are shown below in Table 10.

10.8 Data swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in Table 11.

The **lock** output of **ARM60** is driven HIGH for the duration of the swap operation (cycles 2 & 3) to indicate that both cycles should be allowed to complete without interruption.

The data swapped may be a byte or word quantity (b/w).

The swap operation may be aborted in either the read or write cycle, and in both cases the destination register will not be affected.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	1	Ra	0	0	1
		pc+12						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	1	Ra	1	0	1
	.	alu+.	1	1	R.	1	0	1
	n	alu+.	1	1	R.	1	0	1
	n+1	alu+.	1	1	R.	0	0	1
		pc+12						

Table 10: Store Multiple Registers Instruction Cycle Operations

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	lock
1	pc+8	1	0	(pc+8)	0	0	0	0
2	Rn	b/w	0	(Rn)	0	0	1	1
3	Rn	b/w	1	Rm	0	1	1	1
4	pc+12	1	0	-	1	0	1	0
	pc+12							

Table 11: Data Swap Instruction Cycle Operations

10.9 Software interrupt and exception entry

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle the forced address is constructed, and a mode change may take place. The return address is moved to R14 and the CPSR to SPSR_svc.

During the second cycle the return address is modified to facilitate return, though this modification is less useful than in the case of branch with link.

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in Table 12.

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ntrans
1	pc+8	1	0	(pc+8)	0	0	0	1
2	Xn	1	0	(Xn)	1	0	0	1
3	Xn+4	1	0	(Xn+4)	1	0	0	1
	Xn+8							

For software interrupt, *pc* is the address of the SWI instruction. For interrupts and reset, *pc* is the address of the instruction following the last one to be executed before entering the exception. For prefetch abort, *pc* is the address of the aborting instruction. For data abort, *pc* is the address of the instruction following the one which attempted the aborted data transfer. *Xn* is the appropriate trap address.

Table 12: Software Interrupt Instruction Cycle Operations

10.10 Coprocessor data operation

A coprocessor data operation is a request from **ARM60** for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before driving **cpb** LOW.

If the coprocessor can never do the requested task, it should leave **cpa** and **cpb** HIGH. If it can do the task, but can't commit right now, it should drive **cpa** LOW but leave **cpb** HIGH until it can commit. **ARM60** will busy-wait until **cpb** goes LOW. The cycle timings are shown below in Table 13.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
ready	1	pc+8 pc+12	1	0	(pc+8)	0	0	0	0	0	0
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
		pc+12									

Table 13: Coprocessor Data Operation Instruction Cycle Operations

10.11 Coprocessor data transfer (from memory to coprocessor)

Here the coprocessor should commit to the transfer only when it is ready to accept the data. When **cpb** goes LOW, **ARM60** will produce addresses and expect the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by driving **cpa** and **cpb** HIGH.

ARM60 spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address base during the transfer cycles. The cycle timings are shown below in Table 14.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
1 register ready	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
	2	alu	1	0	(alu)	0	0	1	1	1	1
		pc+12									
1 register not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	0	(alu)	0	0	1	1	1	1
		pc+12									
n registers (n>1) ready	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
	2	alu	1	0	(alu)	1	0	1	1	0	0
	.	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+1	alu+.	1	0	(alu+.)	0	0	1	1	1	1
		pc+12									
m registers (m>1) not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	0	(alu)	1	0	1	1	0	0
ready	.	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+m	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+m+1	alu+.	1	0	(alu+.)	0	0	1	1	1	1
		pc+12									

Table 14: Coprocessor Data Transfer Instruction Cycle Operations

10.12 Coprocessor data transfer (from coprocessor to memory)

The **ARM60** controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the Nrw line is inverted during the transfer cycle. The cycle timings are shown below in Table 15.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
1 register	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
not ready	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	1	1	CPdata	1	0	1	1	0	0
ready	.	alu+.	1	1	CPdata	1	0	1	1	0	0
	n	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+1	alu+.	1	1	CPdata	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
(m>1)	2	pc+8	1	0	-	0	1	1	0	0	1
not ready	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	1	0	1	1	0	0
ready	.	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+m	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+m+1	alu+.	1	1	CPdata	0	0	1	1	1	1
		pc+12									

Table 15: Coprocessor Data Transfer Instruction Cycle Operations

10.13 Coprocessor register transfer (Load from coprocessor)

Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and **ARM60** puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all **ARM60** register load instructions. The cycle timings are shown below in Table 16.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	0	CPdata	0	1	1	1	1	1
	3	pc+12	1	0	-	1	0	1	1	-	-
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	0	CPdata	0	1	1	1	1	1
	n+2	pc+12	1	0	-	1	0	1	1	-	-
		pc+12									

Table 16: Coprocessor Register Transfer Instruction Cycle Operations

10.14 Coprocessor register transfer (Store to coprocessor)

As for the load from coprocessor, except that the last cycle is omitted. The cycle timings are shown below in Table 17.

	Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	1	Rd	0	0	1	1	1	1
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	1	Rd	0	0	1	1	1	1
		pc+12									

Table 17: Coprocessor Register Transfer Instruction Cycle Operations

10.15 Undefined instructions and coprocessor absent

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive **cpa** or **cpb** LOW. These will remain HIGH, causing the undefined instruction trap to be taken. The cycle timings are shown below in Table 18.

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc	Ncpi	cpa	cpb
1	pc+8	1	0	(pc+8)	0	1	0	0	1	1
2	pc+8	1	0	-	0	0	0	1	1	1
3	Xn	1	0	(Xn)	1	0	0	1	1	1
4	Xn+4	1	0	(Xn+4)	1	0	0	1	1	1
	Xn+8									

Table 18: Undefined Instruction Cycle Operations

10.16 Unexecuted instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see Table 19).

Cycle	address	Nbw	Nrw	data	seq	Nmreq	Nopc
1	pc+8	1	0	(pc+8)	1	0	0
	pc+12						

Table 19: Undefined Instruction Cycle Operations

10.17 Instruction speeds

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in Table 20. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

Data Processing	1 S	+ 1 S for SHIFT(Rs) + 1 S + 1 N if R15 written
MSR,MRS	1 S	
LDR	1 S + 1 N + 1 I	+ 1 S + 1 N if R15 loaded
STR	2 N	
LDM	n S + 1 N + 1 I	+ 1 S + 1 N if R15 loaded
STM	(n-1)S + 2 N	
SWP	1 S + 2 N + 1 I	
B,BL	2 S + 1 N	
SWI,trap	2 S + 1 N	
MUL,MLA	1 S + m I	
CDP	1 S + b I	
LDC,STC	(n-1)S + 2 N + b I	
MRC	1 S + b I + 1 C	
MCR	1 S + (b+1)I + 1 C	

n is the number of words transferred.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between 2^(2*m*-3) and 2^(2*m*-1)-1 takes 1S+*m*I cycles for 1<*m*<16. Multiplication by 0 or 1 takes 1S+1I cycles, and multiplication by any number greater than or equal to 2^(29) takes 1S+16I cycles. The maximum time taken by any multiply is therefore 1S+16I cycles.

b is the number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all instructions take one S cycle.

The cycle types (N, S, I and C) are defined in the memory interface chapter.

Table 20: ARM Instruction Speeds

11. DC Parameters

11.1 Absolute Maximum Ratings

Symbol	Parameter	Min	Typ	Max	Units	Note
VDD	Supply voltage	0.0		7.0	V	
Vip	Voltage applied to input pin	-0.5		7.0	V	
Vop	Voltage applied to output pin	-0.5		VDD+ 0.3	V	
Osct	Output short circuit time			1	S	1
Ts	Storage temperature	-65		150	deg.C	
Ta	Ambient operating temperature	-10		80	deg.C	
Pd	Maximum power dissipation			2.0	W	

NOTES

These are stress ratings only. Exceeding the absolute maximum ratings may permanantly damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability. Functional operation of the device at these or any other condition outside those specified is not implied.

The device contains circuitry designed to provide protection from damage by static discharge. It is nonetheless recommended that precautions be taken to avoid applying voltages outside the specified range.

All voltages are measured with respect to VSS.

(1) Not more than one output should be shorted to VDD or VSS at any one time

11.2 Recommended DC Operating Conditions

Symbol	Parameter	Min	Typ	Max	Units	Note
VDD	Supply voltage	4.75	5.0	5.25	V	
Vih	Input HIGH voltage	2.4		VDD	V	1
Vil	Input LOW voltage	0.0		0.8	V	1
Io4	Output current (O4 outputs)			+/-4	mA	
Io8	Output current (OS8 outputs)			+/-8	mA	
Ta	Ambient operating temperature	0		70	deg.C	

NOTES

All voltages are measured with respect to VSS.

(1) All inputs are TTL compatible

11.3 DC Characteristics

Given **VDD** = 5.0V ± 5%, Ta = 0 to 70 °C.

Symbol	Parameter	Min	Typ	Max	Units	Note
Idd	Supply current			50	mA	1
Ilatch	D.C. latch-up current	TBD			mA	1,2
Iin	'I' input leakage current			+/-10	uA	3
Vol	Output LOW voltage			0.4	V	4
Voh	Output HIGH voltage	2.4			V	4
Rp	'IP' input pullup resistor	35k		100k	ohm	5
Cin	Input capacitance		TBD		pF	1,6

NOTES

All voltages are measured with respect to **VSS**.

(1) To be determined

(2) This value represents the current that the input/output pins can tolerate before the chip latches up. As sustained latch-up is catastrophic, this current must never be approached.

(3) For Vin = 0 to **VDD** and only for inputs without pullup resistors.

(4) When sourcing or sinking the maximum rated output current for the output driver (4 or 8mA).

(5) Only certain inputs have pullup resistors.

(6) This value includes the capacitance of the chip carrier and socket.

12. AC Parameters

Important Note - Provisional Figures

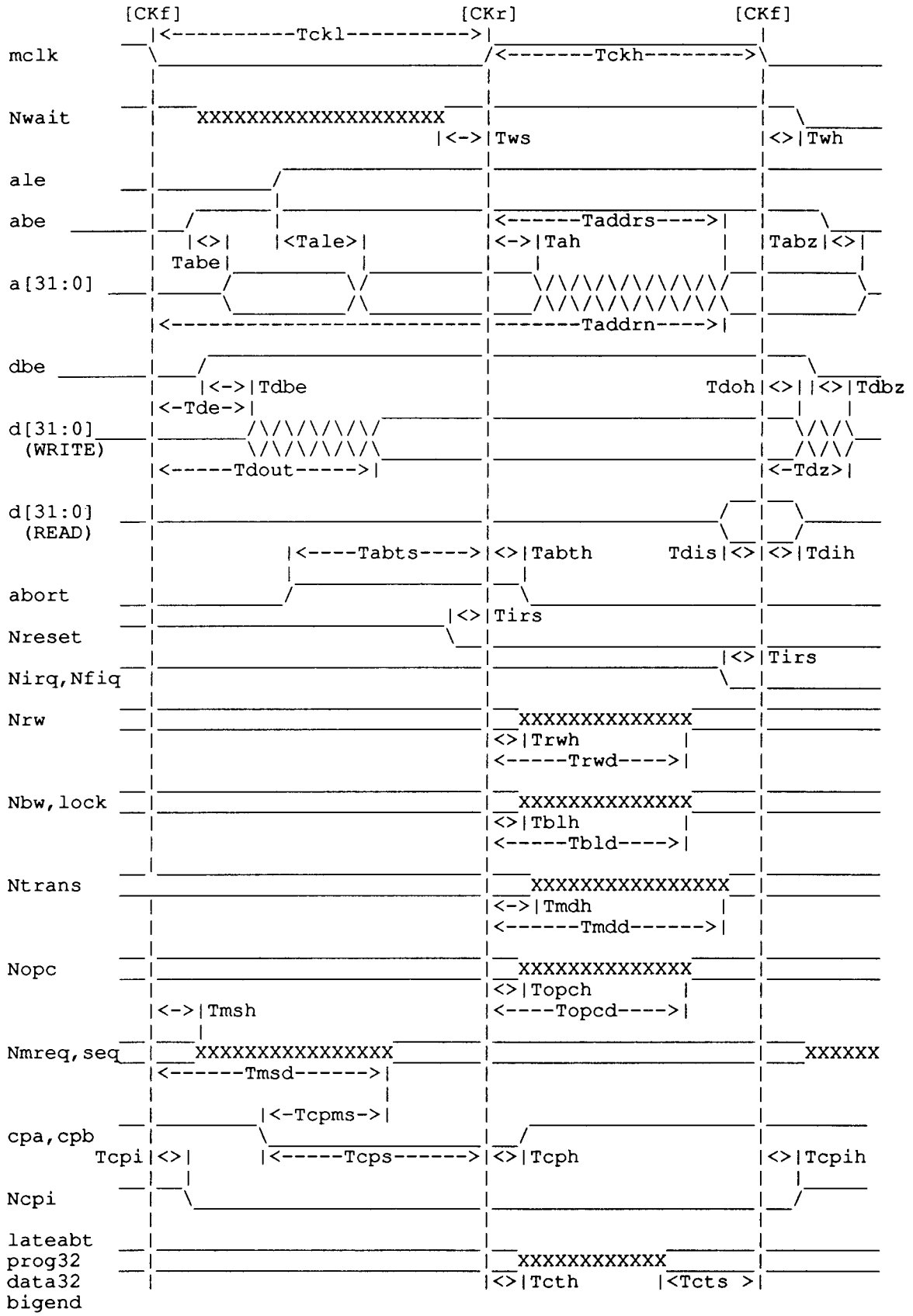
The timing parameters given here are preliminary data and subject to change when device characterisation is complete.

The AC timing diagrams presented in this section assume that the outputs of **ARM60** have been loaded with the capacitive loads shown in the 'Test Load' column of Table 21. These loads have been chosen as typical of the type of system in which **ARM60** might be employed.

The output drivers of **ARM60** exhibit a propagation delay that increases linearly with the increase in load capacitance. An 'Output derating' figure is given for each output driver, showing the approximate rate of increase of output time with increasing load capacitance.

Output Signal	Test Load (pF)	Output derating (ns/pF)
d[31:0]	50	0.072
a[31:0]	50	0.072
lock	25	0.072
Ncpi	25	0.093
Nmreq	25	0.093
seq	25	0.093
Nrw	25	0.072
Nbw	25	0.072
Nopc	25	0.093
Ntrans	25	0.072
tdo	25	0.072

Table 21: AC test loads


[CKf]
[CKr]
[CKf]
mclk
Tckl
Tckh
Nwait
Tws
Twh
ale
abe
Taddrs
Tah
Tabz
Tabe
Tale
a[31:0]
Taddrn
dbe
Tdbe
Tde
Tdoh
Tdbz
d[31:0] (WRITE)
Tdout
Tdz
d[31:0] (READ)
Tabts
Tabth
Tdis
Tdih
abort
Nreset
Tirs
Nirq,Nfiq
Nrw
Trwh
Trwd
Nbw,lock
Tblh
Tbld
Ntrans
Tmdh
Tmdd
Nopc
Topch
Topcd
Tmsh
Nmreq,seq
Tmsd
XXXXXX
Tcpms
cpa,cpb
Tcpi
Tcps
Tcph
Tcpih
Ncpi
lateabt
prog32
data32
bigend
Tcth
Tcts

12.1 Provisional AC Parameters

Symbol	Parameter	Min	Max	Units	Note
Tckl	clock LOW time	25		ns	
Tckh	clock HIGH time	25		ns	
Tws	Nwait setup to CKr	3		ns	
Twh	Nwait hold from CKf	3		ns	
Tale	address latch open		9	ns	
Tabe	address bus enable		16	ns	
Tabz	address bus disable		12	ns	
Taddrn	CKf to address valid		44	ns	1
Taddrs	CKr to address valid		18	ns	
Tah	address hold time	5		ns	
Tdbe	data bus enable		19	ns	
Tdbz	data bus disable		16	ns	
Tde	CKf to data enable	6		ns	
Tdz	CKf to data disable		19	ns	
Tdout	data out delay		23	ns	
Tdoh	data out hold	5		ns	
Tdis	data in setup	TBD		ns	3
Tdih	data in hold	TBD		ns	3
Tabts	abort setup time	10		ns	
Tabth	abort hold time	TBD		ns	3
Tirs	interrupt setup	6		ns	2
Trwd	CKr to Nrw valid		25	ns	
Trwh	Nrw hold time	5		ns	
Tmsd	CKf to Nmreq & seq		25	ns	
Tmsh	Nmreq& seq hold time	5		ns	
Tbld	CKr to Nbw & lock		24	ns	
Tblh	Nbw & lock hold	5		ns	
Tmdd	CKr to Ntrans		25	ns	
Tmdh	Ntrans hold	5		ns	
Topcd	CKr to Nopc valid		16	ns	
Topch	Nopc hold time	5		ns	
Tcps	cpa, cpb setup	7		ns	
Tcph	cpa, cpb hold time	TBD		ns	3
Tcpms	cpa, cpb to Nmreq, seq		TBD	ns	3
Tcpi	CKf to Ncpi delay		16	ns	
Tcpih	Ncpi hold time	5		ns	
Tcts	Config setup time	9		ns	
Tcth	Config hold time	TBD		ns	3

NOTES

All figures are provisional, and assume that 1μ CMOS technology is used to fabricate

(1) Taddrn only applies to N-cycles. For S-cycles use only Taddrs.

(2) Tirs guarantees recognition of the interrupt (or reset) source by the corresponding clock edge. The interrupt and reset inputs may be applied fully asynchronously where the exact cycle of recognition is unimportant.

(3) To be determined

13. Boundary Scan

The boundary-scan interface conforms to the IEEE Std. 1149.1 - 1990, Standard Test Access Port and Boundary-Scan Architecture (please refer to this document for an explanation of the terms used in this chapter and for a description of the TAP controller states). It supports the following public instructions:

BYPASS
SAMPLE/PRELOAD
EXTEST
INTEST
IDCODE
HIGHZ
CLAMP
CLAMPZ

The boundary-scan interface consists of 5 pins on the chip. There are four inputs (**Ntrst, tms, tdi** and **tck**) and one output (**tdo**). In all the descriptions that follow, **tdi** and **tms** are sampled on the rising edge of **tck** and all output transitions on **tdo** occur following the falling edge of **tck**. Figure 32 shows the state transitions that occur in the TAP controller.

13.1 Instruction Register

The instruction register is 4 bits in length.

The fixed value loaded into the instruction register during the *CAPTURE-IR* controller state is **0001**.

13.2 Public Instructions

The following public instructions are supported:

INSTRUCTION	BINARY CODE
BYPASS	1111
SAMPLE/PRELOAD	0011
EXTEST	0000
INTEST	1100
IDCODE	1110
HIGHZ	0111
CLAMP	0101
CLAMPZ	1001

When loading a new instruction, the binary code should be shifted into **tdi** in the order least-significant to most-significant bit.

(1) BYPASS (1111)

The BYPASS instruction connects a 1 bit shift register (the BYPASS register) between **tdi** and **tdo**.

When the BYPASS instruction is loaded into the instruction register, all the boundary-scan cells are placed in their normal (system) mode of operation. This instruction has no effect on the system pins.

In the *CAPTURE-DR* state, a logic '0' is captured by the bypass register. In the *SHIFT-DR* state, test data is shifted into the bypass register via **tdi** and out via **tdo** after a delay of one **tck** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the *UPDATE-DR* state.

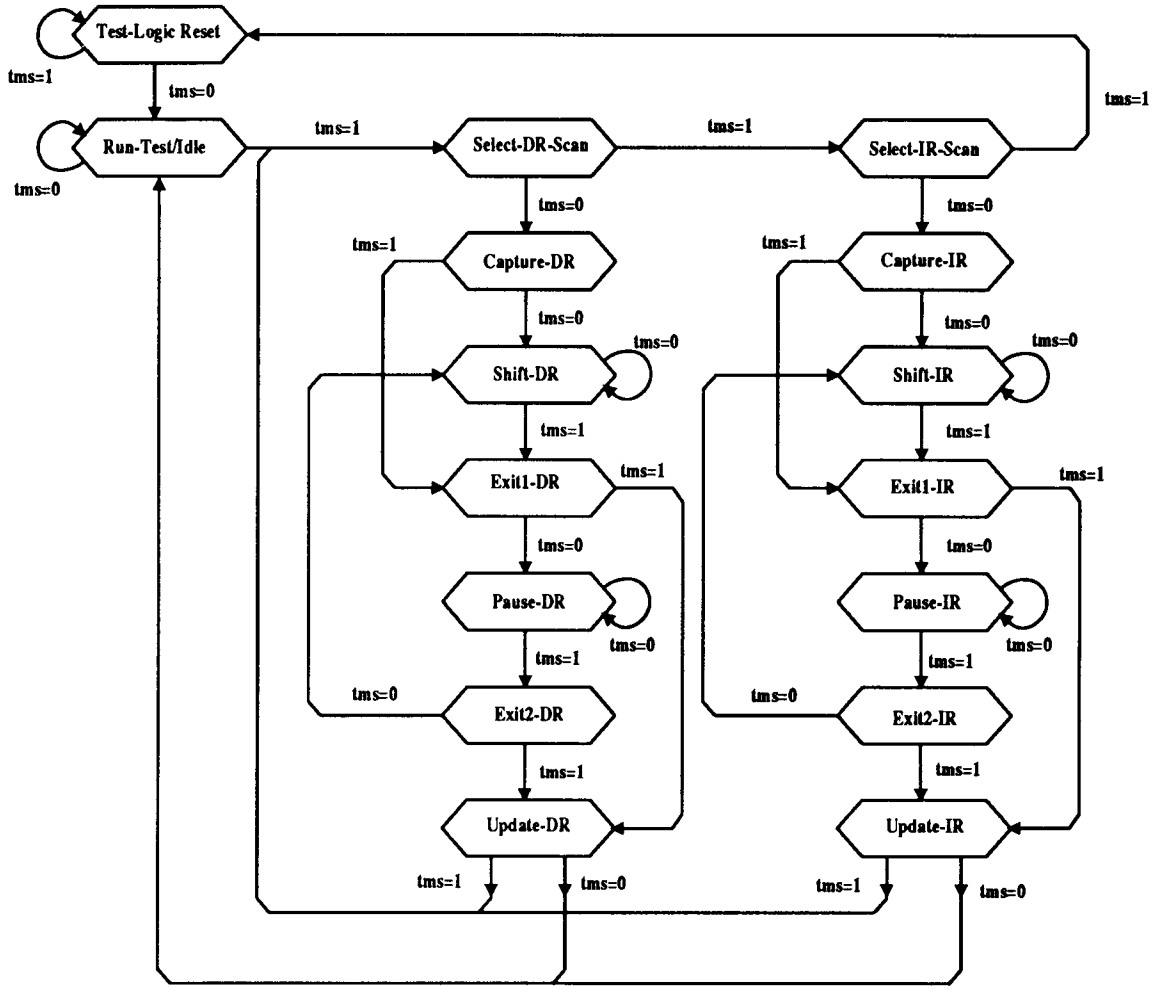


Figure 32: TAP controller state transitions

(2) SAMPLE/PRELOAD (0011)

The BS (boundary-scan) register is placed in test mode by the SAMPLE/PRELOAD instruction.

The SAMPLE/PRELOAD instruction connects the BS register between **tdi** and **tdo**.

When the instruction register is loaded with the SAMPLE/PRELOAD instruction, all the boundary-scan cells are placed in their normal system mode of operation.

In the *CAPTURE-DR* state, a snapshot of the signals at the boundary-scan cells is taken on the rising edge of **tck**. Normal system operation is unaffected. In the *SHIFT-DR* state, the sampled test data is shifted out of the BS register via the **tdo** pin, whilst new data is shifted in via the **tdi** pin to preload the BS register parallel input latch. In the *UPDATE-DR* state, the preloaded data is transferred into the BS register parallel output latch. Note that this data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active. This instruction should be used to preload the boundary-scan register with known data prior to selecting the INTEST, EXTEST, CLAMP or

CLAMPZ instructions; appropriate guard values to be used for each boundary-scan cell are documented in the section entitled *Boundary-Scan (BS) Register.*

(3) EXTEST (0000)

The BS (boundary-scan) register is placed in test mode by the EXTEST instruction.

The EXTEST instruction connects the BS register between **tdi** and **tdo.**

When the instruction register is loaded with the EXTEST instruction, all the boundary-scan cells are placed in their test mode of operation.

In the *CAPTURE-DR* state, inputs from the system pins and outputs from the boundary-scan output cells to the system pins are captured by the boundary-scan cells.

In the *SHIFT-DR* state, the previously captured test data is shifted out of the BS register via the **tdo** pin, whilst new test data is shifted in via the **tdi** pin to the BS register parallel input latch. In the *UPDATE-DR* state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins.

To ensure that the core logic receives a known, stable set of inputs during EXTEST, a set of guarding values must be shifted into some of the boundary-scan cells; this guarding pattern is specified in the section entitled *Boundary-Scan (BS) Register.* To ensure that the guarding pattern is in place from the start of the EXTEST operation, it should be shifted into the BS register using the SAMPLE/PRELOAD instruction prior to selecting EXTEST.

(4) INTEST (1100)

The BS (boundary-scan) register is placed in test mode by the INTEST instruction.

The INTEST instruction connects the BS register between **tdi** and **tdo.**

When the instruction register is loaded with the INTEST instruction, all the boundary-scan cells are placed in their test mode of operation.

In the *CAPTURE-DR* state, the inverse of the data supplied to the core logic from input boundary-scan cells is captured, while the true value of the data that is output from the core logic to output boundary-scan cells is captured.

In the *SHIFT-DR* state, the previously captured test data is shifted out of the BS register via the **tdo** pin, whilst new test data is shifted in via the **tdi** pin to the BS register parallel input latch. In the *UPDATE-DR* state, the new test data is transferred into the BS register parallel output latch. Note that this data is applied immediately to the system logic and system pins. The first INTEST vector should be clocked into the boundary-scan register, using the SAMPLE/PRELOAD instruction, prior to selecting INTEST to ensure that known data is applied to the system logic.

To ensure that the output pads are placed in a known, stable state during INTEST, a set of guarding values must be shifted into some of the boundary-scan cells; this guarding pattern is specified in the section entitled *Boundary-Scan (BS) Register.* To ensure that the guarding pattern is in place from the start of the INTEST operation, it should be shifted into the BS register using the SAMPLE/PRELOAD instruction prior to selecting INTEST.

Single-step operation is possible using the INTEST instruction.

(5) IDCODE (1110)

The IDCODE instruction connects the device identification register (or ID register) between **tdi** and **tdo.** The ID register is a 32-bit register that allows the manufacturer, part number and version of a component to be determined through the TAP.

When the instruction register is loaded with the IDCODE instruction, all the boundary-scan cells are placed in their normal (system) mode of operation.

In the *CAPTURE-DR* state, the device identification code (specified in the section entitled *Device Identification (ID) Code Register*) is captured by the ID register. In the *SHIFT-DR* state, the

previously captured device identification code is shifted out of the ID register via the **tdo** pin, whilst data is shifted in via the **tdi** pin into the ID register. In the *UPDATE-DR* state, the ID register is unaffected.

(6) HIGHZ (0111)

The HIGHZ instruction connects a 1 bit shift register (the BYPASS register) between **tdi** and **tdo**.

When the HIGHZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state.

In the *CAPTURE-DR* state, a logic '0' is captured by the bypass register. In the *SHIFT-DR* state, test data is shifted into the bypass register via **tdi** and out via **tdo** after a delay of one **tck** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the *UPDATE-DR* state.

(7) CLAMP (0101)

The CLAMP instruction connects a 1 bit shift register (the BYPASS register) between **tdi** and **tdo**.

When the CLAMP instruction is loaded into the instruction register, the state of all output signals is defined by the values previously loaded into the boundary-scan register. A guarding pattern (specified in the section entitled *Boundary-Scan (BS) Register*) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMP instruction.

In the *CAPTURE-DR* state, a logic '0' is captured by the bypass register. In the *SHIFT-DR* state, test data is shifted into the bypass register via **tdi** and out via **tdo** after a delay of one **tck** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the *UPDATE-DR* state.

(8) CLAMPZ (1001)

The CLAMPZ instruction connects a 1 bit shift register (the BYPASS register) between **tdi** and **tdo**.

When the CLAMPZ instruction is loaded into the instruction register, all outputs are placed in an inactive drive state, but the data supplied to the disabled output drivers is defined by the values previously loaded into the boundary-scan register. The purpose of this instruction is to ensure, during production testing, that each output driver can be disabled when its data input is either a '0' or a '1'. A guarding pattern (specified in the section entitled *Boundary-Scan (BS) Register*) should be pre-loaded into the boundary-scan register using the SAMPLE/PRELOAD instruction prior to selecting the CLAMPZ instruction.

In the *CAPTURE-DR* state, a logic '0' is captured by the bypass register. In the *SHIFT-DR* state, test data is shifted into the bypass register via **tdi** and out via **tdo** after a delay of one **tck** cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the *UPDATE-DR* state.

13.3 Test Data Registers

Figure 33 is a block diagram of the boundary-scan logic.

13.3.1 Bypass Register

Purpose: This is a single bit register which can be selected as the path between **tdi** and **tdo** to allow the device to be bypassed during boundary-scan testing.

Length: 1 bit

Operating Mode: When the Bypass instruction is the current instruction in the instruction register, serial data is transferred from **tdi** to **tdo** in the *SHIFT-DR* state with a delay of one **tck** cycle.

There is no parallel output from the bypass register.

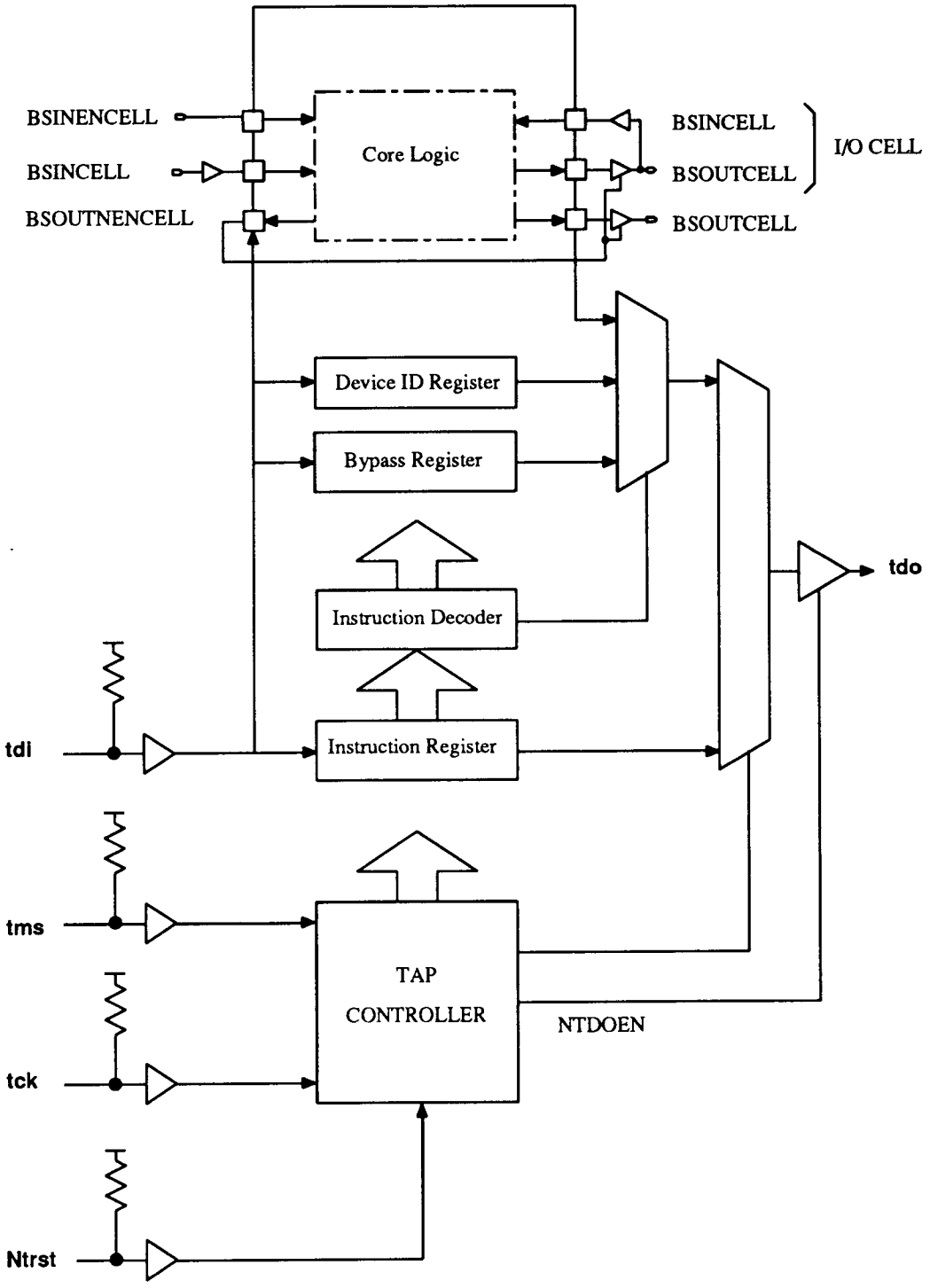


Figure 33: Boundary-Scan Block Diagram

A logic '0' is loaded from the parallel input of the bypass register in the *CAPTURE-DR* state.

13.3.2 Device Identification (ID) Code Register

Purpose: This register is used to read the 32-bit device identification code.

Length: 32 bits

Operating Mode: When the IDCODE instruction is current, the ID register is selected as the serial path between **tdi** and **tdo**.

There is no parallel output from the ID register.

The following 32-bit device identification code is loaded into the ID register during the *CAPTURE-DR* state:

Bits[31:28] : Version code = 0000
Bits[27:12] : Part number code = 0000000001000010
Bits[11:1] : Manufacturer's code = 00000101011
Bit[0] : Start bit = 1

Hexadecimal value of ID code = 00042057H

13.3.3 Boundary-Scan (BS) Register

Purpose: The BS register consists of a serially connected set of cells around the periphery of the device, at the interface between the system (or *core*) logic and the system input/output pads. This register can be used to isolate the system logic from the pins and then apply tests to the system logic, or conversely to isolate the pins from the system logic and then drive or monitor the system pins.

Length: 121 bits

Operating Modes: The BS register is selected as the register to be connected between **tdi** and **tdo** only during the SAMPLE/PRELOAD, EXTEST and INTEST instructions. Values in the BS register are used, but are not changed, during the CLAMP and CLAMPZ instructions.

In the normal (system) mode of operation, straight-through connections between the system logic and pins are maintained and normal system operation is unaffected.

In EXTEST or INTEST, values can be applied to the system logic or output pins independently of the actual values on the input pins and system logic outputs respectively. Additional boundary-scan cells are interposed in the scan chain in order to control the enabling of three-state outputs.

The correspondence between boundary-scan cells and system pins, system direction controls and system output enables is shown below. The cells are listed in the order in which they are connected in the boundary-scan register, starting with the cell closest to **tdi**. All outputs are tri-state outputs. All boundary-scan register cells at input pins can apply tests to the on-chip system logic.

EXTEST/CLAMP guard values specified in the table below should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the EXTEST, CLAMP or CLAMPZ instructions are selected to ensure that known data is applied to the system logic during the test. The INTEST guard values shown in the table below should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the INTEST instruction is selected to ensure that all outputs are disabled. An asterisk in the guard value columns indicates that any value can be substituted (as the test requires), but ones and zeros should always be placed as shown.

The values stored in the BS register after power-up are not defined. Similarly, the values previously clocked into the BS register are not guaranteed to be maintained across a Boundary-Scan reset operation (induced either by forcing **Ntrst** low or entering the *Test-Logic Reset* state).

No.	Cell name	Pin	Type	Output enable BS cell	INTEST guard value	EXTEST/CLAMP guard value
(FROM tdi)						
1	DATA[0]	d[0]	IN	-	*	0
2	DOUT[0]	d[0]	OUT	NENOUT=0	0	*
3	DATA[1]	d[1]	IN	-	*	0
4	DOUT[1]	d[1]	OUT	NENOUT=0	0	*
5	DATA[2]	d[2]	IN	-	*	0
6	DOUT[2]	d[2]	OUT	NENOUT=0	0	*
7	DATA[3]	d[3]	IN	-	*	0
8	DOUT[3]	d[3]	OUT	NENOUT=0	0	*
9	DATA[4]	d[4]	IN	-	*	0
10	DOUT[4]	d[4]	OUT	NENOUT=0	0	*
11	DATA[5]	d[5]	IN	-	*	0
12	DOUT[5]	d[5]	OUT	NENOUT=0	0	*
13	DATA[6]	d[6]	IN	-	*	0
14	DOUT[6]	d[6]	OUT	NENOUT=0	0	*
15	DATA[7]	d[7]	IN	-	*	0
16	DOUT[7]	d[7]	OUT	NENOUT=0	0	*
17	DATA[8]	d[8]	IN	-	*	0
18	DOUT[8]	d[8]	OUT	NENOUT=0	0	*
19	DATA[9]	d[9]	IN	-	*	0
20	DOUT[9]	d[9]	OUT	NENOUT=0	0	*
21	DATA[10]	d[10]	IN	-	*	0
22	DOUT[10]	d[10]	OUT	NENOUT=0	0	*
23	DATA[11]	d[11]	IN	-	*	0
24	DOUT[11]	d[11]	OUT	NENOUT=0	0	*
25	DATA[12]	d[12]	IN	-	*	0
26	DOUT[12]	d[12]	OUT	NENOUT=0	0	*
27	DATA[13]	d[13]	IN	-	*	0
28	DOUT[13]	d[13]	OUT	NENOUT=0	0	*
29	DATA[14]	d[14]	IN	-	*	0
30	DOUT[14]	d[14]	OUT	NENOUT=0	0	*
31	DATA[15]	d[15]	IN	-	*	0
32	DOUT[15]	d[15]	OUT	NENOUT=0	0	*
33	DATA[16]	d[16]	IN	-	*	0
34	DOUT[16]	d[16]	OUT	NENOUT=0	0	*
35	DATA[17]	d[17]	IN	-	*	0
36	DOUT[17]	d[17]	OUT	NENOUT=0	0	*
37	DATA[18]	d[18]	IN	-	*	0
38	DOUT[18]	d[18]	OUT	NENOUT=0	0	*
39	DATA[19]	d[19]	IN	-	*	0
40	DOUT[19]	d[19]	OUT	NENOUT=0	0	*
41	DATA[20]	d[20]	IN	-	*	0
42	DOUT[20]	d[20]	OUT	NENOUT=0	0	*
43	DATA[21]	d[21]	IN	-	*	0
44	DOUT[21]	d[21]	OUT	NENOUT=0	0	*
45	DATA[22]	d[22]	IN	-	*	0
46	DOUT[22]	d[22]	OUT	NENOUT=0	0	*
47	DATA[23]	d[23]	IN	-	*	0
48	DOUT[23]	d[23]	OUT	NENOUT=0	0	*
49	DATA[24]	d[24]	IN	-	*	0
50	DOUT[24]	d[24]	OUT	NENOUT=0	0	*
51	DATA[25]	d[25]	IN	-	*	0
52	DOUT[25]	d[25]	OUT	NENOUT=0	0	*
53	DATA[26]	d[26]	IN	-	*	0
54	DOUT[26]	d[26]	OUT	NENOUT=0	0	*
55	DATA[27]	d[27]	IN	-	*	0
56	DOUT[27]	d[27]	OUT	NENOUT=0	0	*
57	DATA[28]	d[28]	IN	-	*	0
58	DOUT[28]	d[28]	OUT	NENOUT=0	0	*
59	DATA[29]	d[29]	IN	-	*	0
60	DOUT[29]	d[29]	OUT	NENOUT=0	0	*

61	DATA[30]	d[30]	IN	-	*	0
62	DOUT[30]	d[30]	OUT	NENOUT=0	0	*
63	DATA[31]	d[31]	IN	-	*	0
64	DOUT[31]	d[31]	OUT	NENOUT=0	0	*
65	CPA	cpa	IN	-	*	1
66	NENOUT	-	OUTEN0	-	1	*
67	NCE	-	OUTEN0	-	1	*
68	LOCK	lock	OUT	NCE=0	0	*
69	BIGEND	bigend	IN	-	*	0
70	NCPI	Ncpi	OUT	NCE=0	0	*
71	DBE	dbe	IN	-	*	0
72	NBW	Nbw	OUT	NCE=0	0	*
73	MCLK	mclk	IN	-	*	0
74	NWAIT	Nwait	IN	-	*	0
75	LATEABT	LATEABT	IN	-	*	1
76	PROG32	prog32	IN	-	*	1
77	DATA32	data32	IN	-	*	1
78	NRW	Nrw	OUT	NCE=0	0	*
79	NOPC	Nopc	OUT	NCE=0	0	*
80	NMREQ	Nmreq	OUT	NCE=0	0	*
81	SEQ	seq	OUT	NCE=0	0	*
82	ABORT	abort	IN	-	*	0
83	NIRQ	Nirq	IN	-	*	1
84	NFIQ	Nfiq	IN	-	*	1
85	NRESET	Nreset	IN	-	*	0
86	ALE	ale	IN	-	*	1
87	CPB	cpb	IN	-	*	1
88	NTRANS	Ntrans	OUT	NCE=0	0	*
89	A[31]	a[31]	OUT	ABE=1	0	*
90	A[30]	a[30]	OUT	ABE=1	0	*
91	A[29]	a[29]	OUT	ABE=1	0	*
92	A[28]	a[28]	OUT	ABE=1	0	*
93	A[27]	a[27]	OUT	ABE=1	0	*
94	A[26]	a[26]	OUT	ABE=1	0	*
95	A[25]	a[25]	OUT	ABE=1	0	*
96	A[24]	a[24]	OUT	ABE=1	0	*
97	A[23]	a[23]	OUT	ABE=1	0	*
98	A[22]	a[22]	OUT	ABE=1	0	*
99	A[21]	a[21]	OUT	ABE=1	0	*
100	A[20]	a[20]	OUT	ABE=1	0	*
101	A[19]	a[19]	OUT	ABE=1	0	*
102	A[18]	a[18]	OUT	ABE=1	0	*
103	A[17]	a[17]	OUT	ABE=1	0	*
104	A[16]	a[16]	OUT	ABE=1	0	*
105	A[15]	a[15]	OUT	ABE=1	0	*
106	A[14]	a[14]	OUT	ABE=1	0	*
107	A[13]	a[13]	OUT	ABE=1	0	*
108	A[12]	a[12]	OUT	ABE=1	0	*
109	A[11]	a[11]	OUT	ABE=1	0	*
110	A[10]	a[10]	OUT	ABE=1	0	*
111	A[9]	a[9]	OUT	ABE=1	0	*
112	A[8]	a[8]	OUT	ABE=1	0	*
113	A[7]	a[7]	OUT	ABE=1	0	*
114	A[6]	a[6]	OUT	ABE=1	0	*
115	A[5]	a[5]	OUT	ABE=1	0	*
116	A[4]	a[4]	OUT	ABE=1	0	*
117	A[3]	a[3]	OUT	ABE=1	0	*
118	A[2]	a[2]	OUT	ABE=1	0	*
119	A[1]	a[1]	OUT	ABE=1	0	*
120	A[0]	a[0]	OUT	ABE=1	0	*
121	ABE	abe	INEN1	-	0	*
(TO tdo)						

KEY

IN	Input pad
OUT	Output pad
INEN1	Input enable active high
OUTEN0	Output enable active low

13.3.4 Output Enable Boundary-scan Cells

The following boundary-scan cells control the output drivers of tri-state outputs as shown:

No.	Cell name	Pin	Type	Outputs controlled
66	NENOUT	-	OUTEN0	d[31:0]
67	NCE	-	OUTEN0	Nrw,Nbw,lock,Ntrans, Nmreq,seq,Nopc,Ncpi
121	ABE	abe	INEN1	a[31:0]

In the case of type OUTEN0 enable cells (**NENOUT** & **NCE**), loading a '1' into the cell will disable the associated drivers, while in the case of type INEN1 enable cell (**ABE**), loading a '0' into the cell will disable the associated drivers.

When the SAMPLE/PRELOAD or INTEST instructions are active, the value captured in the **NENOUT** cell will reflect the state of the **NENOUT** signal from the core. However, the input of the **NCE** cell is tied permanently to Vss, so a logic '0' will always be captured by this cell if the SAMPLE/PRELOAD or INTEST instructions are active.

To put all ARM60 three-state outputs into their high impedance state, a logic '1' should be clocked into the output enable boundary-scan cells **NENOUT** & **NCE**, and a logic '0' should be clocked into **ABE**. Alternatively, the HIGHZ instruction can be used.

13.3.5 Single-step Operation

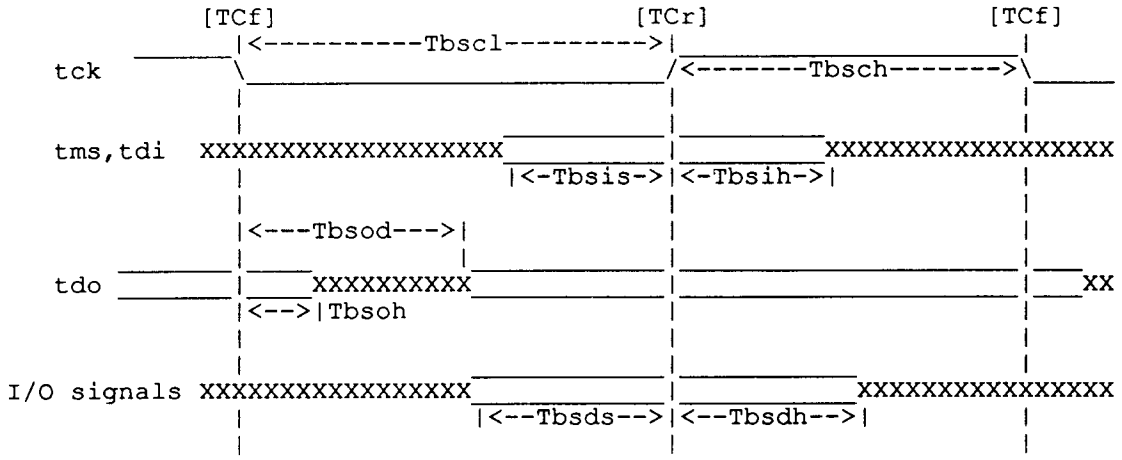
ARM60 is a static design and there is no minimum clock speed. It can therefore be single-stepped while the INTEST instruction is selected. This can be achieved by serialising a parallel stimulus and clocking the resulting serial vectors into the boundary-scan register. When the boundary-scan register is updated, new test stimuli are applied to the core logic inputs; the effect of these stimuli can then be observed on the core logic outputs by capturing them in the boundary-scan register.

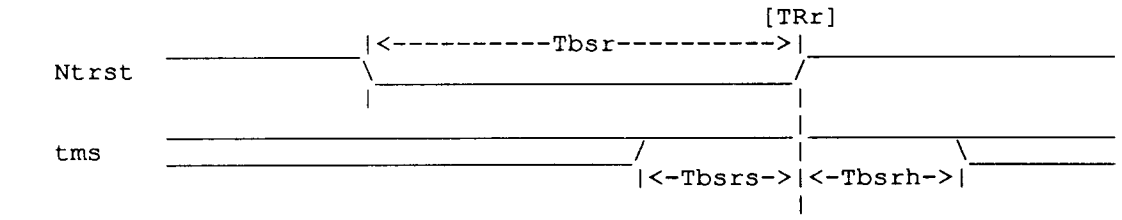
13.4 Pin information

tck, tms, tdi and **Ntrst** are TTL level inputs with on-chip pull up resistors, and **tdo** is a CMOS level output (see the chapter on DC characteristics for full details of these pins).

The **tdo** output should not be overdriven when active.

13.5 AC Parameters





Symbol	Parameter	Min	Typ	Max	Units	Note
Tbscl	tck low period	50			ns	1
Tbsch	tck high period	50			ns	1
Tbsis	tdi,tms setup to [TCr]	10			ns	
Tbsih	tdi,tms hold from [TCr]	10			ns	
Tbsod	TCf to tdo			40	ns	2
Tbsoh	tdo hold time	5			ns	2
Tbsds	I/O signal setup to [TCr]	5			ns	3
Tbsdh	I/O signal hold from [TCr]	20			ns	3
Tbsr	Reset period	50			ns	
Tbsrs	tms setup to [TRr]	10			ns	4
Tbsrh	tms hold from [TRr]	10			ns	4

NOTES

(1) **tck** may be stopped indefinintely in either the low or high phase.

(2) Assumes a 25pF load on **tdo**. Output timing derates at 0.072ns/pF of extra load applied.

(3) For correct data latching, the I/O signals (from the core and the pads) must be setup and held with respect to the rising edge of tck in the *CAPTURE-DR* state of the SAMPLE/PRELOAD, INTEST and EXTEST instructions.

(4) The **tms** input must be held high as **Ntrst** is taken high at the end of the boundary-scan reset sequence.

14. Packaging and Pinout

ARM60 is packaged in a 100 pin plastic quad flat pack (PQFP).

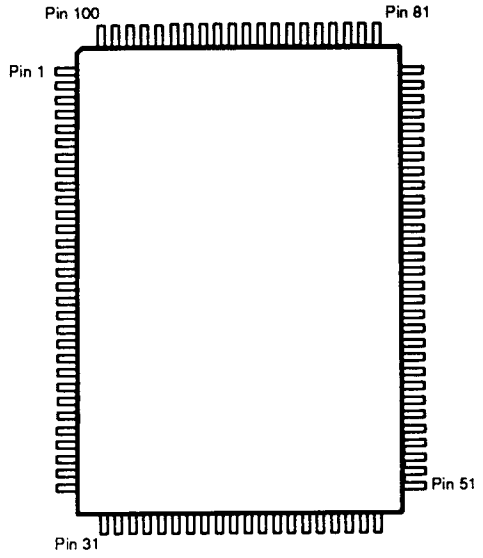


Figure 34: 100 pin PQFP Package

Pin	Name	Pin	Name	Pin	Name	Pin	Name
1	**d[27]**	26	**Nreset**	51	**VDD**	76	**d[4]**
2	**d[28]**	27	**ale**	52	**VSS**	77	**d[5]**
3	**d[29]**	28	**cpb**	53	**a[10]**	78	**d[6]**
4	**d[30]**	29	**Ntrans**	54	**a[9]**	79	**d[7]**
5	**d[31]**	30	**a[31]**	55	**a[8]**	80	**VSS**
6	**cpa**	31	**a[30]**	56	**a[7]**	81	**VDD**
7	**VSS**	32	**a[29]**	57	**a[6]**	82	**d[8]**
8	**VDD**	33	**a[28]**	58	**a[5]**	83	**d[9]**
9	**lock**	34	**a[27]**	59	**a[4]**	84	**d[10]**
10	**bigend**	35	**a[26]**	60	**a[3]**	85	**d[11]**
11	**Ncpi**	36	**a[25]**	61	**a[2]**	86	**d[12]**
12	**dbe**	37	**a[24]**	62	**a[1]**	87	**d[13]**
13	**Nbw**	38	**a[23]**	63	**a[0]**	88	**d[14]**
14	**mclk**	39	**a[22]**	64	**VSS**	89	**d[15]**
15	**Nwait**	40	**a[21]**	65	**VSS**	90	**d[16]**
16	**lateabt**	41	**a[20]**	66	**abe**	91	**d[17]**
17	**prog32**	42	**a[19]**	67	**tck**	92	**d[18]**
18	**data32**	43	**a[18]**	68	**tms**	93	**d[19]**
19	**Nrw**	44	**a[17]**	69	**Ntrst**	94	**d[20]**
20	**Nopc**	45	**a[16]**	70	**tdi**	95	**d[21]**
21	**Nmreq**	46	**a[15]**	71	**tdo**	96	**d[22]**
22	**seq**	47	**a[14]**	72	**d[0]**	97	**d[23]**
23	**abort**	48	**a[13]**	73	**d[1]**	98	**d[24]**
24	**Nirq**	49	**a[12]**	74	**d[2]**	99	**d[25]**
25	**Nfiq**	50	**a[11]**	75	**d[3]**	100	**d[26]**

GEC PLESSEY
SEMICONDUCTORS

HEADQUARTERS OPERATIONS
GEC PLESSEY SEMICONDUCTORS
Cheney Manor, Swindon,
Wiltshire SN2 2QW, United Kingdom.
Tel: (0793) 518000 Tx: 449637
Fax: (0793) 518411

GEC PLESSEY SEMICONDUCTORS
Sequoia Research Park, 1500 Green Hills Road,
Scotts Valley, California 95066,
United States of America. Tel (408) 438 2900
ITT Telex: 4940840 Fax: (408) 438 5576

CUSTOMER SERVICE CENTRES
- **FRANCE & BENELUX** Les Ulis Cedex Tel: (1) 64 46 23 45 Tx: 602858F Fax : (1) 64 46 06 07
- **GERMANY** Munich Tel: (089) 3609 06-0 Tx: 523980 Fax : (089) 3609 06-55
- **ITALY** Milan Tel: (02) 33001044/45 Tx: 331347 Fax: (GR3) 316904
- **JAPAN** Tokyo Tel: (03) 3296-0281 Fax: (03) 3296-0228
- **NORTH AMERICA Integrated Circuits,** Scotts Valley, USA Tel (408) 438 2900 ITT Tx: 4940840 Fax: (408) 438 7023.
  SOS, Microwave and Hybrid Products, Farmingdale, USA Tel (516) 293 8686 Fax: (516) 293 0061.
- **SOUTH EAST ASIA** Singapore Tel: 2919291 Fax: 2916455
- **SWEDEN** Johanneshov Tel: 46 8 7228690 Fax: 46 8 7227879
- **UNITED KINGDOM & SCANDINAVIA**
  Swindon Tel: (0793) 518510 Tx: 444410 Fax : (0793) 518582

These are supported by Agents and Distributors in major countries world-wide.

© GEC Plessey Semiconductors 1992 Publication No. DS 3553 Issue No. 1.0 August 1992