VLSI TECHNOLOGY, INC.

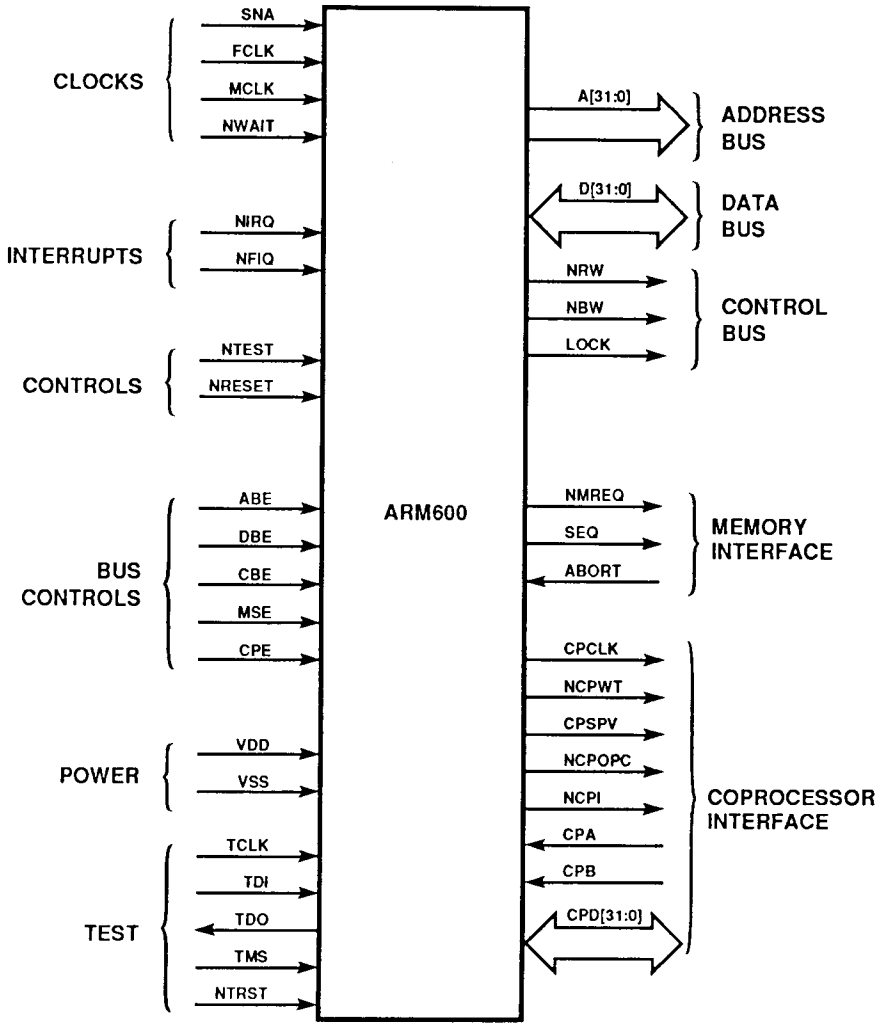
ARM600

32-BIT ENHANCED MICROPROCESSOR

FEATURES

- 32-bit data bus
- 32-bit data address bus
- Simple but powerful instruction set
- On-chip Memory Management Unit (MMU)
- On-chip 32-entry Translation Look-aside Buffer (TLB)
- On-chip 4 Kbyte Cache
- On-chip Write Buffer
- Fully static
- Low power
- Cache access speed independent of external memory timing
- Coprocessor interface for instruction set extension
- IEEE 1149.1-1990 Boundary Scan Standard Test Port
- C compiler support
- 20/23 MHz cache clock
- 12 MHz bus clock
- 160-lead Quad Flat Pack (QFP) package

FUNCTIONAL DIAGRAM



DESCRIPTION

ARM600 is a general-purpose 32-bit microprocessor with 4 Kbyte cache, Write Buffer and Memory Management Unit (MMU) combined in a single chip. It is software compatible with the ARM™ family and can be used with the existing ARM support chips.

The ARM600 architecture is based on 'Reduced Instruction Set Computers' (RISC) principles, and the instruction set and related decode mechanism are greatly simplified compared with micro-programmed 'Complex Instruction Set Computers' (CISC).

The on-chip mixed data and instruction cache substantially raises the average execution speed, and reduces the average amount of memory bandwidth required by the processor. This allows the external memory to support additional processors or Direct Memory Access (DMA) channels with minimal performance loss.

The instruction set comprises ten basic instruction types. Two of these make use of the on-chip arithmetic logic unit, barrel shifter and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide. Three classes of instruction control the transfer of data between main memory and the register bank, one optimized for flexibility of addressing, another for rapid context switching and the third for swapping data. Two instructions control the flow and privilege level of execution, and another three types are dedicated to the control of external coprocessors that 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 that depend on sophisticated compiler technology to manage complicated instruction interdependencies.

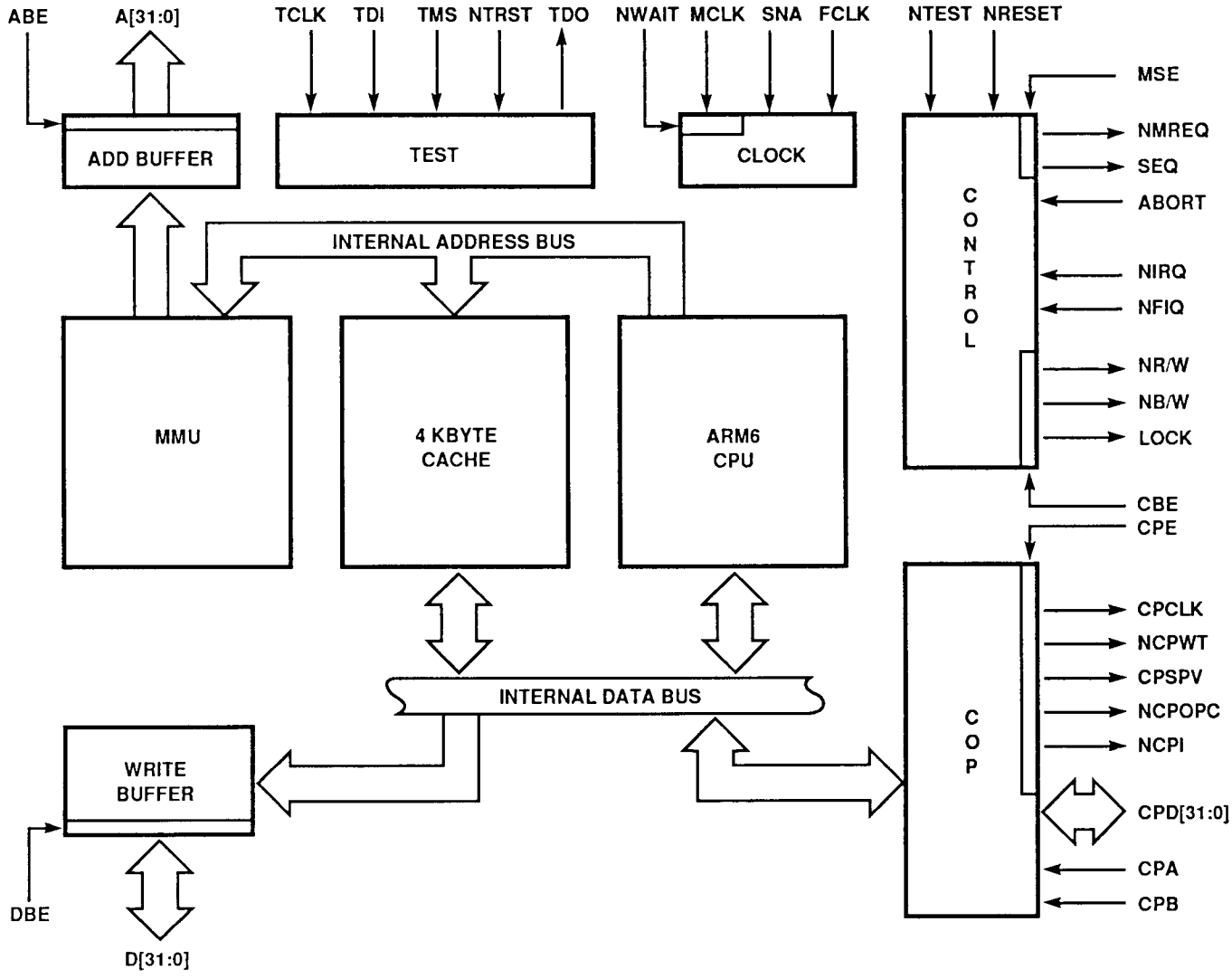
Pipelining is employed so that all parts of the processor and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched.

The memory interface has been designed to allow the performance potential to be realized 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 random access memories (DRAMs).

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

The ARM600 is a fully static part and has been designed to minimize its power requirements. This makes it ideal for portable applications where both these features are essential.

BLOCK DIAGRAM



SIGNAL DESCRIPTIONS

Name	Pin	Type	Description
A[31:0]	157-150, 147-140, 137-136, 133-123, 120-118	OCZ	Address bus. This bus signals the address requested for memory accesses. It changes during MCLK HIGH.
ABE	158	IT	Address bus enable. When this input is LOW, the address bus A[31:0] is put into a high impedance state (Note 1).
ABORT	8	IT	External abort. This input allows the memory system to tell the processor that a requested access has failed. This input is only monitored when ARM600 is accessing external memory.
NBW	160	OCZ	Not byte/word. An output signal used by the processor to indicate to the external memory system when a data transfer of a byte length is required. NBW is HIGH for word transfers and LOW for byte transfers, and is valid for both read and write operations.
CBE	4	IT	Control bus enable. When this input is LOW, the following control bus outputs are put into a high impedance state (Note 1):–NBW, LOCK, NR/W.
CPA	22	IT	Coprocessor absent. A coprocessor which is capable of performing the operation requested by ARM600 asserting NCPI, should take CPA LOW immediately. ARM 600 samples CPA when CPCLK and NCPI are both LOW. If CPA is HIGH at this time, ARM600 will abort the coprocessor handshake and take the undefined instruction trap. If CPA is LOW and remains LOW, ARM will busy-wait until CPB is LOW and then complete the coprocessor instruction. If no coprocessors are fitted, CPA must be driven HIGH.
CPB	23	IT	Coprocessor busy. A coprocessor which is capable of performing the operation requested by ARM600 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 drive CPB LOW. ARM600 samples CPB when CPCLK and NCPI are both LOW. If no coprocessors are fitted, CPB must be driven HIGH.
CPCLK	15	OCZ	Coprocessor clock. This pin, in conjunction with NCPWT, provides the clock by which all ARM600 coprocessor interactions are timed. CPCLK is derived from MCLK or FCLK depending on whether the processor is accessing external or internal memory. The coprocessors must therefore be able to operate at FCLK speeds.
CPD[31:0]	24-29, 32-40, 43-50, 53, 56-63	ITOTZ	Coprocessor Data bus. These are bidirectional signal paths which are used for data transfers between the processor and external coprocessors, as follows: for processor instruction fetches (when NCPOPC is LOW), the opcode is sent to the coprocessors by driving CPD[31:0] while CPCLK and NCPWT are HIGH. Coprocessor instructions are broadcast unaltered, but non-coprocessor instructions are replaced by the last coprocessor instruction broadcast but with CPD[26] LOW. During register and data transfers from ARM600 to a coprocessor, the data is driven onto CPD[31:0] while CPCLK and NCPWT are HIGH. During register and data transfers from the coprocessor to ARM600, CPD[31:0] are inputs, and the data must be valid around the falling edge of CPCLK.
CPDBE	115	IT	Coprocessor Data bus enable. When this input is LOW, the coprocessor data bus, CPD[31:0] is put into a high impedance state (Note 1). Note that this signal is ANDed internally with CPE. It is only used for off-board testing, and should be tied HIGH during normal use.
CPE	4	IT	Coprocessor bus enable. When this input is LOW, the following coprocessor bus drivers are put into a high impedance state (Note 1):–CPCLK, CPD[31:0], CPSPV, NCPI, NCPOPC, NCPWT
NCPI	19	OCZ	Not Coprocessor Instruction. When ARM600 executes a coprocessor instruction, it will take this output LOW and wait for a response from the appropriate coprocessor, which the coprocessor signals on the CPA and CPB inputs. The action taken will depend upon this response. NCPI changes while CPCLK is LOW.

SIGNAL DESCRIPTIONS (Cont.)

Name	Pin	Type	Description
NCPOPC	20	OCZ	Opcode fetch. NCPOPC is driven LOW to indicate to the coprocessors that an instruction will be broadcast on CPD[31:0] when CPCLK and NCPWT go HIGH. NCPOPC is held valid when CPCLK is LOW, and changes when CPCLK and NCPWT are HIGH.
CPSPV	18	OCZ	Coprocessor supervisor mode. As instructions are broadcast to the coprocessors on CPD[31:0], this output reflects the mode in which each instruction was fetched by the processor (CPSPV is HIGH for supervisor/irq/fiq mode fetches, CPSPV is LOW for user mode fetches). The coprocessors may use this information to prevent user-mode programs executing protected coprocessor instructions. CPSPV changes while CPCLK and NCPWT are HIGH.
NCPWT	14	OCZ	Not Coprocessor Wait. This signal qualifies the coprocessor clock, CPCLK. It changes while CPCLK is LOW, and must be gated with the clock within the coprocessor bus transactions.
D[31:0}	110–107, 104-96,	ITOTZ	Data bus. These are bidirectional signal paths which are used for data transfers between the processor and external memory.
	93-92, 89-83,		for read operations (when nR/W is LOW), the input data must be valid before the falling edge of MCLK.
	80-71		for write operations (when nR/W is HIGH), the output data will become valid while MCLK is LOW.
DBE	111	IT	Data bus enable. When this input is LOW, the data bus, D[31:0] is put into a high impedance state (Note 1). The drivers will always be high impedance except during write operations, and DBE must be driven HIGH in systems which do not require the data bus for DMA or similar activities.
FCLK	10	IT	Fast clock input. When the ARM600 CPU is accessing the cache, performing an internal cycle, or communicating directly with the coprocessor, it is clocked with the Fast clock, FCLK.
NFIQ	117	IT	Not fast interrupt request. If FIQs are enabled, the processor will respond to a LOW level on this input by taking the FIQ interrupt exception. This is an asynchronous, level-sensitive input, and must be held LOW until a suitable response is received from the processor.
NIRQ	116	IT	Not interrupt request. As NFIQ, but with lower priority. May be taken LOW asynchronously to interrupt the processor when the IRQ enable is active.
LOCK	3	OCZ	Locked operation. LOCK is driven HIGH, to signal a "locked" memory access sequence, and the memory manager should wait until LOCK goes LOW before allowing another device to access the memory. LOCK changes while MCLK is HIGH, and remains HIGH for the duration of the locked memory sequence.
MCLK	9	IT	Memory clock input. This clock times all ARM600 memory accesses. The LOW or HIGH period of MCLK may be stretched when accessing slow peripherals; alternatively, the NWAIT input may be used with a free-running MCLK to achieve similar effects.
NMREQ	113	OCZ	Not memory request. This is a pipelined signal that changes while MCLK is LOW to indicate whether the following cycle will be Active (processor accessing external memory) or Latent (processor not accessing external memory). An Active cycle is flagged when NMREQ is LOW.
MSE	114	IT	Memory request/sequential enable. When this input is LOW, the following control outputs are put into a high impedance state (Note 1). NMREQ, SEQ.

Note:

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

SIGNAL DESCRIPTIONS (Cont.)

Name	Pin	Type	Description
NRESET	5	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, and the on-chip caches, MMU, and write buffer to be disabled. When NRESET is driven HIGH, the processor will re-start from address 0. NRESET must remain LOW (and NWAZT must remain HIGH) for at least 2 FLCK clock cycles, and 5 MCLK clock cycles. During the LOW period the processor will perform idle cycles but with incrementing addresses.
NRW	159	OCZ	Not read/write. When HIGH this signal indicates a processor write operation; when LOW, a read operation. The signal changes while MCLK is HIGH.
SEQ	112	OCZ	Sequential address. This signal is the inverse of NMREQ, and is provided for compatibility with existing ARM memory systems.
SNA	6	IT	Synchronous/not Asynchronous. This pin determines the bus interface mode, and should be hard wired HIGH and LOW, depending on the relationship between FCLK and MCLK in the application.
NTEST	13	IT	Not test. When this input is LOW, ARM600 enters a special test mode which is only used for off-board testing. NTEST must be driven HIGH while ARM600 is in-circuit.
TCLK	64	IT	Test interface reference Clock. This times all the transfers on the test interface.
TDI	67	IT	Test interface data input.
TDO	68	OCZ	Test interface data output.
TMS	65	IT	Test interface mode select.
NTRST	66	IT	Test interface reset. Note this pin does NOT have an internal pull-up resistor.
NWAIT	7	IT	Not wait. When LOW this signal allows extra MCLK cycles to be inserted into memory accesses. It must change during the LOW phase of the MCLK cycle that is to be extended.
VDD	1, 12, 16, 30, 41, 52, 55, 69, 81, 94, 106, 121, 135, 139, 148		Positive Supply
VSS	2, 11, 17, 31, 42, 51, 54, 70, 82, 91, 95, 105, 122, 134, 138, 149		Ground Supply

PROGRAMMER'S MODEL

INTRODUCTION

ARM600 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.

ARM600 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 (scv): 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.

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 organization is shown below. 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. Registers 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

REGISTER ORGANIZATION

GENERAL REGISTERS AND PROGRAM COUNTER

User32 Mode	FIQ32 Mode	Supervisor32 Mode	Abort32 Mode	IRQ32 Mode	Undefined32 Mode
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_undef
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_undef
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

PROGRAM STATUS REGISTERS

CPRS	CPRS	CPRS	CPRS	CPRS	CPRS
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_undef

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 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) that are loaded with the CPSR when an exception occurs. There is one SPSR for each privileged mode.

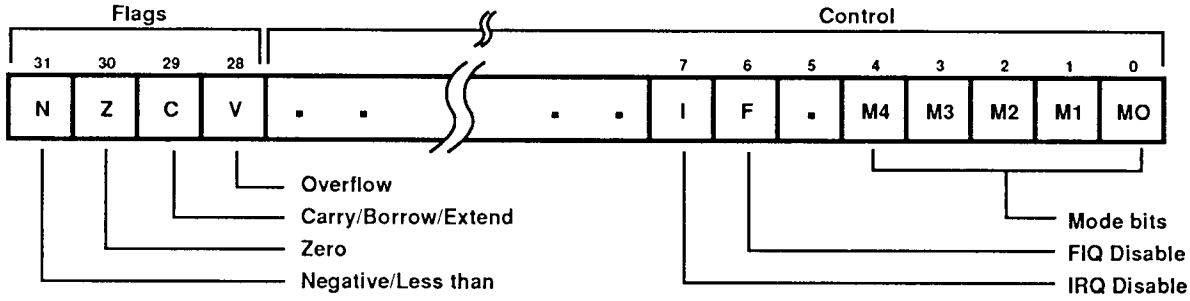
The format of the Program Status Registers is shown below. The N, Z, C and V bits are the *condition code flags*. The condition code flags in the CPSR may be changed as a result of arithmetic and logical operations in the processor and may be tested by all instructions to determine if the instruction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

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

FORMAT OF THE PROGRAM STATUS REGISTERS (PSRS)



THE MODE BITS

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

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 lo 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.

ARM600 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 that 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 (see Exception Priorities, page 10).

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 synchronization 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 minimizing 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, ARM600 checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

When a FIQ is detected, ARM600 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.

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 PC (but note that this is not possible from User mode). If the I flag is clear, ARM600 checks for a LOW level on the output of the IRQ synchronizer at the end of each instruction.

When an IRQ is detected, ARM600 performs the following:

(1) Saves the address of the next instruction to be executed plus 8 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.

Abort

The Abort signal 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. ARM600 checks for an Abort during memory access (N and S) cycles and distinguishes between two types of aborts.

(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 white 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) 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 (SWP) instruction is aborted as though it had not executed.

(c) LDM and STM instructions 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, ARM600 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 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 that 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.

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, ARM600 performs the following:

(1) Saves the addresses 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.

Undefined Instruction Trap

When ARM600 executes a coprocessor instruction or an undefined instruction, it offers it to any coprocessors which may be present. If a coprocessor signals that it can perform this instruction but is busy at that moment, ARM600 will wait until the coprocessor is ready. If no coprocessor can handle the instruction ARM600 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 ARM600 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.

Reset

When the NRESET signal goes LOW, ARM600 abandons the currently executing instruction and then continues to fetch instructions from memory that it interprets as NOPs.

When NRESET goes HIGH again, ARM600 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

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 that is only operative when the processor is configured for a 26-bit program space.

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 (i.e. the F flag in the CPSR is clear), ARM600 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 the 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.

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 synchronizer (*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 ARM600 will be executing the instruction at &1C.

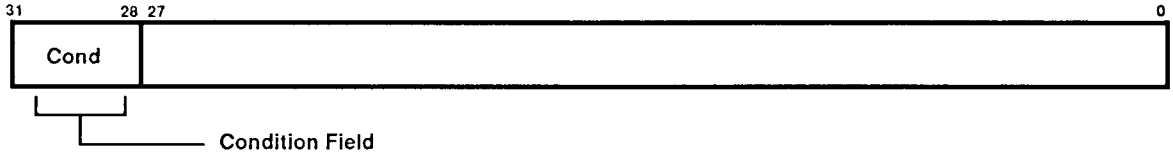
Tsyncmax is 3 processor cycles, *Tldm* is 20 cycles, *Texc* is 3 cycles, and *Tfiq* is two cycles. The total time is therefore 28 processor cycles, which is just over 1 microsecond in a system which uses a continuous 25 MHz processor clock. In a DRAM based system running at 4 and 8 MHz, this time becomes 4.5 microseconds, and if bus bandwidth is being used to support video or other DMA activity, the time will increase accordingly.

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 synchronizer (*Tsyncmin*) plus *Tfiq*. This is four processor cycles.

INSTRUCTION SET

THE CONDITION FIELD



Condition Field

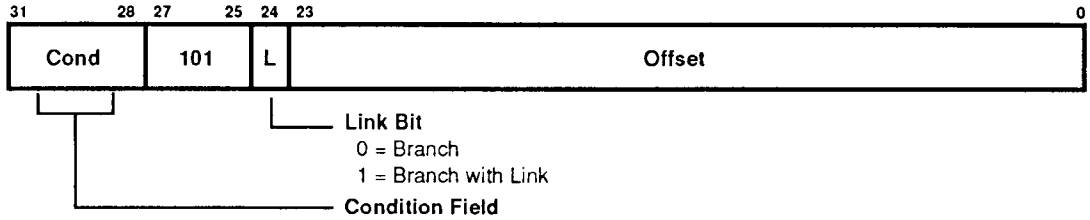
0000 = EQ – Z set (equal)
0001 = NE – Z clear (not equal)
0010 = CS – C set (unsigned higher or same)
0011 = CC – C clear (unsigned lower)
0100 = MI – N set (negative)
0101 = PL – N clear (positive or zero)
0110 = VS – V set (overflow)
0111 = VC – V clear (no overflow)
1000 = HI – C set and Z clear (unsigned higher)
1001 = LS – C clear or Z set (unsigned lower or same)
1010 = GE – N set and V set, or N clear and V clear (greater or equal)
1011 = LT – N set and V clear, or N clear and V set (less than)
1100 = GT – Z clear, and either N set and V set, or N clear and V clear (greater than)
1101 = LE – Z set, or N set and V clear, or N clear and V set (less than or equal)
1110 = AL – always
1111 = NV – never

All ARM600 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.

If the ALways condition is specified, the instruction will be executed irrespective of the flags. The NeVer 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 other condition codes have meanings as detailed above, 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.

BRANCH AND BRANCH WITH LINK (B, BL)



The instruction is only executed if the condition specified in the condition field is true.

Branch instructions contain a signed two'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 +/– 32 Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be two words (8 bytes) ahead of the current instruction.

Branches beyond +/– 32 Mbytes 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.

The Link Bit

Branch with Link 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.

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 the Condition Field (EQ, NE, VS, etc.). If absent then AL (ALways) will be used.

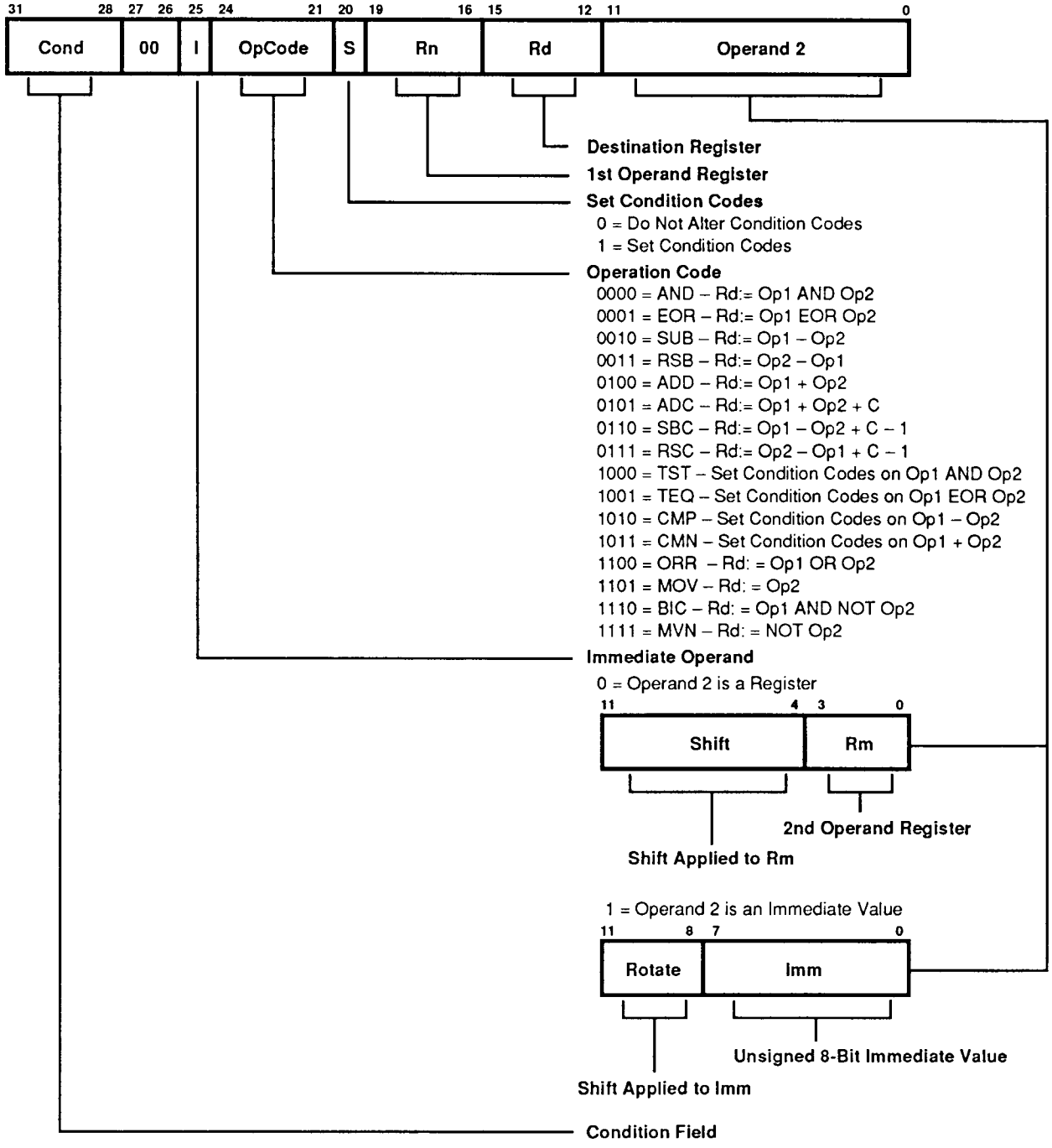
<expression> is the destination. The assembler calculates the offset.

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

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
```

DATA PROCESSING



The instruction is only executed if the condition specified in the condition field is true.

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 Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set.

Operations

The operations supported are:

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)

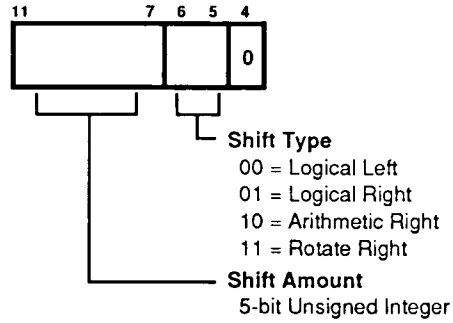
CPSR Flags

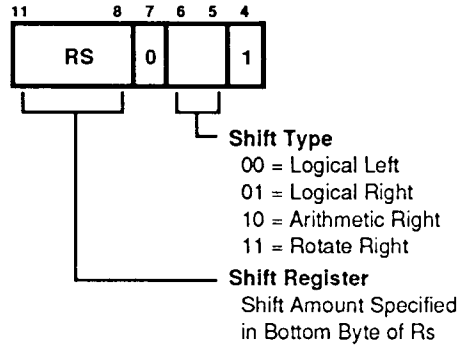
The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15) the V flag in the CPSR 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 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 two'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 two'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 two's complement signed).

Shifts

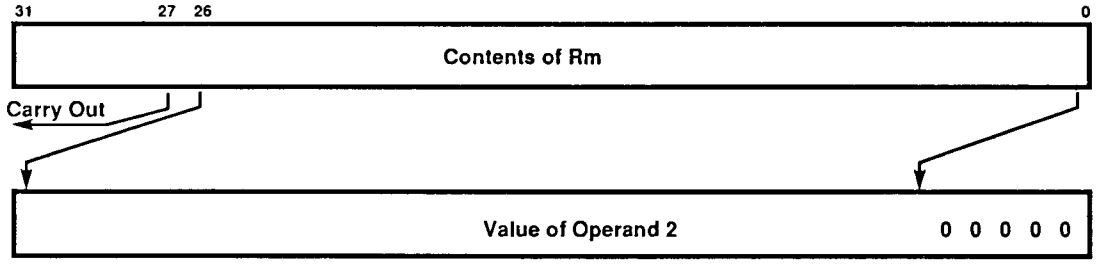
When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should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





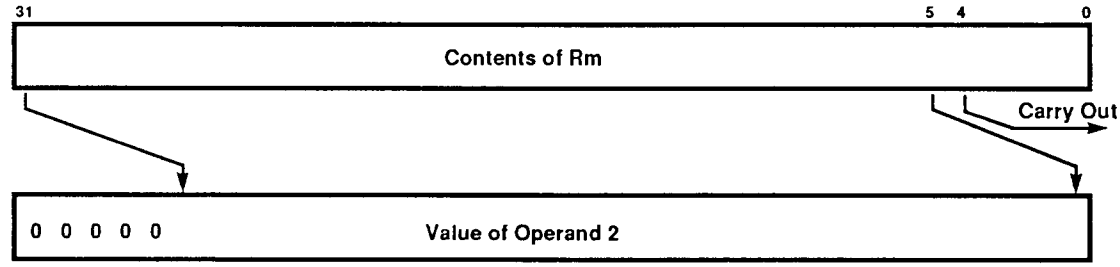
INSTRUCTION SPECIFIED SHIFT AMOUNT

When the shift amount is specified in the instruction, it is contained in a 5-bit field that 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 that 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. For example, the effect of LSL #5 is:



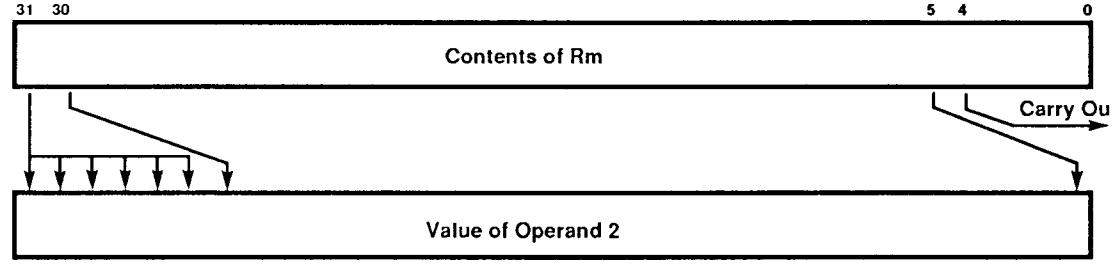
Note that LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

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



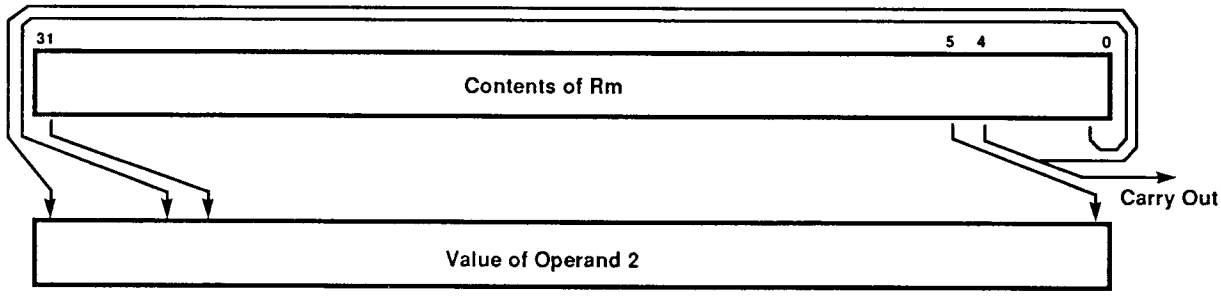
The form of the shift field that might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and 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 two's complement notation. For example, ASR #5:



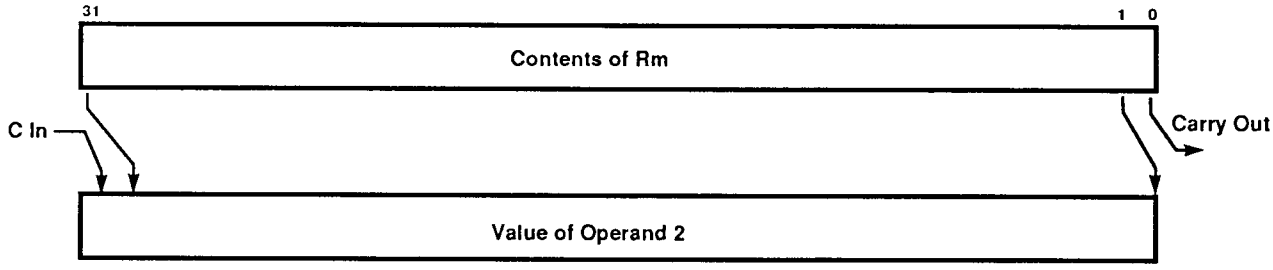
The form of the shift field that might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

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:



The form of the shift field that 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:



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 a data swap instruction.

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 two.

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.

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.

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.

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.

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.

{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.

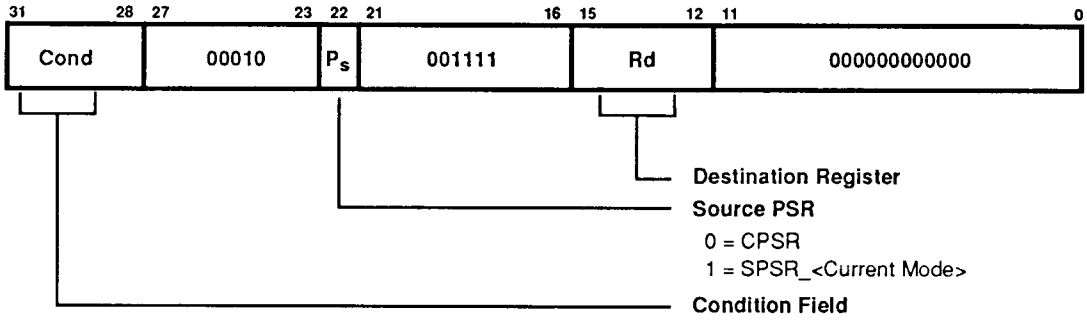
(ASL is a synonym for LSL, the two assemble to the same code.)

Examples

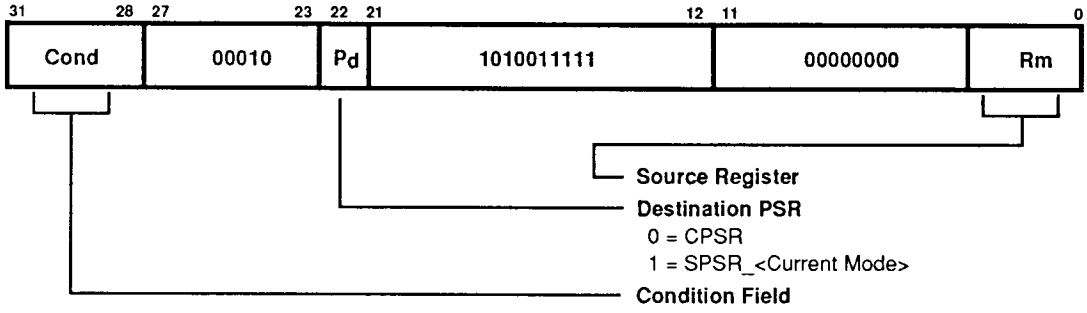
```
ADDEQ R2,R4,R5          ; if the Z flag is set make R2:=R4+R5
TEQS R4,#3              ; test R4 for equality with 3
                        ; (the S is in fact redundant as the
                        ; assembler inserts it automatically)
SUB R4, R5, R7, LSR R2  ; logical right shift R7 by the number 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>
```

PSR TRANSFER (MRS, MSR)

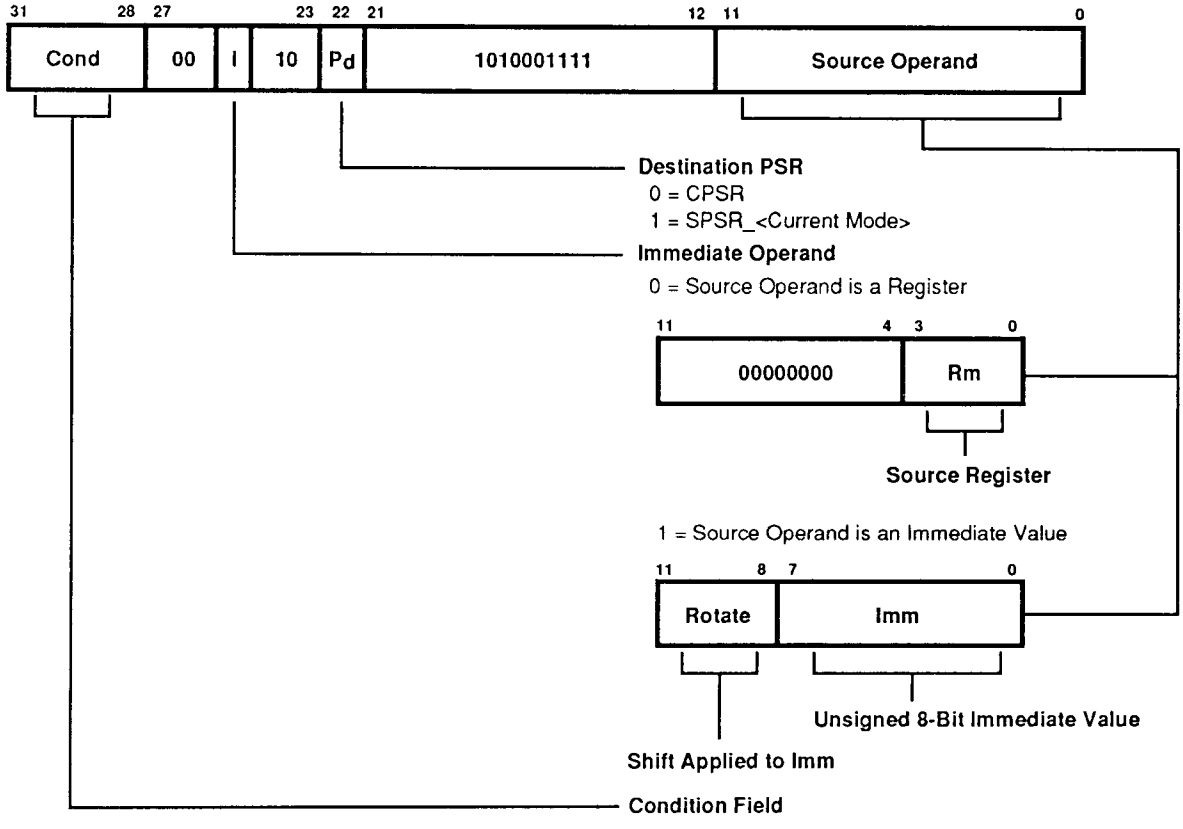
MRS (Transfer PSR Contents to a Register)



MSR (Transfer Register Contents to PSR)



MSR (Transfer Register Contents or Immediate Value to PSR Flag Bits Only)



These instructions allow access to the CPSR and SPSR registers and are only executed if the condition is true. 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 V3) 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.

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.

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

Reserved bits

Only eleven bits of the PSR are defined in ARM600 (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 ARM600 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 pack 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.

Assembler syntax

(1) MRS - transfer PSR contents to a register

MSR{cond} Rd,<cpsr>

(2) MSR - transfer register contents to PSR

MSR{cond} <psr>,Rm

(3) 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.

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

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

The expression should symbolize 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.

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.

Examples

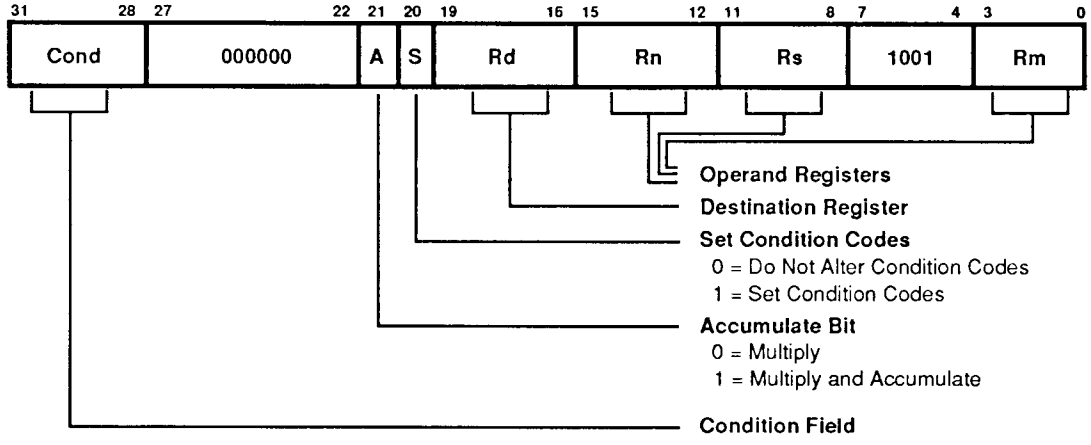
In User mode the instructions behave as follows:

```
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[3l: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]
```

MULTIPLY AND MULTIPLY-ACCUMULATE (MUL, MLA)



The instruction is only executed if the condition specified in the condition field is true.

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 (two's complement) or unsigned integers.

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.

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.

Assembler Syntax

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

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

{cond} – two-character condition mnemonic.

{S} – set condition codes if S present.

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

Examples

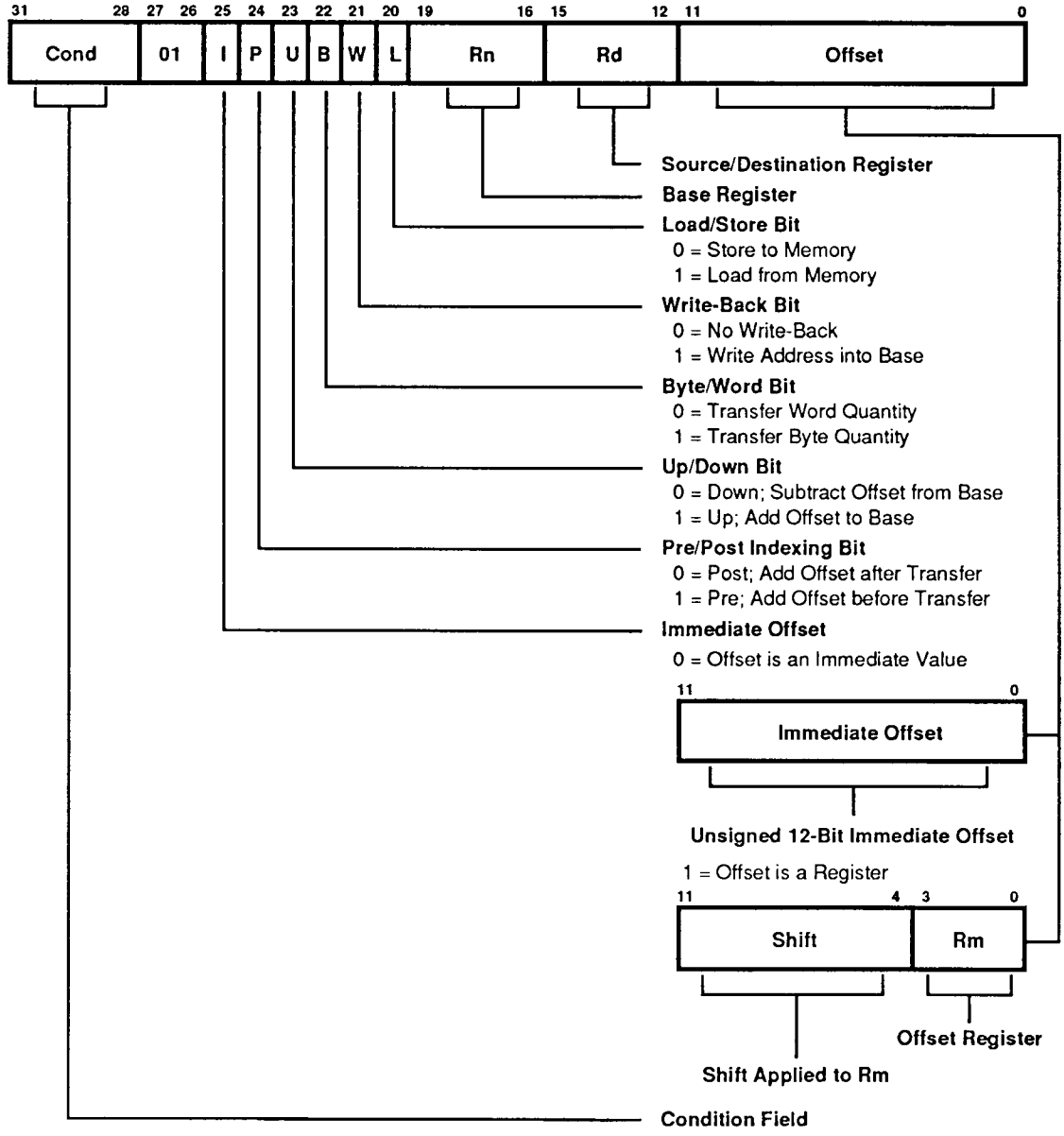
```
MUL R1, R2, R3          ; R1:=R2*R3
MLAEQS R1, R2, R3, R4   ; conditionally R1:=R2*R3+R4,
                        ; setting condition codes
```

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.)

SINGLE DATA TRANSFER (LDR, STR)



The instruction is only executed if the condition specified in the condition field is true.

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.

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.

Shifted Register Offset
The 8 shift control bits are described in the data processing instructions but the register specified shift amounts are not available in this instruction class.

Bytes and Words
This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM600 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 D[7:0] if the supplied address is on a word boundary, on D[15:8] 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 filled with zeros.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across D[31:0]. 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 D[31].

BIG ENDIAN CONFIGURATION
A byte load (LDRB) expects the data on D[31:24] if the supplied address is on a word boundary, on D[23:16] 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 D[31:0]. 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 D[31].

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.

Restriction on the Use of Base Register
When configured for late aborts, the following code is very difficult to unwind as Rm 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.

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 signal 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.

The ARM600 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.

Assembler Syntax

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

LDR – load from memory into a register.

STR – store from a register into memory.

{cond} – two-character condition mnemonic.

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

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 ARM600 pipelining. In this case base write-back shall not be specified.

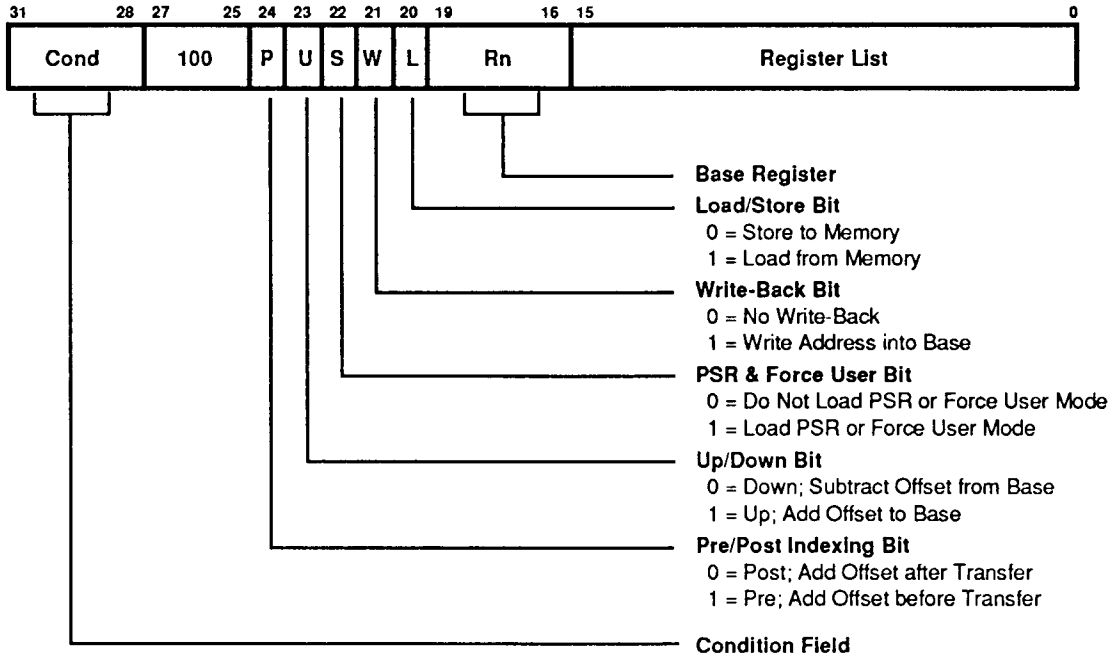
<shift> is a general shift operation (see section on Data Processing Instructions) but note that the shift amount may not be specified by a register.

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

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
```

BLOCK DATA TRANSFER (LDM, STM)



The instruction is only executed if the condition specified in the condition field is true.

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 the main memory.

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.

Addressing Modes

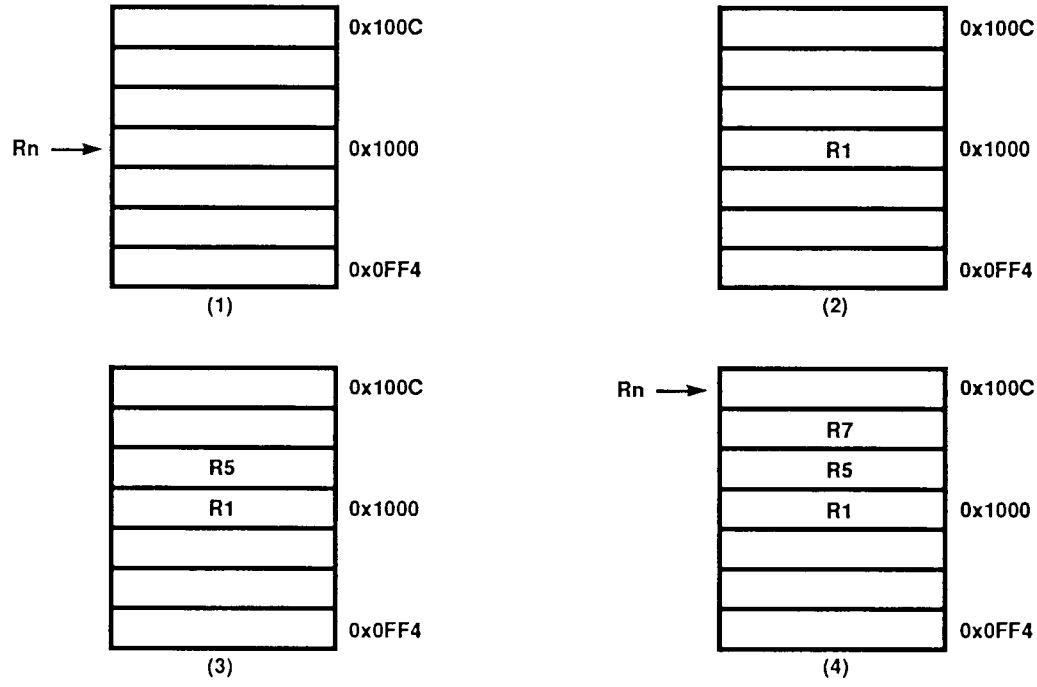
The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest 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.)

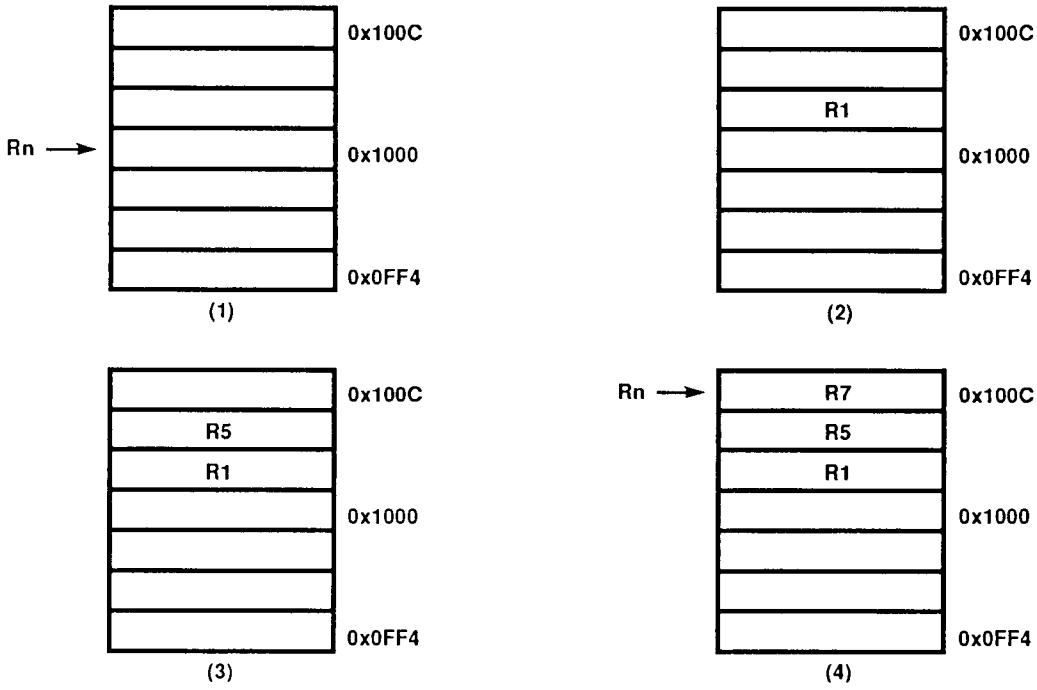
Address Alignment

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

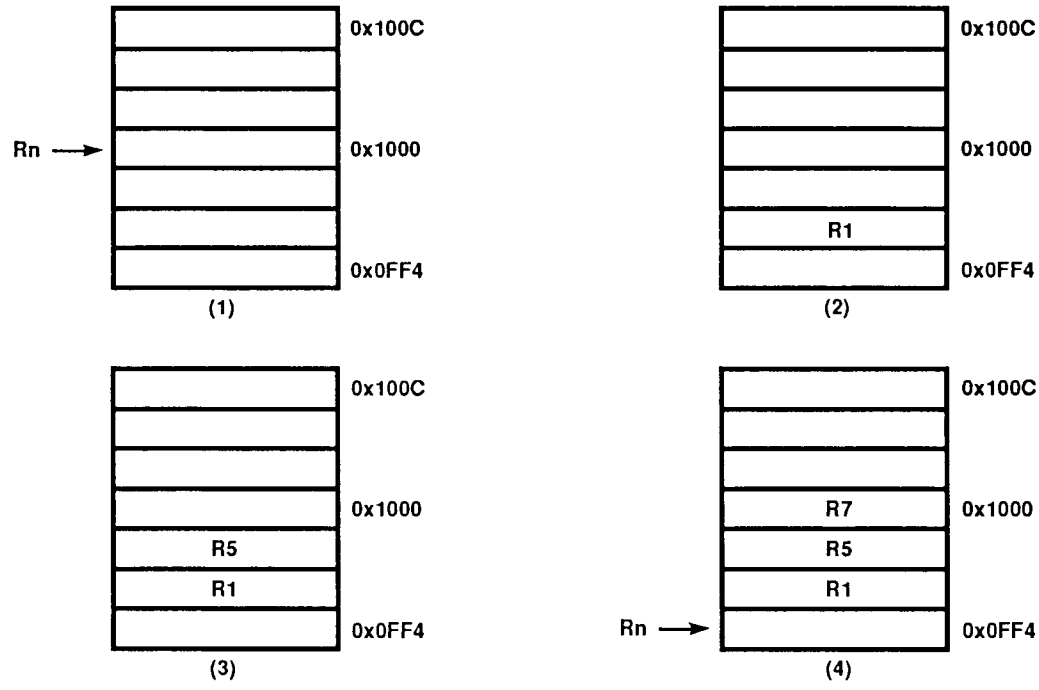
POST-INCREMENT ADDRESSING



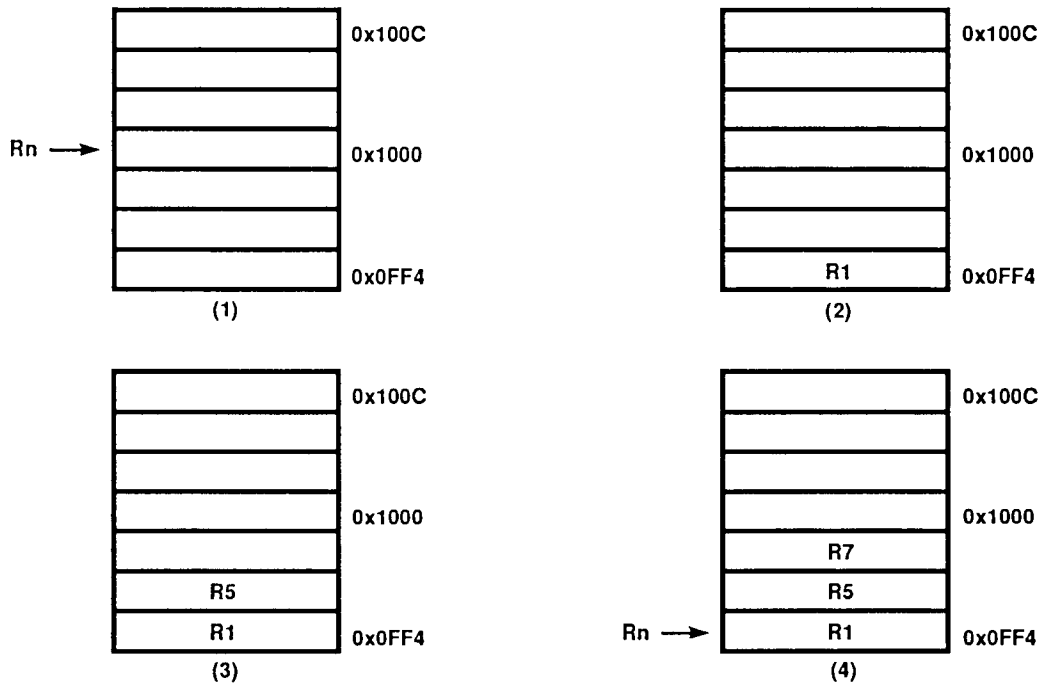
PRE-INCREMENT ADDRESSING



POST-DECREMENT ADDRESSING



PRE-DECREMENT ADDRESSING



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 use 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).

Use of R15 as the Base

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

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.

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 ARM600 is to be used in a virtual memory system.

The state of the lateabt input does not affect the behavior 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, ARM600 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 ARM600 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.

Assembler Syntax

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

{cond} – two character condition mnemonic.

Rn is an expression evaluating to a valid register number.

<Rlist> can be either a list of registers and register ranges enclosed in {} (e.g. {R0, R2-R7, R10}), or an expression evaluating to the 16-bit operand.

{!} 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 as shown in the adjacent table.

NAMES AND BIT PATTERNS

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.

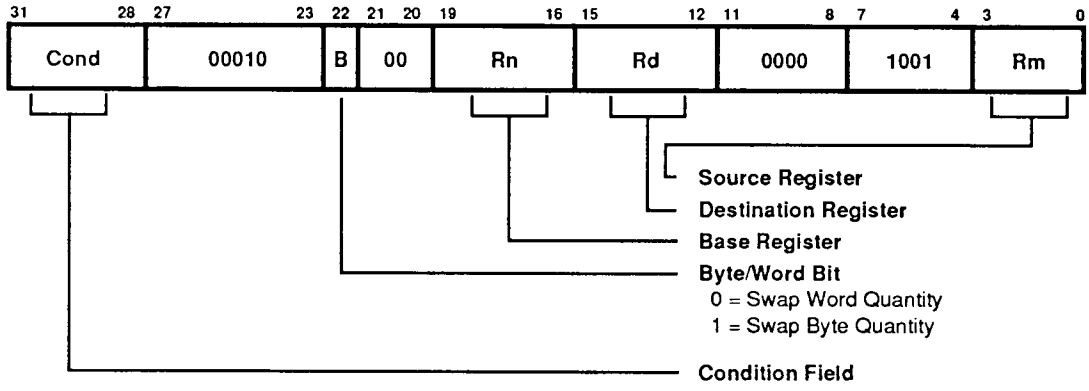
These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

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)
```

```
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
```

SINGLE DATA SWAP (SWP)



The instruction is only executed if the condition specified in the condition field is true.

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 processor while it is performing a locked operation.

Bytes and Words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM600 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.

Use of R15

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

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 behavior of an aborted SWP instruction is the same regardless of the state of the LATEABT configuration input.

Assembler Syntax

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

{cond} – two-character condition mnemonic.

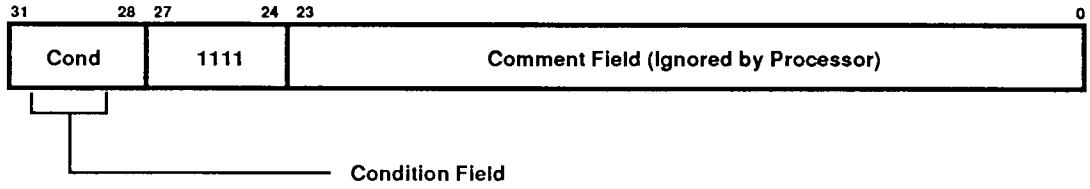
{B} – if B is present then byte transfer, otherwise word transfer.

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

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
```

SOFTWARE INTERRUPT (SWI)



The instruction is only executed if the condition specified in the condition field is true.

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. The PC is forced to a fixed value (&08) and the CPSR is saved in SPSR_svc. If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

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. If the supervisor code wishes to use software interrupts within itself, it must first save a copy of the return address and SPSR.

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.

Assembler Syntax

SWI{cond} <expression>

{cond} – two character condition mnemonic.

<expression> is evaluated and placed in the comment field (which is ignored by ARM600).

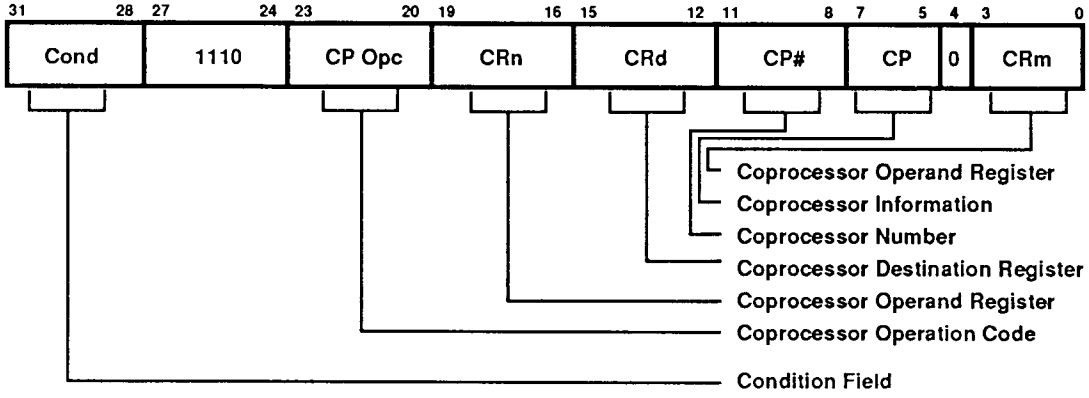
Examples

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
```

COPROCESSOR DATA OPERATIONS



COPROCESSOR DATA OPERATIONS (CDP)

The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to ARM600, 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 ARM600 activity allowing the coprocessor and ARM600 to perform independent tasks in parallel.

The Coprocessor Fields

Only bit 4 and bits 24 to 31 are significant to ARM600; 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.

Assembler Syntax

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

{cond} – two character condition mnemonic.

CP# – the unique number of the required coprocessor.

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

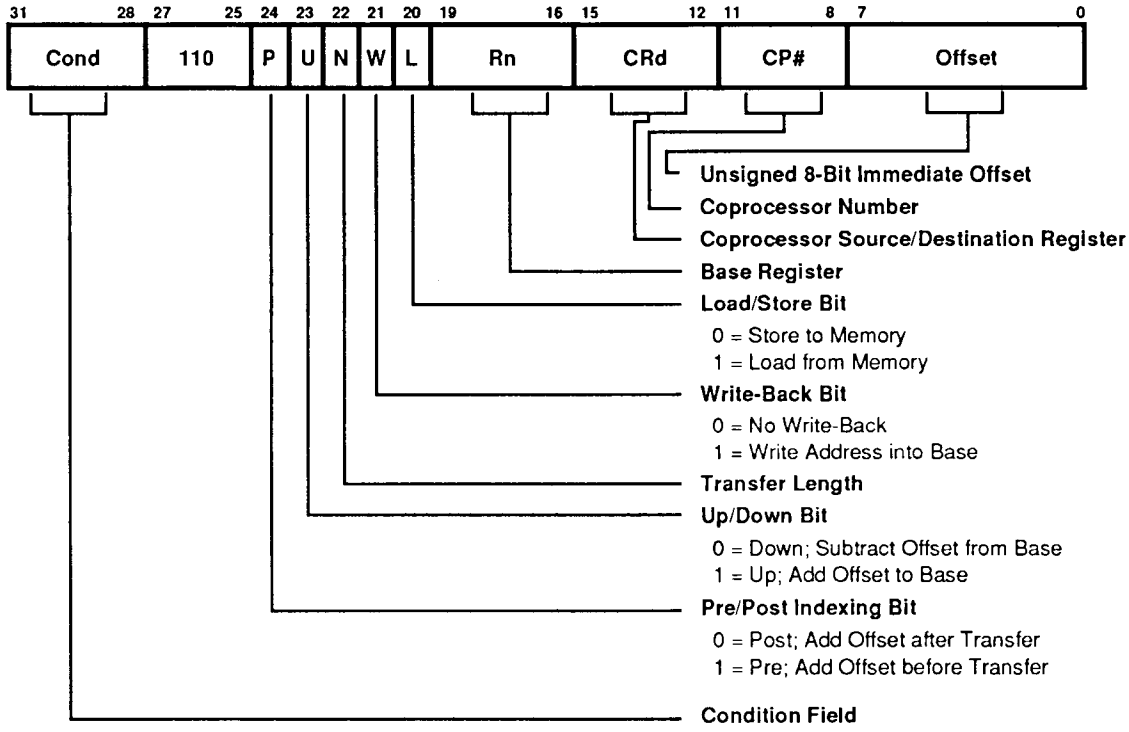
CRd, CRn and CRm are expressions evaluating to a valid coprocessor register number.

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

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
```

COPROCESSOR DATA TRANSFERS (LDC, STC)



The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to transfer one or more words of data between a coprocessor and main memory. ARM600 is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred. This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessor's registers directly to memory.

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.

Addressing Modes

ARM600 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.

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.

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

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 input does not affect the behavior of LDC and STC instructions in the event of an Abort exception.

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.

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 ARM600 register number. NOTE if Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM600 pipelining.

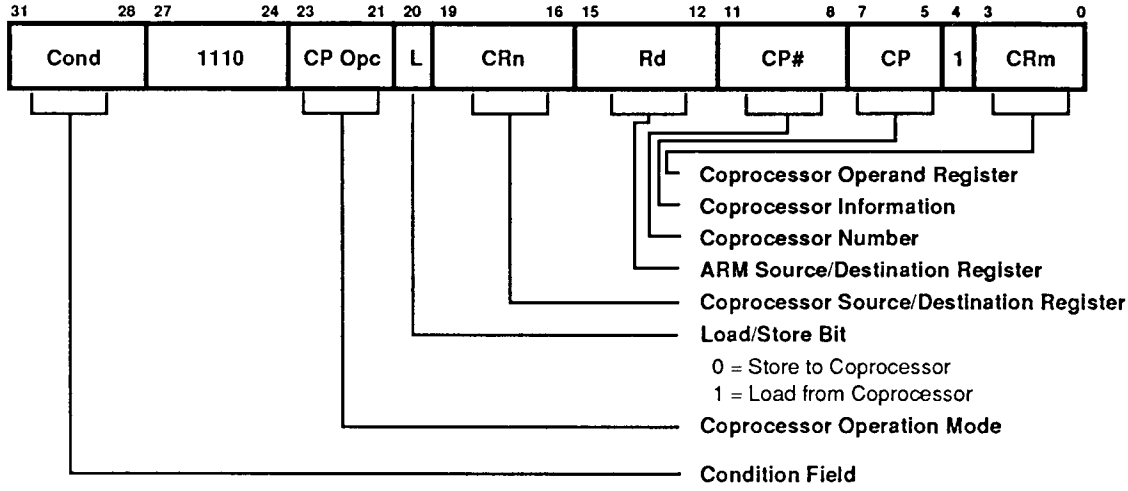
{!} write back the base register (set the W bit) if ! is present.

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.

COPROCESSOR REGISTER TRANSFERS (MRC, MCR)



The instruction is only executed if the condition specified in the condition field is true.

This class of instruction is used to communicate information directly between ARM600 and a coprocessor. An example of a coprocessor to ARM600 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 ARM600 register. A FLOAT of a 32-bit value in an ARM600 register into a floating point value within the coprocessor illustrates the use of an ARM600 register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the ARM600 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.

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.

Transfers to R15

When a coprocessor register transfer to ARM600 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.

Transfers From R15

A coprocessor register transfer from ARM600 with R15 as the source register will store the PC+12.

Assembler Syntax

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

MRC – move from coprocessor to ARM600 register (L=1).

MCR – move from ARM600 register to coprocessor (L=0).

{cond} – two character condition mnemonic.

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 ARM600 register number.

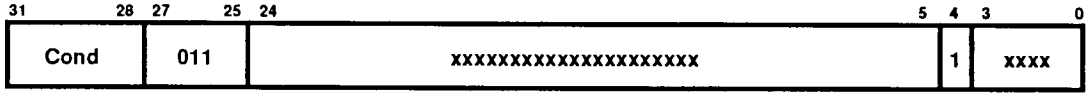
CRn and CRm are expressions evaluating to a valid coprocessor register number.

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

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
```

UNDEFINED INSTRUCTION, (Note 1)

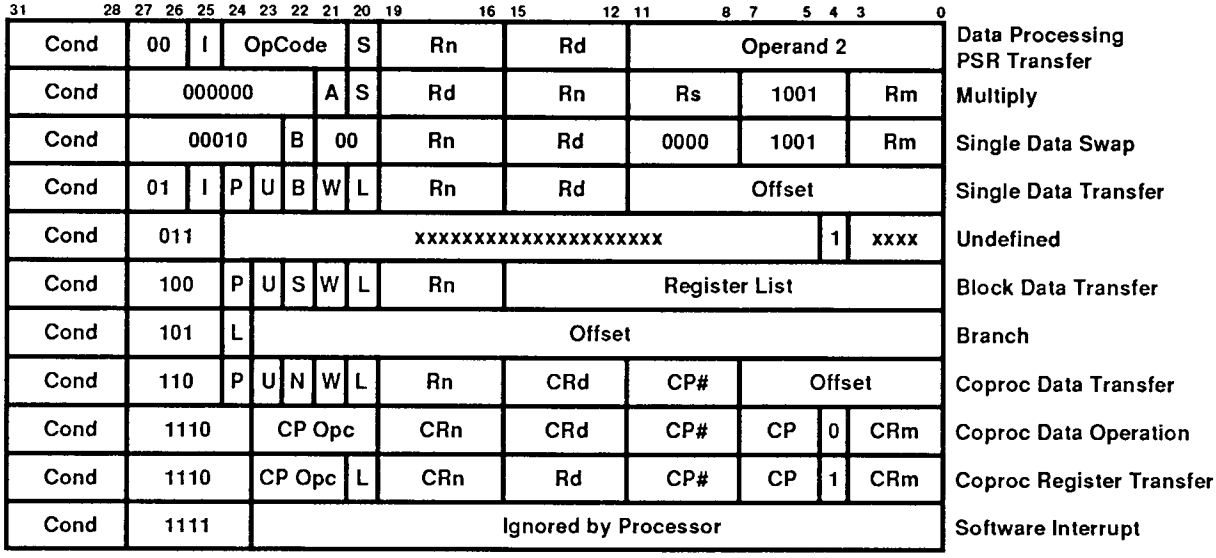


If the condition specified in the condition field is true, the undefined instruction trap will be taken.

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.

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.

INSTRUCTION SET SUMMARY, (Note 1)



Note:
1. 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.

INSTRUCTION SET EXAMPLES

The following examples show ways in which the basic ARM600 instructions can be combined to provide more efficient code.

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
```

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 1sb), 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 1sb of Rb
        EOR     Rc, Rc, Ra, LSL#12  ; (involved!)
        EOR     Ra, Rc, Rc, LSR#20  ; (similarly involved!)
;
; new seed in Ra, Rb as before
```

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 method is not quite optimal, but close. An example of where it is less than optimal 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
```

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
```

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
```

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
```

ADDITIONAL INSTRUCTIONS FOR ARM600

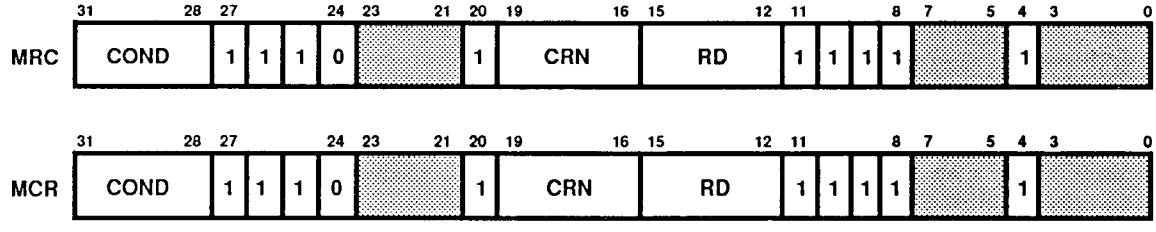
The operation and configuration of ARM600 is controlled both directly via coprocessor instructions and indirectly via the Memory Management Page tables. The coprocessor instructions manipulate a number of on-chip registers which control the configuration of the Cache, Write Buffer, MMU and a number of other configuration options.

To ensure backwards compatibility of future CPUs, all reserved or unused bits in registers and coprocessor instructions shall be programmed to '0'. Invalid registers must not be read/ written. The following bits shall be programmed to '0'.

Register 1 bits[30:9]
Register 2 bits[13:0]
Register 5 bits[31:0]
Register 6 bits[11:0]
Register 7 bits[31:0]

INTERNAL COPROCESSOR INSTRUCTIONS

The on-chip registers may be read using MRC instructions and written using MCR instructions. These operations are only allowed in non-user modes and the undefined instruction trap will be taken if accesses are attempted in user mode.

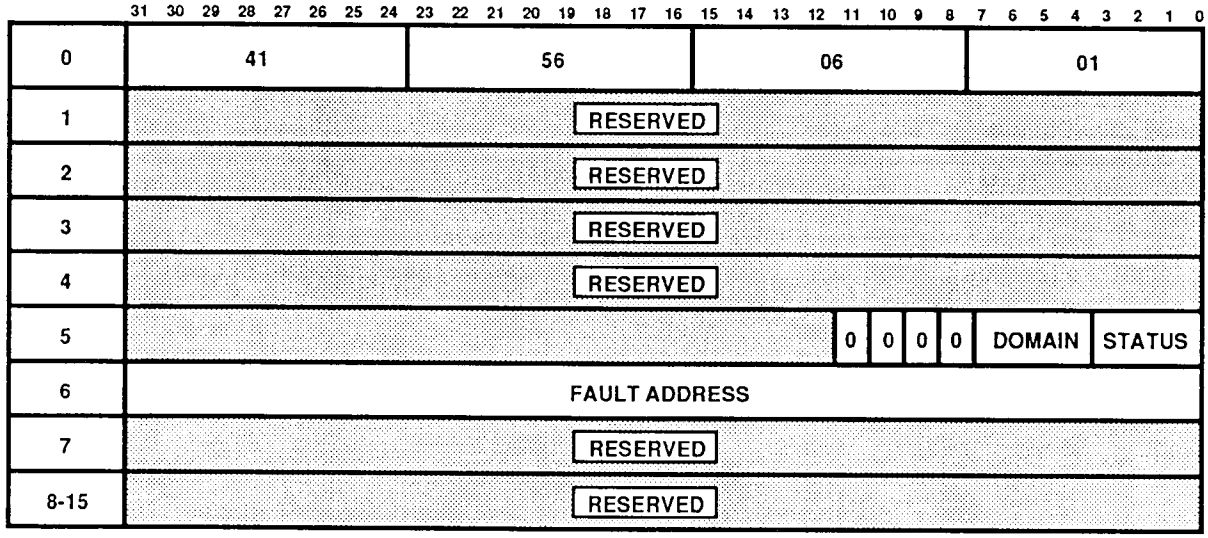


Cond = ARM condition codes
CRn = ARM600 Register
Rd = ARM Register

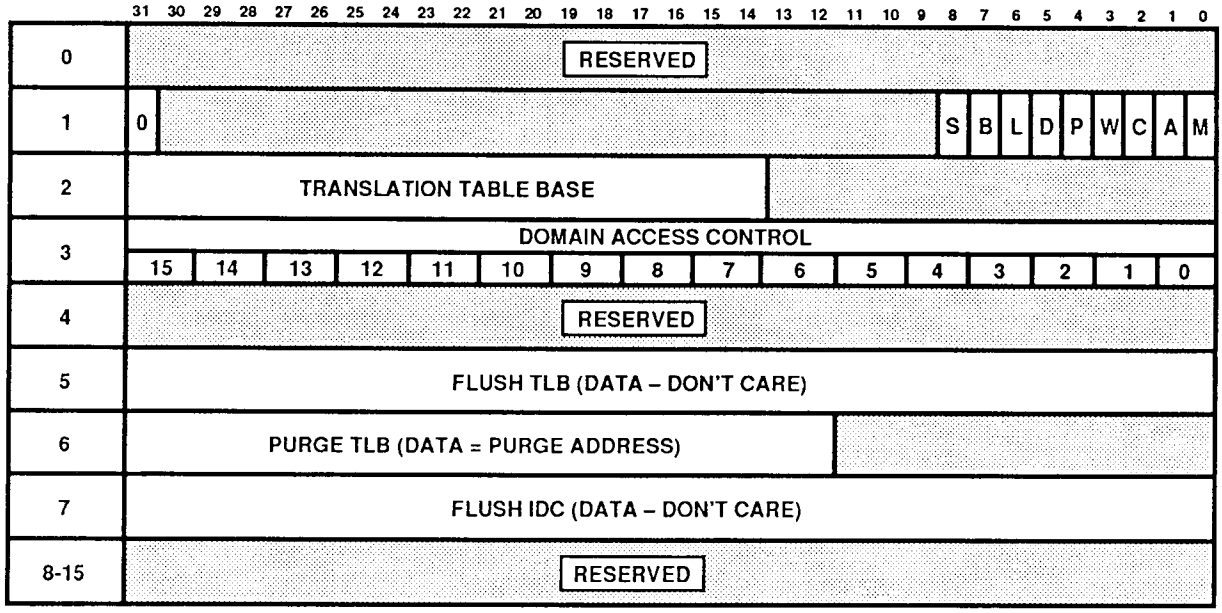
ARM600 REGISTERS

ARM600 contains registers which control the cache and MMU operation. These registers are accessed using CPRT instructions to Coprocessor #15 with the processor in a privileged mode. Only some of registers 0-7 are valid: an access to an invalid register will cause neither the access nor an undefined instruction trap, and therefore should never be carried out; an access to any of the registers 8-15 will cause the undefined instruction trap to be taken.

REGISTER READS

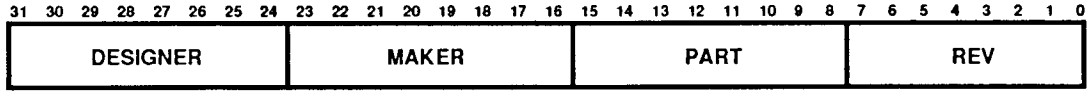


REGISTER WRITES



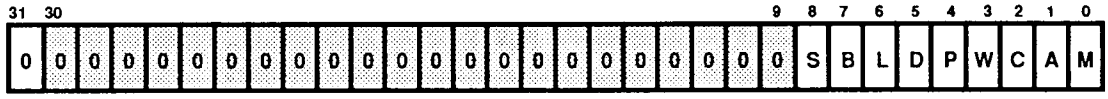
REGISTER 0. ID

Register 0 is a read-only identity register that returns the ARM Ltd code for this chip. The code returned on initial product is 41560601.



REGISTER 1. CONTROL

Register 1 is a write only register containing a number of control bits. All bits in this register are forced LOW by reset.



M Bit 0 MMU Enable/disable
- 0 – On-chip Memory Management Unit turned off
- 1 – On-chip Memory Management Unit turned on

A Bit 1 Address Fault Enable/Disable
- 0 – Alignment fault disabled
- 1 – Alignment fault enabled

C Bit 2 Cache Enable/Disable
- 0 – Instruction/data cache turned off
- 1 – Instruction/data cache turned on

W Bit 3 Write buffer Enable/Disable
- 0 – Write buffer turned off
- 1 – Write buffer turned on

P Bit 4 ARM 32/26 Bit Program Space
- 0 – 26-bit Program Space selected
- 1 – 32-bit Program Space selected

D Bit 5 ARM 32/26 Bit Data Space
- 0 – 26-bit Data Space selected
- 1 – 32-bit Data Space selected

L Bit 6 Late Abort Timing
- 0 – Early abort mode selected
- 1 – Late abort mode selected

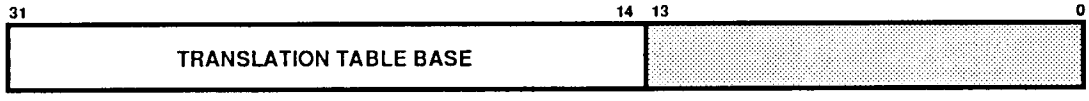
B Bit 7 Big/Little Endian
- 0 – Little-endian operation
- 1 – Big-endian operation

S Bit 8 System

This bit controls the ARM600 permission system selection.

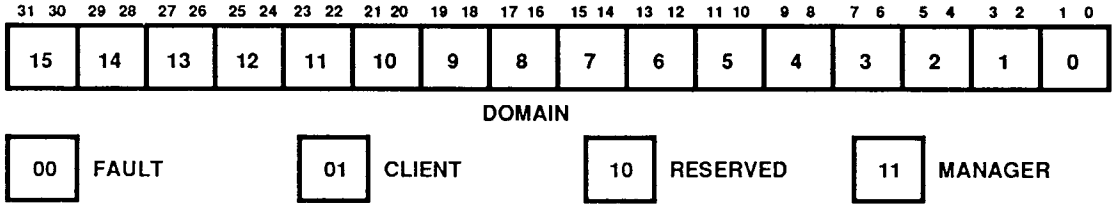
REGISTER 2. TRANSLATION TABLE BASE

Register 2 is a write-only register which holds the base of the currently active Level One page table.



REGISTER 3. DOMAIN ACCESS CONTROL

Register 3 is a write-only register which holds the current access control for domains 0 to 15. See the section on the MMU for more details.



REGISTER 4. RESERVED

Register 4 is Reserved. Accessing this register has no effect, but should never be attempted.

REGISTER 5.

Read: Fault Status

Reading register 5 returns the status of the last *data* fault. It is not updated for a *prefetch* fault. (See the section on the MMU for more details.) Note that only the bottom 12 bits are returned. The upper 20 bits will be the last value on the internal data bus, and therefore will have no meaning. Bits 11:8 are always returned as zero.

Write: Translation Lookaside Buffer Flush

Writing Register 5 flushes the TLB. (The data written is discarded).

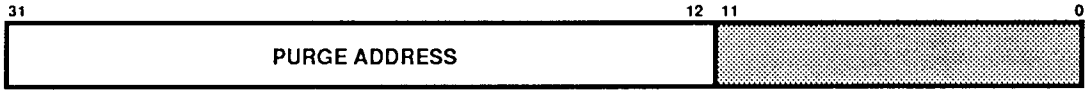
REGISTER 6.

Read: Fault Address

Reading register 6 returns the virtual address of the last *data* fault.

Write: TLB Purge

Writing register 6 purges the TLB; the data is treated as an address and the TLB is searched for a corresponding page table descriptor. If a match is found, the corresponding entry is marked as invalid. This allows the page table descriptors in main memory to be updated and invalid entries in the on-chip TLB to be purged without requiring the entire TLB to be flushed.



REGISTER 7. IDC FLUSH

Register 7 is a write-only register. The data written to this register is discarded and the Instruction Data Cache is flushed.

REGISTER 8-15. RESERVED

Accessing any of these registers will cause the undefined instruction trap to be taken.

INSTRUCTION AND DATA CACHE (IDC)

ARM600 contains a 4 Kbyte mixed instruction and data cache; the IDC has 256 lines of 16 bytes (4 words), organized as 4 blocks of 64 lines (making it 64-way set associative), and uses the virtual addresses generated by the processor core. The IDC is always reloaded a line at a time (four words). It may be enabled or disabled via the ARM600 Control Register and is disabled on RESET. The operation of the cache is further controlled by two bits: Cacheable and Updateable, which are stored in the Memory Management Page Tables. For this reason, in order to use the IDC, the MMU must be enabled. The two functions may however be enabled simultaneously, with a single write to the Control Register.

THE CACHEABLE BIT

The Cacheable bit determines whether data being read may be placed in the IDC and used for subsequent read operations. Typically main memory will be marked as Cacheable to improve system performance, and I/O space as Non-cacheable to stop the data being stored in ARM600's cache. (For example, if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of initial data held in the cache). The Cacheable bit can be configured for both pages and sections.

THE UPDATEABLE BIT

The Updateable bit determines whether the data in the cache should be updated during a write operation to maintain consistency with the external memory. (In certain cases automatic updating of cached data is not required: for instance, when using the MEMC1a memory manager, a read operation in the address space between 3400000H-3FFFFFFH would access the ROMs, but a write operation in the same address space would change a MEMC register, and should not affect the cached ROM data). The Updateable bit can only be configured by the Level One descriptor: that is an entire section or all the pages for a single Level One descriptor share the same configuration.

IDC OPERATION

When the processor performs a read or write operation, the translation entry of that address is inspected and the state of the Cacheable and Updateable bits determines the subsequent action.

Cacheable Reads C=1

The cache is searched for the relevant data; if found in the cache, the data is fed to the processor using fast clock cycle (from FCLK). If the data is not found in the cache, an external memory access is initiated to read the appropriate line of data (4 words) from external memory and it is stored in a pseudo-randomly chosen entry in the cache (a linefetch operation).

Uncacheable Reads C=0

The cache is not searched for the relevant data; instead an external memory access is initiated. No linefetch operation is performed, and the cache is not updated.

Updateable Writes U=1

An external memory access is initiated, and the cache is searched; if the cache holds a copy of the data from the address being written to, then the cache data is simultaneously updated.

Non-Updateable Writes U=0

An external memory access is initiated, but the cache is not searched and the contents of the cache are not affected.

IDC VALIDITY

The IDC operates with virtual addresses, so care must be taken to ensure that its contents remain consistent with the virtual to physical mappings performed by the Memory Management Unit. If the Memory Mappings are changed, the IDC validity must be ensured.

Software IDC Flush

The entire IDC may be marked as invalid by writing to the ARM600 IDC Flush Register (Register 7). The cache will be flushed immediately the register is written, but note that the following two instruction fetches may come from the cache before the register is written.

Doubly Mapped Space

Since the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency, since each virtual address will have a separate entry in the cache, and only one entry will be updated on a processor write operation. To avoid any cache inconsistencies, both doubly-mapped virtual addresses should be marked as uncacheable.

READ-LOCK-WRITE

The IDC treats the Read-Lock-Write instruction as a special case. The read phase always forces a read of external memory, regardless of whether the data is contained in the cache. The write phase is treated as a normal write operation (and if marked as Update-able, and the data is already in the cache, the cache will be updated). Externally the two phases are flagged as indivisible by asserting the LOCK signal.

IDC ENABLE/DISABLE AND RESET

The IDC is automatically disabled and flushed on RESET. Once enabled, cacheable read accesses will cause lines to be placed in the cache. If subsequently disabled, no new lines will be placed in the cache, and the cache is not searched, but, Updateable write operations will continue to operate, thus maintaining consistency with the external memory. If the cache is subsequently re-enabled, it must be flushed if data already in the cache no longer matches that in external memory.

To Enablethe IDC

Ensure the MMU is enabled (set bit 0 in Control Register).

Enable the IDC (set bit 2 in Control Register). (Note 1)

To Disable the IDC

Disable the IDC (clear bit 2 in Control Register). (Note 2)

Notes:

1. The MMU and IDC may be enabled simultaneously.
2. Updateable writes continue but no line fetches are performed. To fully inhibit the cache's operation it should be disabled and then flushed to ensure it contains no valid entries.

WRITE BUFFER (WB)

The ARM600 Write Buffer is provided to improve system performance. It can buffer up to 8 words of data, and 2 independent addresses. It may be enabled or disabled via the W bit (bit 3) in the ARM600 Control Register and the buffer is disabled and flushed on RESET. The operation of the Write Buffer is further controlled by one bit, B, or Bufferable, which is stored in the Memory Management Page Tables. For this reason, in order to use the Write Buffer, the MMU must be enabled. The two functions may, however, be enabled simultaneously, with a single write to the Control Register. For a write to use the Write Buffer, both the W bit in the Control Register, and the B bit in the corresponding page table must be set.

BUFFERABLE BIT

This bit controls whether a write operation may or may not use the Write Buffer. (Typically main memory will be bufferable and I/O space unbufferable). The Bufferable bit can be configured for both pages and sections.

WRITE BUFFER OPERATION

When the CPU performs a write operation, the translation entry for that address is inspected and the state of the B bit determines the subsequent action. If the Write Buffer is disabled via the ARM600 Control Register, bufferable writes are treated in the same way as unbuffered writes.

Bufferable Write

If the Write Buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the Write Buffer at FCLK speeds and the CPU continues execution. The Write Buffer then performs the external write in parallel. If however the Write Buffer is full (either because there are already 8 words of data in the buffer, or because there is no slot for the new address) then the processor is stalled, until there is sufficient space in the buffer.

Unbufferable Writes

If the Write Buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write completes externally, which may require synchronization and several external clock cycles.

Read-Lock-Write

The write phase of a read-lock-write sequence is treated as an Unbuffered write, even if it is marked as buffered. Note that a single write requires one address slot and one data slot in the write buffer; a sequential write of n words requires one address slot and n data slots. The total of 8 data slots in the buffer may be used as required. So for instance there could be one non-sequential write and one sequential write of 7 words in the buffer, and the processor could continue as normal: a third write or an eighth word in the second write would stall the processor until the first write had completed.

To Enable the Write Buffer

Ensure the MMU is enabled (set bit 0 in Control Register).

Enable the Write Buffer (set bit 3 in Control Register). (Note 1)

To Disable the Write Buffer

Disable the Write Buffer (clear bit 3 in Control Register). (Note 2)

EXTERNAL COPROCESSORS

COPROCESSOR INTERFACE

The functionality of the ARM600 instruction set may be extended by the addition of up to 15 external coprocessors numbered from 0 to 14. When a particular coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the relevant coprocessor hard ware will then increase the system performance in a software-compatible way.

The coprocessor interface timing is specified by CPCLK and NCPWT, signals generated by ARM600. CPCLK is derived from either MCLK or FCLK depending on whether the CPU is accessing external memory or the cache or write buffer, the coprocessors must therefore be able to operate at FCLK speeds. NCPWT is output to qualify CPCLK. The bus transactions controlled by the coprocessor should be controlled by the AND of CPCLK and NCPWT. NCPWT changes during CPCLK LOW. Exporting the raw CPCLK permits faster clock for the coprocessor to use for its own internal cycles. A coprocessor cycle is defined to be the period between consecutive falling edges of qualified CPCLK, though the timing is defined with respect to CPCLK itself. Seven dedicated signals control the coprocessor interface, CPCLK coprocessor clock, coprocessor wait (NCPWT), coprocessor instruction (NCPI), coprocessor absent (CPA), coprocessor busy (CPB), coprocessor opcode (NCPOPC) and coprocessor supervisor (CPSPV).

Notes:

1. The MMU and Write Buffer may be enabled simultaneously.
2. Any writes already in the write buffer will complete normally.

External coprocessors monitor the data flow between the processor and the cache, and must be capable of operating at the FCLK frequency of ARM600. The inter-chip delays associated with transmitting data between the coprocessor and processor are such that a single cycle of pipelining has been introduced between devices. This means that coprocessors see instructions one cycle after the processor fetched them, and therefore all coprocessor instruction handshaking takes an extra cycle at the start while the coprocessor catches up with the processor.

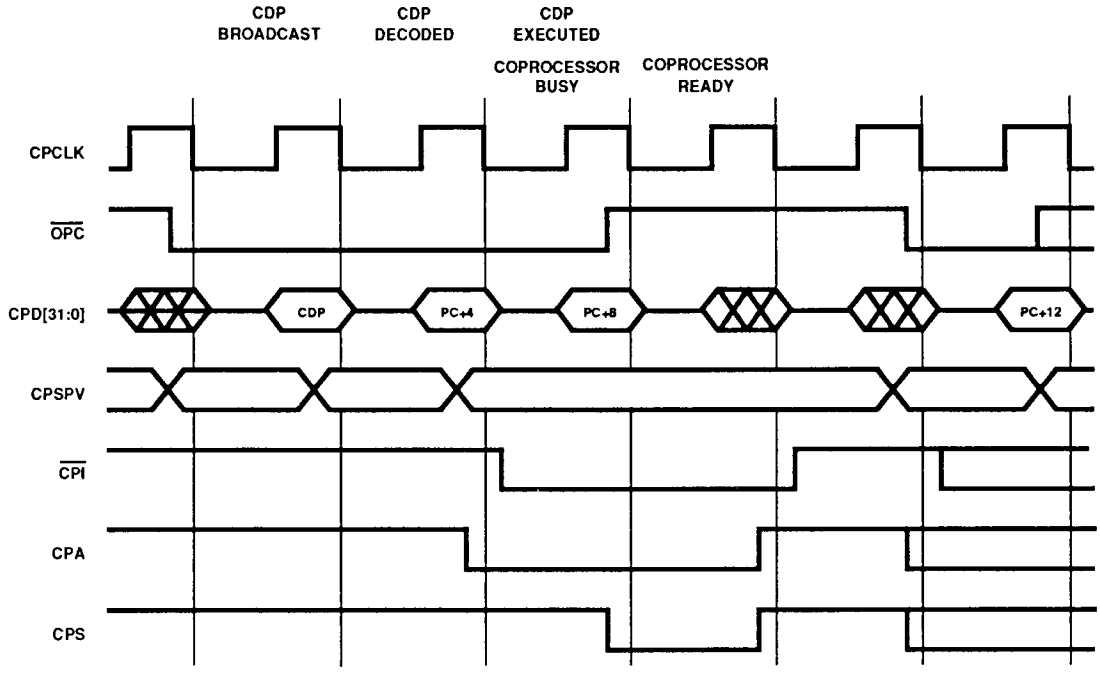
The extra cycle does allow instructions to be "vetted," so that only real coprocessor instructions (and undefined instructions), are broadcast, thus saving power. At other times, the last coprocessor instruction is left on the bus, but with CPD[26] forced LOW. This is then interpreted by the coprocessor as not a coprocessor instruction and not an undefined instruction.

In the case of CPRT and CPDT operations where data is transmitted from the coprocessor to the processor/ memory, the input data passes through another single cycle pipeline (to ensure proper data setup times for the transfer). In this case, the coprocessor goes from running one cycle behind the processor, to running one cycle ahead.

PIPELINE FOLLOWING

In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. This is achieved by having each coprocessor maintain a copy of the processor's instruction pipeline. If nOPC is LOW when CPCLK is LOW, then ARM600 will broadcast a processor instruction that cycle. The coprocessors should latch the instruction off CPD[31:0] at the end of the cycle (as CPCLK falls when NCPWT is HIGH) and clock it into their instruction pipelines. To reduce the number of transitions on CPD[31:0], ARM600 inspects the instruction stream and only broadcasts coprocessor instructions and undefined instructions.

COPROCESSOR EXAMPLE



COPROCESSOR PRESENT/ABSENT

ARM600 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 CPID field to see which coprocessor it is for. Every coprocessor in a system must have a unique number and if the number matches the contents of the CPID field, the coprocessor should pull the CPA (coprocessor absent) line LOW. If no coprocessor has a number which matches the CP# field, CPA must be pulled HIGH, and ARM600 will take the undefined instruction trap. Otherwise ARM600 observes the CPA line going LOW, and waits until the coprocessor flags that it is not busy (using CPB).

ARM600's internal control registers are implemented as coprocessor 15. External coprocessors may use numbers between 0 and 14.

BUSY-WAITING

If CPA goes LOW, ARM600 will watch the CPB (coprocessor busy) line. Only the coprocessor which is pulling CPA LOW is allowed to drive CPB LOW, and it should do so when it is ready to complete the instruction. ARM600 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 ARM600 will return from processing the interrupt to retry the coprocessor instruction.

When CPB goes LOW, the instruction continues to completion; in the case of register transfer or data transfer instructions, this will involve data transfers taking place along the coprocessor data bus (CPD[31:0]) between the coprocessor and ARM600. Data operations do not transfer any data, and complete as soon as the coprocessor ceases to be busy.

All three interface signals are sampled by both ARM600 and the coprocessor(s) on the rising edge of CPCLK when NCPWT is HIGH. If all three are LOW, the instruction is committed to execution, and where transfers are involved they will start in the next CPCLK cycle. If NCPI has gone HIGH after being LOW, and before the instruction is committed, ARM600 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. An external pull-up resistor is normally required on both CPA and CPB.

DATA TRANSFER CYCLES

Once the coprocessor has gone not-busy in a data transfer instruction, it must supply or accept data at the ARM600 bus rate (defined by CPCLK). The direction of transfer is defined by the L-bit in the instruction being executed. The coprocessor is responsible for determining the number of words to be transferred; ARM600 will continue to increment the address by one work per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor releasing CPA and CPB to float HIGH.

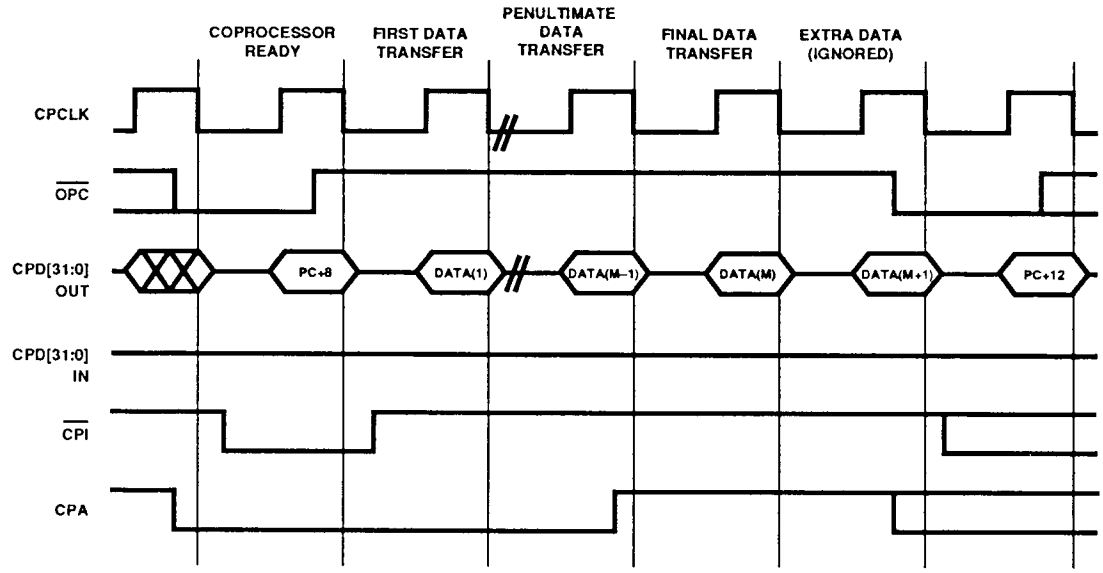
The data being transferred to/from memory is pipelined by one cycle within ARM600. In the case of a coprocessor load from memory, this means that ARM600 is one word ahead of the coprocessor, and always fetches one extra word of data. This extra fetch will not adversely affect ARM600 or the coprocessor, but may cause unexpected faults in the memory system (e.g. if the extra fetch accessed a read-sensitive peripheral).

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 ARM600 interrupt latency, since the instruction is not interruptible 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.

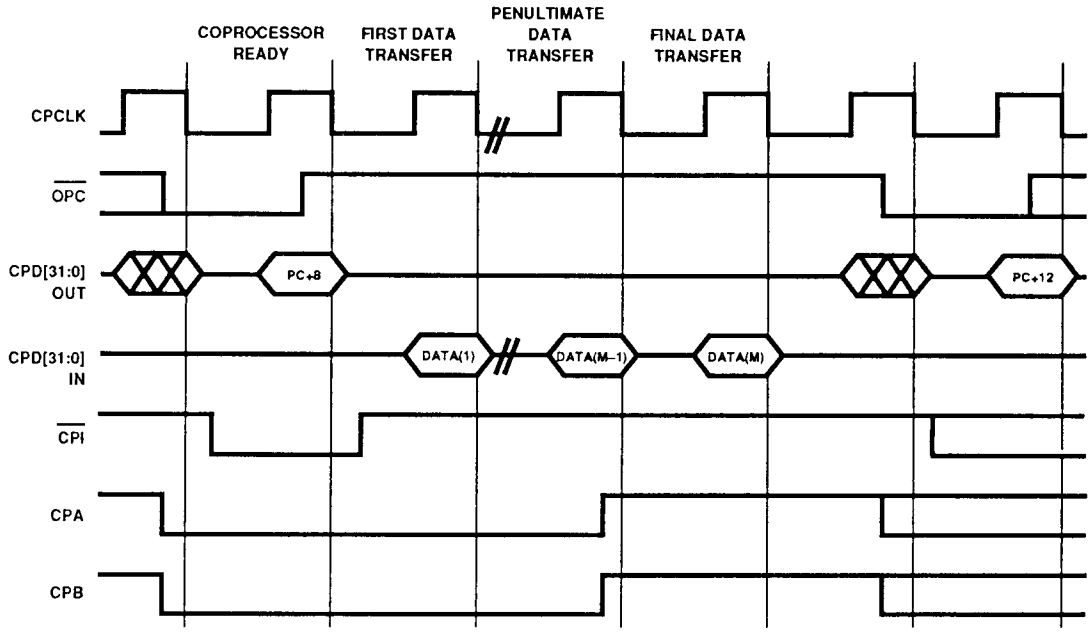
REGISTER TRANSFER CYCLE

Register transfer operations involve the transfer of a single word between ARM600 and the appropriate coprocessor along CPD[31:0]. The transfer takes place in the cycle after the one in which ARM600 and the coprocessor committed to the instruction.

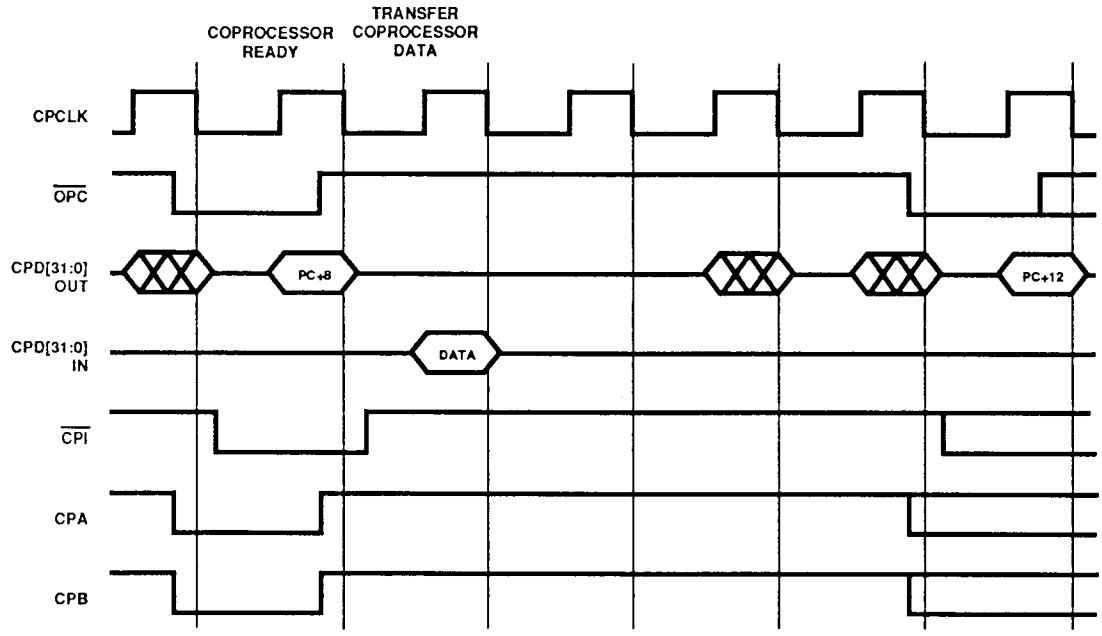
COPROCESSOR DATA TRANSFER (FROM MEMORY TO COPROCESSOR)



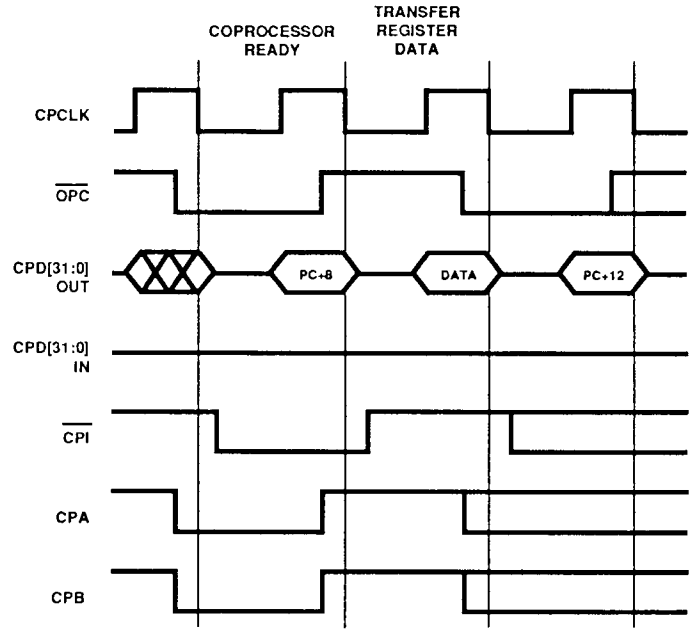
COPROCESSOR DATA TRANSFER (FROM COPROCESSOR TO MEMORY)



COPROCESSOR REGISTER TRANSFER (LOAD FROM COPROCESSOR)



COPROCESSOR REGISTER TRANSFER (STORE TO COPROCESSOR)



PRIVILEGED INSTRUCTIONS

The coprocessor may restrict certain instruction for use in a privileged (non-user) mode only. To do this, the coprocessor may use the CPSPV output of ARM600; this signal is valid while CPCLK is LOW, and applies to the instruction being broadcast during that cycle. When CPSPV is HIGH, the broadcast instruction is privileged.

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 realize 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.

IDEMPOTENCY

A consequence of the implementation of the coprocessor interface, with the interruptible 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, i.e. must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to an ARM600 register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as ARM600 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 ARM600 to be held up until the result is generated, because the result is confined to stay within the coprocessor.

MEMORY MANAGEMENT UNIT

The MMU performs two primary functions: it translates virtual addresses into physical addresses, and it controls memory access permissions. The MMU hardware required to perform these functions consists of a Translation Look-aside Buffer (TLB), access control logic, and translation table walking logic.

The MMU supports memory accesses based on Sections or Pages. Sections are comprised of 1 MB blocks of memory. Two different page sizes are supported: Small Pages consist of 4Kb blocks of memory and Large Pages consist of 64 Kb blocks of memory. (Large Pages are supported to allow mapping of a large region of memory while using only a single entry in the TLB). Additional access control mechanisms are extended within Small Pages to 1 Kb Sub-Pages and within Large Pages to 16 Kb Sub-Pages.

The MMU also supports the concept of domains – areas of memory that can be defined to possess individual access rights. The Domain Access Control Register is used to specify access rights for up to 16 separate domains.

The TLB caches 32 translated entries. During most memory accesses, the TLB provides the translation information to the access control logic.

If the TLB contains a translated entry for the virtual address, the access control logic determines whether access is permitted. If access is permitted, the MMU outputs the appropriate physical address corresponding to the virtual address. If access is not permitted, the MMU signals the CPU to abort.

If the TLB misses (it does not contain a translated entry for the virtual address), the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. Once retrieved, the translation information is placed into the TLB, possibly overwriting an existing value. The entry to be overwritten is chosen cyclically.

When the MMU is turned off (as happens on RESET), the virtual address is output directly onto the physical address bus.

MMU PROGRAM ACCESSIBLE REGISTERS

The ARM600 Processor provides several 32-bit registers which determine the operation of the MMU. The format for these registers is shown directly below. A brief description of the registers is provided below. Each register will be discussed in more detail within the section that describes its use.

Data is written to and read from the MMU's registers using the ARM CPU's MRC and MCR coprocessor instructions.

Translation Table Base Register holds the physical address of the base of the translation table maintained in main memory. Note that this base must reside on 16 Kb boundaries.

Domain Access Control Register consists of 16 two-bit fields, each of which defines the access permissions for one of the 16 domains (D15-D0).

Fault Status Register indicates the domain and type of access being attempted when an abort occurred. Bits 7:4 specify which of the 16 domains (D15-D0) was being accessed when a fault occurred. Bits 3:1 indicate the type of access being attempted. The encoding of these bits is different for internal and external faults (as indicated by bit 0 in the register) and is shown in the table on page 56. A write to this register flushes the TLB.

Fault Address Register holds the virtual address of the access which was attempted when a fault occurred. A write to this register causes the data written to be treated as an address and, if it is found in the TLB, the entry is marked as invalid. (This operation is known as a TLB purge). The Fault Status Register and Fault Address Register are only updated for data faults, not for prefetch faults.

ADDRESS TRANSLATION

The MMU translates virtual addresses generated by the CPU into physical addresses to access external memory, and also derives and checks the access permission. Translation information, which consists of both the address translation data and the access permission data resides in a translation table located in physical memory. The MMU provides the logic needed to traverse this translation table, obtain the translated address, and check the access permission.

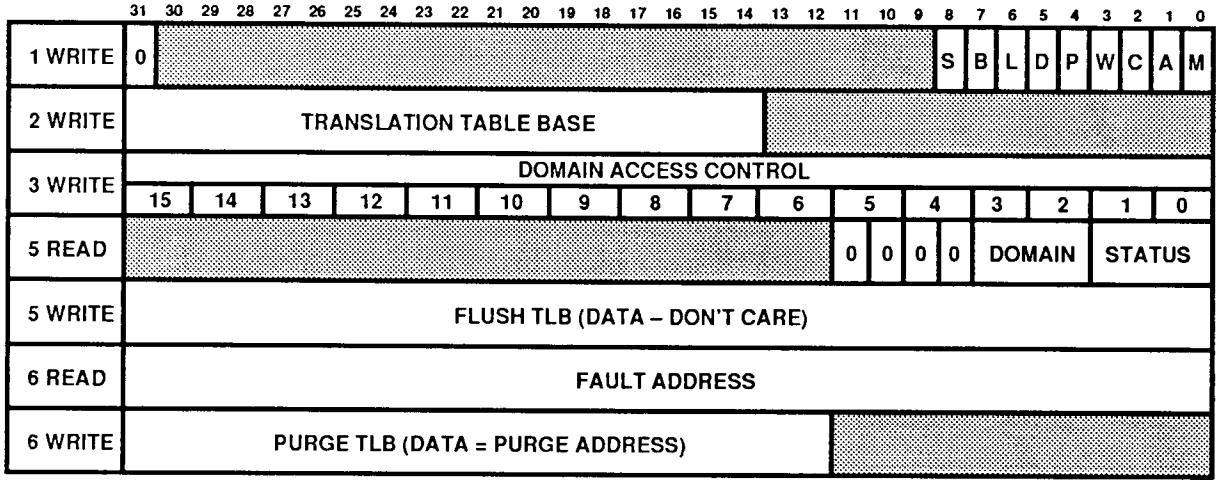
There are three routes by which the address translation (and hence permission check) takes place. The route taken depends on whether the address in question has been marked as a section-mapped access or a page-mapped access; and there are two sizes of page-mapped access (large pages and small pages). However, the translation process always starts out in the same way, as described below, with a Level One fetch. A section-mapped access only requires a Level One fetch, but a page-mapped access also requires a Level Two fetch.

TRANSLATION PROCESS

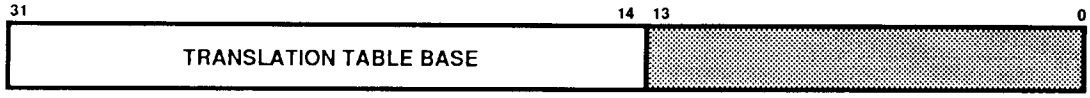
Translation Table Base

The translation process is initiated when the on-chip TLB does not contain an entry for the requested virtual address. The Translation Table Base (TTB) Register points to the base of a table in physical memory which contains Section and/or Page descriptors. The 14 low-order bits of the TTB Register are set to zero as illustrated below; the table must reside on a 16 Kb boundary.

MMU REGISTER SUMMARY



TRANSLATION TABLE BASE REGISTER



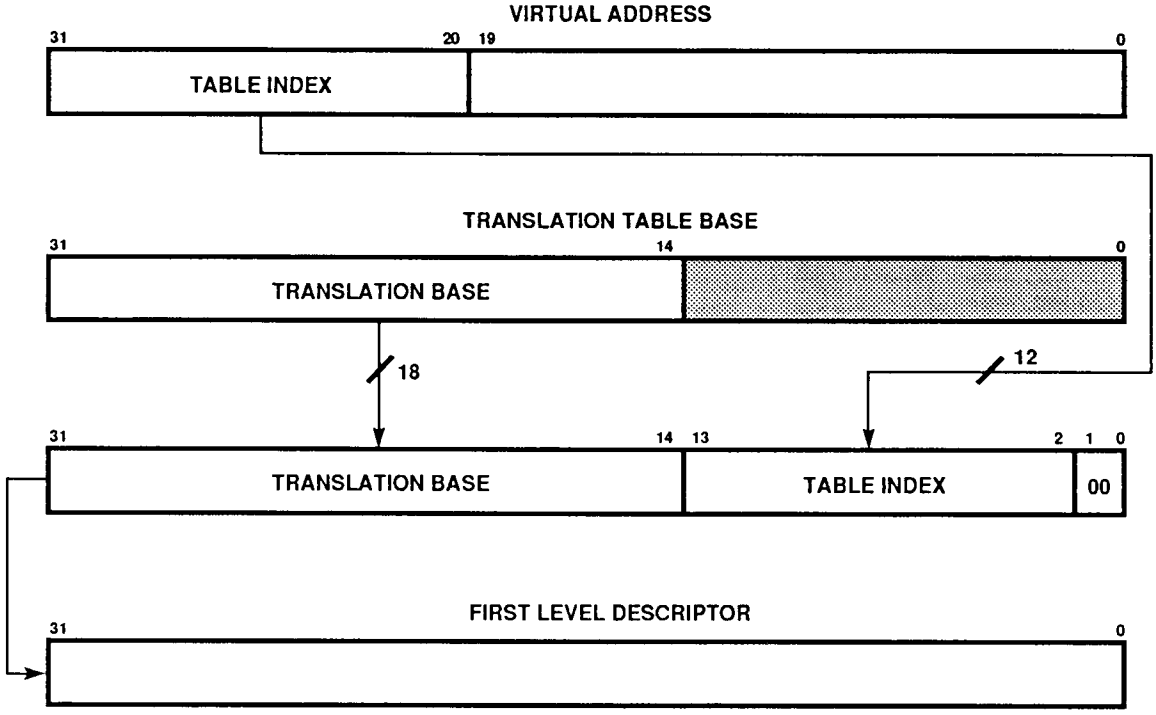
Level One Fetch

Bits 31:14 of the Translation Table Base register are concatenated with bits 31:20 of the virtual address to produce a 30-bit address as illustrated directly below. This address selects a four-byte translation table entry which is a First Level Descriptor for either a Section or a Page (bit 1 of the descriptor returned specifies whether it is for a Section or Page).

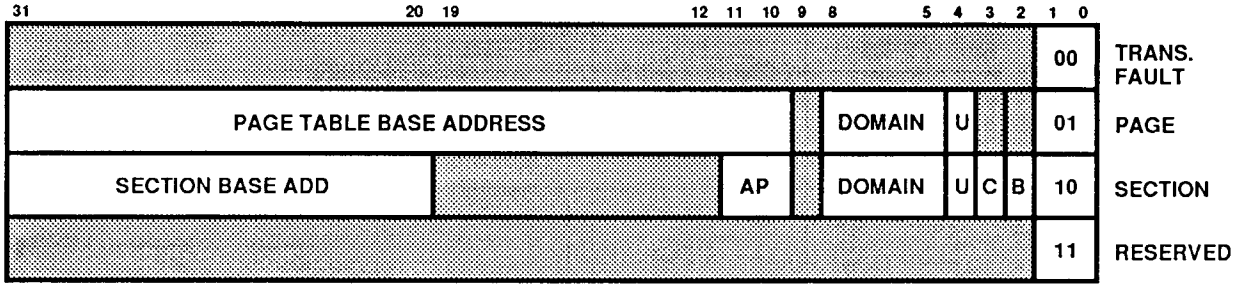
Level One Descriptor

The Level One Descriptor returned is either a Page Table Descriptor or a Section Descriptor, and its format varies accordingly. (The format of Level One Descriptors is illustrated in the figure below.)

ACCESSING THE TRANSLATION TABLE FIRST LEVEL DESCRIPTORS



LEVEL ONE DESCRIPTORS



The two least significant bits indicate the descriptor type and validity, and are interpreted as shown in the table.

Page Table Descriptor

- Bits 3:2 are always written as 0.
- Bit 4 Updateable: indicates that the data in the cache should be updated during a write operation to maintain consistency with external memory (if the cache is enabled).
- Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.
- Bits 31:10 form the base for referencing the Page Table Entry. (The page table index for the entry is derived from the virtual address as illustrated on page 54).

If a Page Table Descriptor is returned from the Level One fetch, a Level Two fetch is initiated as described below.

Section Descriptor

Bits 4:2 (U,C, & B) control the cache- and write-buffer-related functions as follows:

U – Updateable: indicates that the data in the cache should be updated during a write operation to maintain consistency with external memory (if the cache is enabled).

C – Cacheable: indicates that data at this address will be placed in the cache (if the cache is enabled).

B – Bufferable: indicates that data at this address will be written through the write buffer (if the write buffer is enabled).

- Bits 8:5 specify one of the sixteen possible domains (held in the Domain Access Control Register) that contain the primary access controls.
- Bits 11:10 (AP) specify the access permissions for this section and are interpreted as shown in table directly above. Their interpretation is dependent upon the setting of the S bit (Control Register bit 8). Note that the Domain Access Control specifies the primary access control; the AP bits only have an effect in client mode. Refer to section on access permissions.
- Bits 19:12 are always written as 0.
- Bits 31:20 form the corresponding bits of the physical address for the 1 MByte section.

INTERPRETING LEVEL ONE DESCRIPTOR BITS [1:0]

Value	Meaning	Notes
00	Invalid	Generates a Section Translation Fault
01	Page	Indicates that this is a Page Descriptor
10	Section	Indicates that this is a Section Descriptor
11	Reserved	Reserved for future use (currently as for invalid)

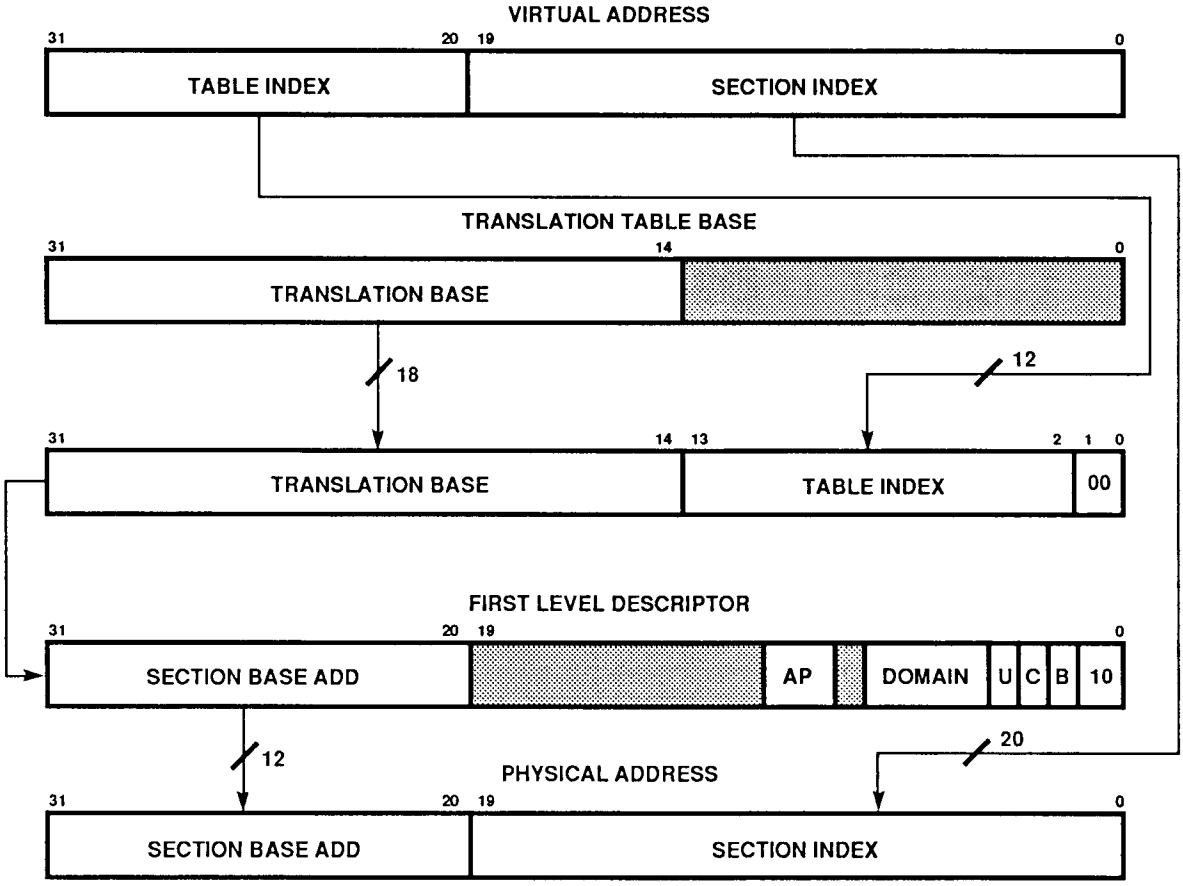
INTERPRETING ACCESS PERMISSION (AP) BITS

AP	S	Permissions		Notes
		Supervisor	User	
00	0	No Access	No Access	Any access generates a permission fault.
00	1	Read Only	No Access	
01	x	Read/Write	No Access	Access allowed only in Supervisor mode.
10	x	Read/Write	Read Only	Writes in User mode cause permission fault.
11	x	Read/Write	Read/Write	All access types permitted in both modes.

Translating Section References

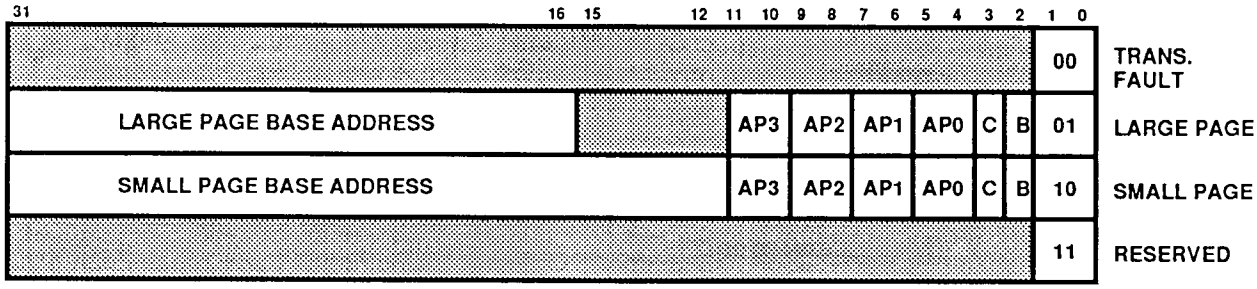
The figure below illustrates the complete Section translation sequence. Note that the access permissions contained in the Level One descriptor must be checked before the physical address is generated. The sequence for checking access permissions is described below.

SECTION TRANSLATION



Level Two Descriptor
If the Level One fetch returns a Page Table Descriptor, this provides the base address of the page table to be used. The page table is then accessed as shown on page 54, and a Page Table Entry, or Level Two Descriptor, is returned. This in turn may define either a Small Page or a Large Page access. The illustratation below shows the format of Level Two Descriptors.

PAGE TABLE ENTRY (LEVEL TWO DESCRIPTOR)



The two least significant bits indicate the page size and validity, and are interpreted as shown in the table.

- Bit 2 C – Cacheable: indicates that data at this address will be placed in the IDC (if the cache is enabled).
- Bit 3 B – Bufferable: indicates that data at this address will be written through the Write Buffer (if the Write Buffer is enabled).
- Bits 11:4 specify the access permissions (AP3 – AP0) for the four subpages and interpretation of these bits is described earlier in the table located on page 51.
- For large pages, bits 15:12 are programmed as 0.
- Bits 31:12 (small pages) or bits 31:16 (large pages) are used to form the corresponding bits of the physical address – the physical page number. (The page index is derived from the virtual address as illustrated on pages 54 and 55).

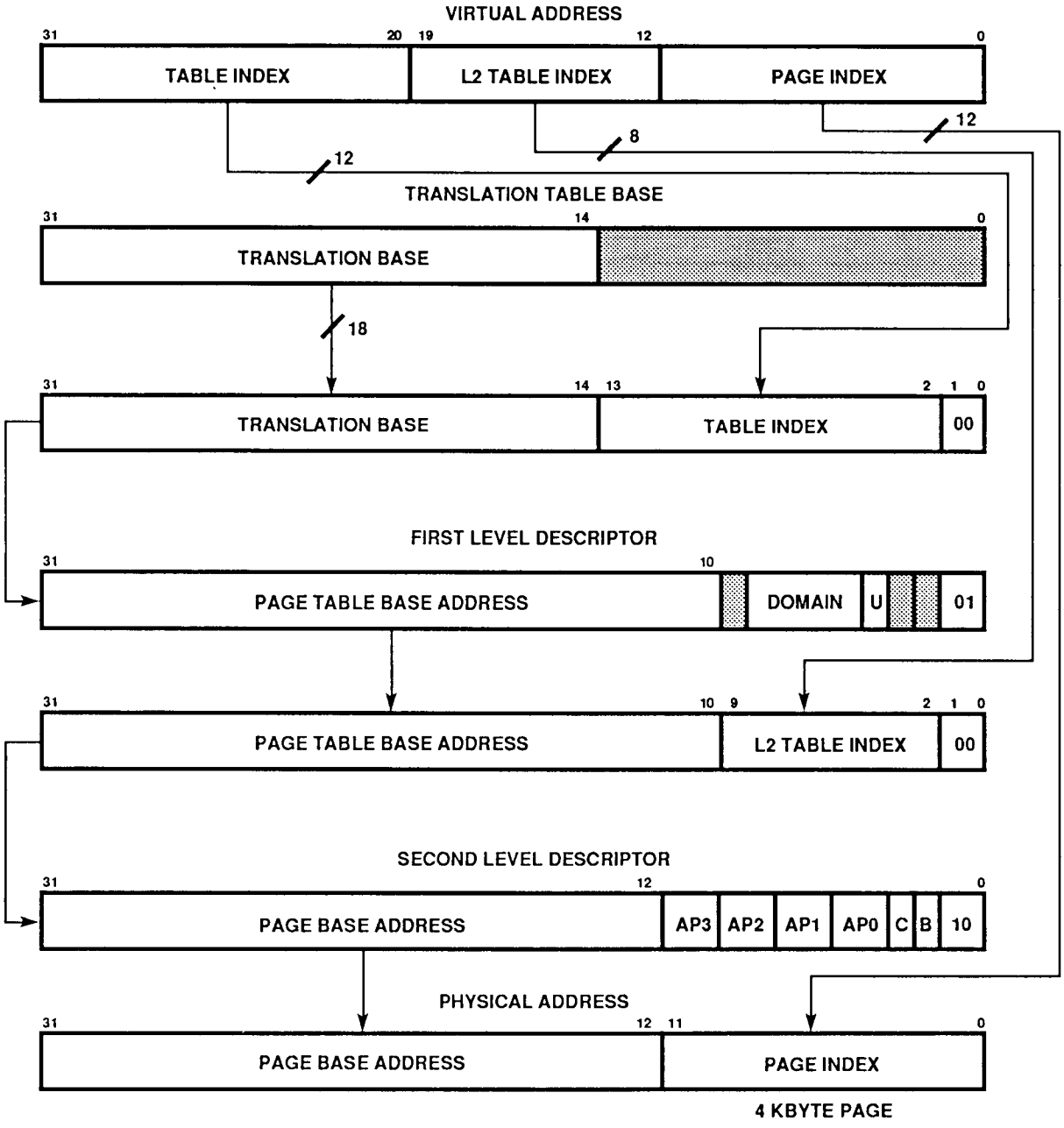
INTERPRETING PAGE TABLE ENTRY BITS 1:0

Value	Meaning	Notes
00	Invalid	Generates a Page Translation Fault
01	Large Page	Indicates that this is a 64 Kb Page
10	Small Page	Indicates that this is a 4Kb Page
11	Reserved	Reserved for future use (currently as for invalid)

Translating Small Page References

The illustration below shows the complete translation sequence for a 4 Kb Small Page. Page translation involves one additional step beyond that of a section translation: the Level One descriptor is the Page Table descriptor, and this is used to point to the Level Two descriptor, of Page Table Entry. (Note that the access permissions are now contained in the Level Two descriptor and must be checked before the physical address is generated. The sequence for checking access permissions is described on page 57).

SMALL PAGE TRANSLATION



Translating Large Page References
The figure below illustrates the complete translation sequence for a 64Kb Large Page. Note that since the upper four bits of the Page Index and low-order four bits of the Page Table index of overlap, each Page Table Entry for a Large Page must be duplicated 16 times (in consecutive memory locations) in the Page Table.

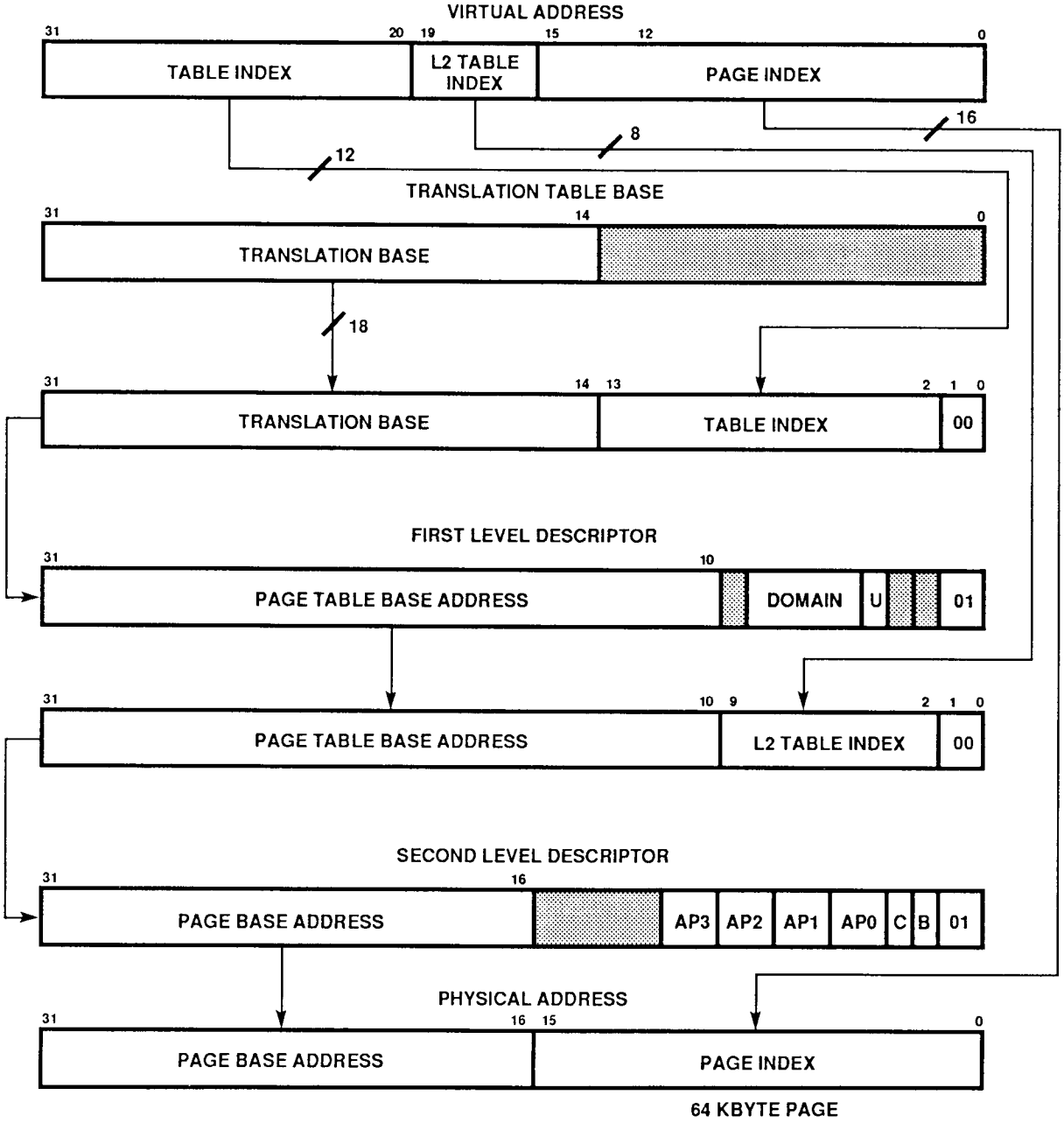
MMU Faults and CPU Aborts
The MMU generates four types of faults:

- Alignment Fault
- Translation Fault
- Domain Fault
- Permission Fault

In addition, an external abort may be raised on external data access.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as the result of a memory access, the MMU will abort the access and signal the fault condition to the CPU. The MMU is also capable of retaining status and address information about the abort. The CPU recognizes two types of abort: data aborts and prefetch aborts, and these are treated differently by the MMU.

LARGE PAGE TRANSLATION



If the MMU detects an access violation, it will do so before the external memory access takes place, and it will therefore inhibit the access. External aborts will not necessarily inhibit the external access, as described in the section on external aborts.

Fault Address & Fault Status Registers (FAR & FSR)

Aborts resulting from data accesses (data aborts) are acted upon by the CPU immediately, and the MMU places an encoded 4-bit value FS[3:0], along with the 4-bit encoded domain number in the Fault Status Register (FSR). In addition the virtual processor address which caused the data abort is latched into the Fault Address Register (FAR). If an access violation simultaneously generates more than one source of abort, they are encoded in the priority given in the table below.

CPU instructions on the other hand are prefetched, so prefetch aborts simply flag the instruction as it enters the instruction pipeline. Only when the instruction is executed does it cause an abort; an abort is not acted upon if the instruction is not used (i.e. it is branched around). As a result, because instruction prefetch aborts may or may not be acted upon, the MMU status information is not preserved for the resulting CPU abort; for a prefetch abort, the MMU does not update the FSR or FAR.

The sections that follow describe the various access permissions and controls supported by the MMU and detail how these are interpreted to generate faults.

PRIORITY ENCODING OF FAULT STRESS, (Note 1)

	Source		FS [3210]	Domain [3:0]	FAR	Note
Highest	Write Buffer		00x0	x		2
	Bus Error	LF Section	0100	valid	valid	3
		Page	0110	valid	valid	
	Bus Error	Section	1000	valid	valid	
		Page	1010	valid	valid	
	Alignment		00x1	x	valid	
	Bus Error Trans	L1	1100	x	valid	
		L2	1110	valid	valid	
	Translation	Section	0101	see note	valid	4
		Page	0111	valid	valid	
	Domain	Section	1001	valid	valid	
		Page	1011	valid	valid	
	Permission	Section	1101	valid	valid	
Lowest		Page	1111	valid	valid	

x is undefined: may read as 0 or 1

Notes:

1. Any abort masked by the priority encoding may be regenerated by fixing the primary abort and restarting the instruction.
2. The Write Buffer Bus Error is asynchronous and the Fault Address Register reflects the first data operation that could be aborted. This instruction must be restarted, (and it could of course generate an abort of its own).
3. This assumes that the error was flagged on word 0 of the linefetch.
4. In fact this register will contain bits[8:5] of the Level 1 entry which are undefined, but would encode the domain in a valid entry.

Access Control

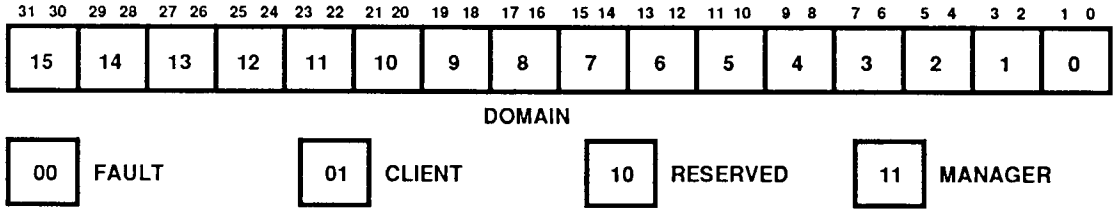
MMU accesses are primarily controlled via domains. There are 16 domains, and each has a 2-bit field to define it. Two basic kinds of users are supported: Clients and Managers. Clients use a domain; Managers control the behavior of the domain. The domains are defined in the Domain Access Control Register. The figure below illustrates how the 32 bits of the register are allocated to define the sixteen 2-bit domains.

The table defines how the bits within each domain are interpreted to specify the access permissions.

INTERPRETING ACCESS BITS IN DOMAIN ACCESS CONTROL REGISTER

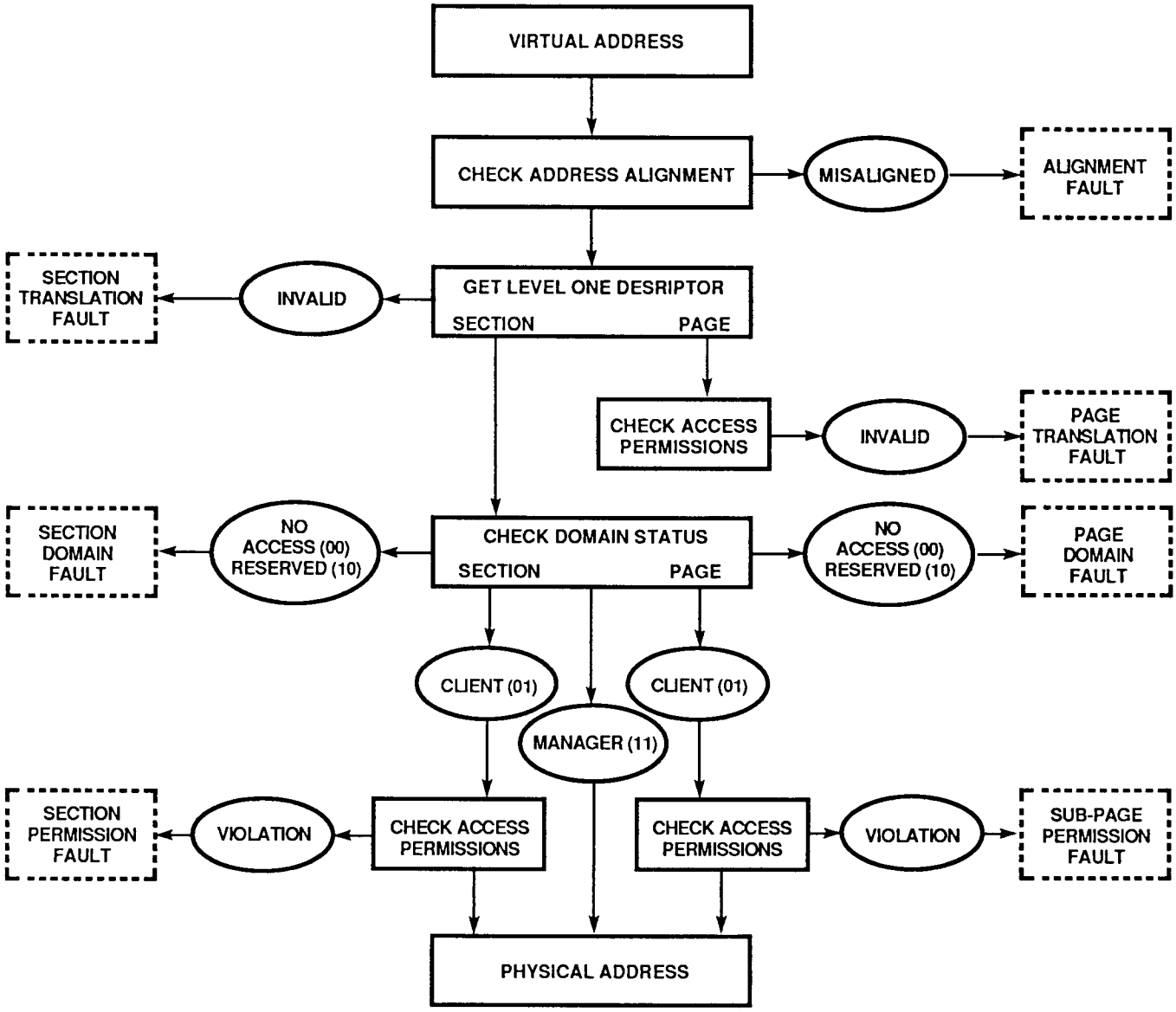
Value	Meaning	Notes
00	No Access	Any access will generate a Domain Fault.
01	Client	Accesses are checked against the access permission.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are NOT checked against the access Permission bits so a Permission fault cannot be generated

DOMAIN ACCESS CONTROL REGISTER FORMAT



FAULT CHECKING SEQUENCE

The sequence by which the MMU checks for access faults is slightly different for Sections and Pages. The figure below illustrates the sequence for both types of accesses. The sections and figures that follow describe the conditions that generate each of the faults.



SEQUENCE FOR CHECKING FAULTS

Alignment Fault

If Alignment fault is enabled (bit 1 in Control Register set), the MMU will generate an alignment fault on any data word access the address of which is not word-aligned irrespective of whether the MMU is enabled or not; in other words, if either of virtual address bits [1:0] are not 0. Alignment fault will not be generated on any instruction fetch, nor on any byte access. Note that if the access generates an alignment fault, the access sequence will abort without reference to further permission checks.

Translation Fault

There are two types of translation fault: section and page.

- A Section Translation Fault is generated if the Level One descriptor is marked as invalid. This happens if bits[1:0] of the descriptor are both 0 or both 1.
- A Page Translation Fault is generated if the Page Table Entry is marked as invalid. This happens if bits[1:0] of the entry are both 0 or both 1.

Domain Fault

There are two types of domain fault: section and page.

In both cases the Level One descriptor holds the 4-bit Domain field which selects one of the 16 two-bit domains in the Domain Access Control Register. The two bits of the specified domain are then checked for access permissions as detailed in the table on page 57. In the case of a section, the domain is checked once the Level One descriptor is returned, and in the case of a page, the domain is checked once the Page Table Entry is returned.

If the specified access is either No Access (00) or Reserved (10) then either a Section Domain Fault or Page Domain Fault occurs.

Permission Fault

There are two types of permission fault: section and sub-page. Permission fault is checked at the same time as domain fault. If the two-bit domain field returns client (01), then the permission access check is invoked as follows:

SECTION

If the Level One descriptor defines a section-mapped access, then the AP bits of the descriptor define whether or not the access is allowed according to the table below. Their interpretation is dependent upon the settling of the S bit (Control Register bit 8). If the access is not allowed, then a Section Permission fault is generated.

SUB-PAGE

If the Level One descriptor defines a page-mapped access, then the Level Two descriptor specifies four access permission fields (AP3...AP0) each corresponding to one quarter of the page. Hence for small pages, AP3 is selected by the top 1 Kb of the page, and AP0 is selected by the bottom 1 Kb of the page; for large pages, AP3 is selected by the top 4 Kb of the page, and AP0 is selected by the bottom 4 Kb of the page. The selected AP bits are then interpreted in exactly the same way as for a section (see table below), the only difference being that the fault generated is a sub-page permission fault.

INTERPRETING ACCESS PERMISSION (AP) BITS

AP	S	Permissions		Notes
		Supervisor	User	
00	0	No Access	No Access	Any access generates a permission fault.
00	1	Read Only	No Access	Supervisor read only permitted.
01	x	Read/Write	No Access	Supervisor read or write only permitted.
10	x	Read/Write	Read Only	Writes in User mode cause permission fault.
11	x	Read/Write	Read/Write	All access types permitted in both modes.

EXTERNAL ABORTS

In addition to the MMU-generated aborts, ARM600 has an external abort pin which may be used to flag an error on an external memory access. However, some accesses aborted in this way are not restartable, so this pin must be used with great care. The following section describes the restrictions.

- Uncacheable reads
- Unbuffered writes
- Level one descriptor fetch
- Level two descriptor fetch
- Read-lock-write sequence

These accesses may be aborted and restarted safely. If any of the above are aborted the external access will cease on the following cycle. In the case of a read-lock-write sequence in which the read aborts, the write will not happen.

Cacheable reads (linefetches)

A linefetch may be aborted safely provided that the abort is flagged on word 0. In this case, the IDC will not be updated or corrupted, and the access will be restartable. It is not advisable to flag an abort on any word other than word 0 of a linefetch, as the IDC will contain a corrupt line, and the instruction may not be restartable. Externally, a linefetch which is externally aborted will continue to the end as though it had not aborted.

Buffered writes

Buffered writes cannot be safely externally aborted. Because the processor will have moved on before the external abort is received, this class of abort is not restartable. If the system does flag this type of abort, then the Fault Status Register will record the fact, but this is a non-recoverable error, and the machine must be reset. Therefore, the system should be configured such that it does not do buffered writes to areas of memory which are capable of flagging an external abort. If a buffered write burst is externally aborted, then the external write will continue to the end.

INTERACTION OF THE MMU, IDC AND WRITE BUFFER

The MMU, IDC and WB may be enabled/disabled independently. However there are only five valid combinations. There are no hardware interlocks on these restrictions, so invalid combinations will cause undefined results.

MMU	IDC	WB
off	off	off
on	off	off
on	on	off
on	off	on
on	on	on

The following procedures must be observed.

To Enable the MMU

Program the Translation Table Base and Domain Access Control Registers

Program Level 1 and Level 2 page tables as required

Enable the MMU by setting bit 0 in the Control Register. (Note 1)

To Disable the MMU

Disable the WB by clearing bit 3 in the Control Register.

Disable the IDC by clearing bit 2 in the Control Register.

Disable the MMU by clearing bit 0 in the Control Register. (Note 2)

ARM600 BUS INTERFACE

OVERVIEW

The ARM600 bus interface has two distinct modes of operation: synchronous and asynchronous. The mode is configured by tying the SNA pin either HIGH or LOW. The memory interface is always clocked by MCLK, and the core is clocked by a combination of FCLK and MCLK. The two modes differ in the relationship between FCLK and MCLK: in asynchronous mode, (SNA tied LOW), the two clocks may be completely asynchronous of unrelated frequency; in synchronous mode (SNA tied HIGH), MCLK may only make transitions on the falling edge of FCLK. In systems where such a relationship between the two clocks exist, the synchronization penalty can be removed by selecting synchronous operation.

ASYNCHRONOUS MODE

In this mode, FCLK and MCLK may be completely asynchronous. This mode should be selected when the two clocks are of unrelated frequency. There is a synchronization penalty whenever the internal core clock switches between the two input clocks. This penalty is symmetric, and varies between nothing and a whole period of the clock to which the core is resynchronizing. Thus on average when changing from FCLK to MCLK it is half an MCLK period, and when changing from MCLK to FCLK it is half an FCLK period.

SYNCHRONOUS MODE

In this mode, there is a tightly defined relationship between FCLK and MCLK. MCLK may only make transitions on the falling edge of FCLK. An amount of jitter between the two clocks is permitted, and the device will function correctly, but MCLK must not be later than FCLK. MCLK may lead FCLK by up to the difference in MCLK and FCLK periods without affecting the critical paths.

Notes:

1. Care must be taken if the translated address differs from the untranslated address as the two instructions following the enabling of the MMU will have been fetched using "flat translation" and enabling the MMU may be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. Consider the following code sequence:

```
MOV R1, #&1
MCR 15,0,R1,0,0; Enable MMU
Fetch Flat
Fetch Flat
Fetch Translated
```

2. If the MMU is enabled, then disabled and subsequently re-enabled the contents of the TLB will have been reserved. If these are now invalid, the TLB should be flushed before re-enabling the MMU. Disabling of all three functions may be done simultaneously.

CYCLE SPEED

The speed of the memory bus interface may be controlled in two ways:

- The LOW and HIGH phases of MCLK may be stretched.
- The NWAIT pin can be used to insert entire MCLK cycles into any access. When LOW this signal inserts an extra MCLK cycle into the current access. It must change while MCLK is LOW.

CYCLE TYPES

ARM600 can perform many different bus cycles, and they may be combined in any order.

- Idle
- Line Fetch
- Unbuffered Write
- Uncacheable Read
- Buffered Write
- Translation Level 1 Fetch
- Translation Level 2 Fetch
- Read-Lock-Write

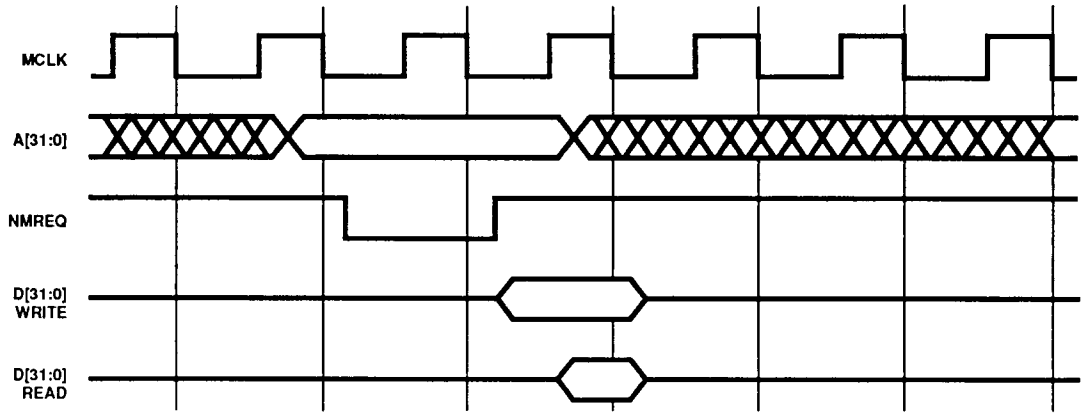
Linefetch, unbuffered writes, uncacheable reads and buffered writes all have very similar bus cycles. They differ only in the number of idle cycles at the end, which in turn depends on what cycle follows. The following table defines this for each bus cycle. The diagrams give examples of typical sequences.

CYCLE TYPE DETAILS

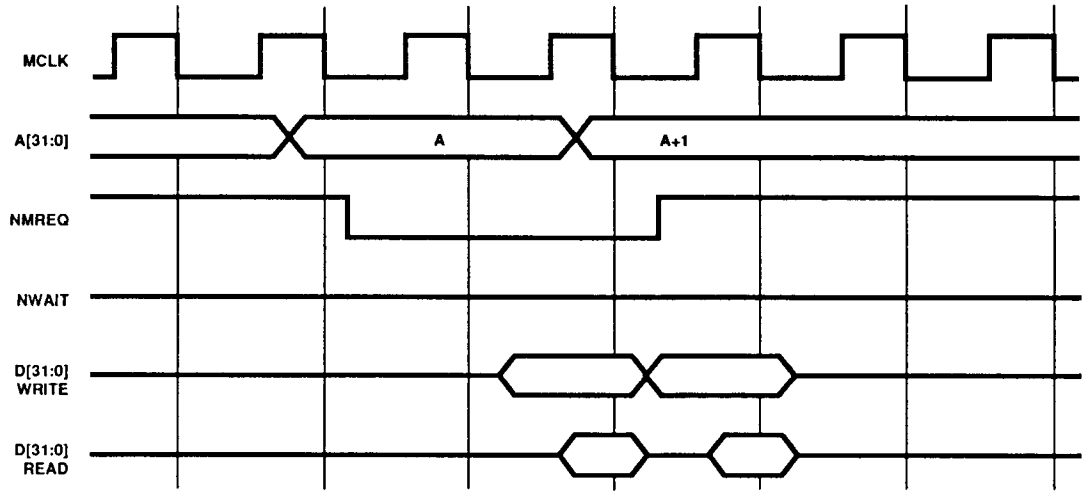
		A[31:0]	nRW	nMREQ	D[31:0]
	IDLE	OLD	OLD	1	
	LINEFETCH	R	A	I	
		R	A	M	
		R	A+4	M	D
		R	A+8	M	D
		R	A+C	M	D
		R	A+C	I	D
UNCACHEABLE READ/ UNBUFFERED WRITE	ONE WORD	R/W	A	I	
		R/W	A	M	D
		R/W	A	I	
	MANY WORD START	R/W	A	I	
		R/W	A	M	
					D
	REPEAT	R/W	A+4	M	
					D
	END	R/W	A+4	M	
		R/W	A+4	I	D
	BUFFERED WRITE	W	A	I	
		R	A	M	
					D
	MORE	W	A	M	
					D
	LEVEL ONE OR TWO	R	A	I	
		R	A	M	
		R	A	I	D
READ-LOCKED-WRITE	READ PHASE	R	aL	I	
		R	aL	M	
		R	aL	I	D
	WRITE PHASE UNBUFFERED	W	aL	I	
		W	aL	M	
		W	aL	I	D
	WRITE PHASE BUFFERED	W	aL	I	
		W	aL	M	
					D
	WRITE PHASE ABORTED	W	aL	I	
		W	aL	I	

EXAMPLE CYCLES

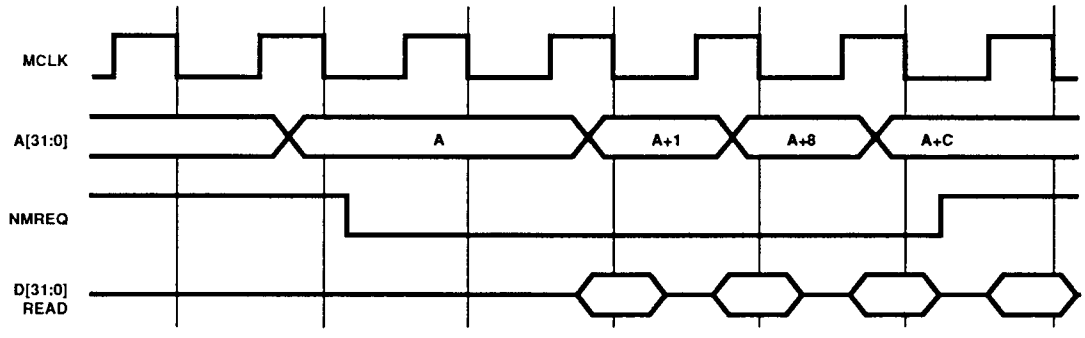
ONE WORD READ OR WRITE



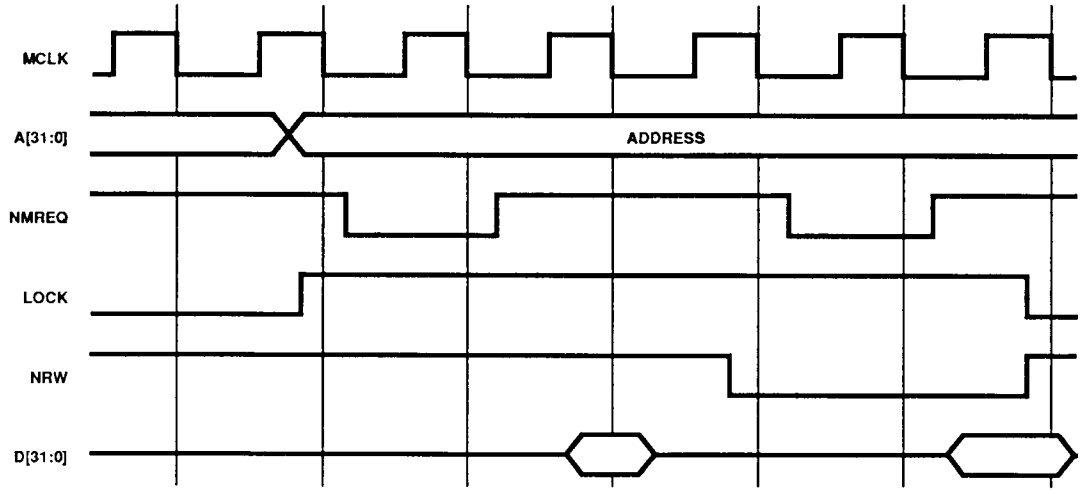
TWO WORD WRITE OR READ



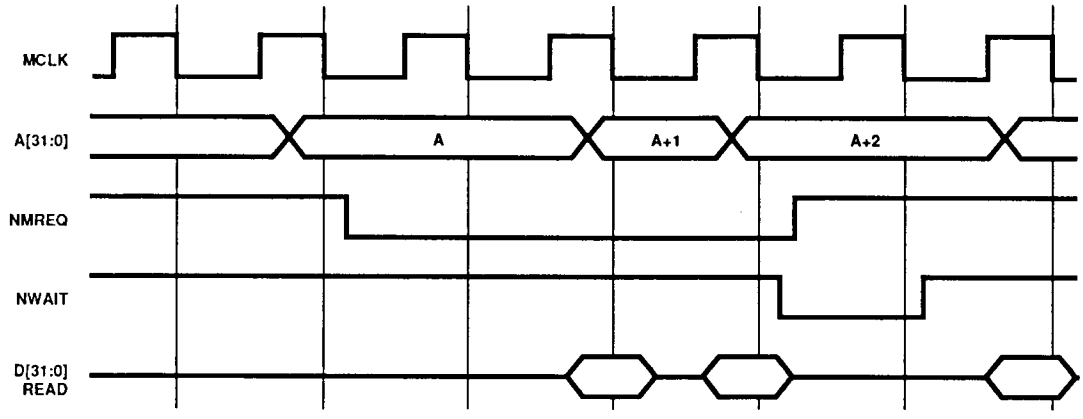
LINEFETCH



READ-LOCKED-WRITE



USE OF NWAIT PIN TO STRETCH A CYCLE



In this example the NWAIT pin has been used to introduce an extra MCLK cycle to ease the system timing. This allows all control signals to be sampled on the rising edges of MCLK. More sophisticated systems may sample control signals on both rising and falling edges of MCLK to improve performance.

BOUNDARY SCAN TEST INTERFACE

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 section and for a description of the TAP controller states).

INSTRUCTION REGISTER

The instruction Register is 4 bits in length. There is no parity bit.

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

PUBLIC INSTRUCTIONS

The following public instructions are supported:

Instruction	Binary Code
BYPASS	1111
SAMPLE/PRELOAD	0011
EXTEST	0000
INTEST	1100
IDCODE	1110
HI-Z	0111
CLAMP	0101
CLAMPZ	1001

In the descriptions that follow, TDI and TMS are sampled on the rising edge of TCK and all output transitions on TDO occur as a result of the falling edge of TCK.

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.

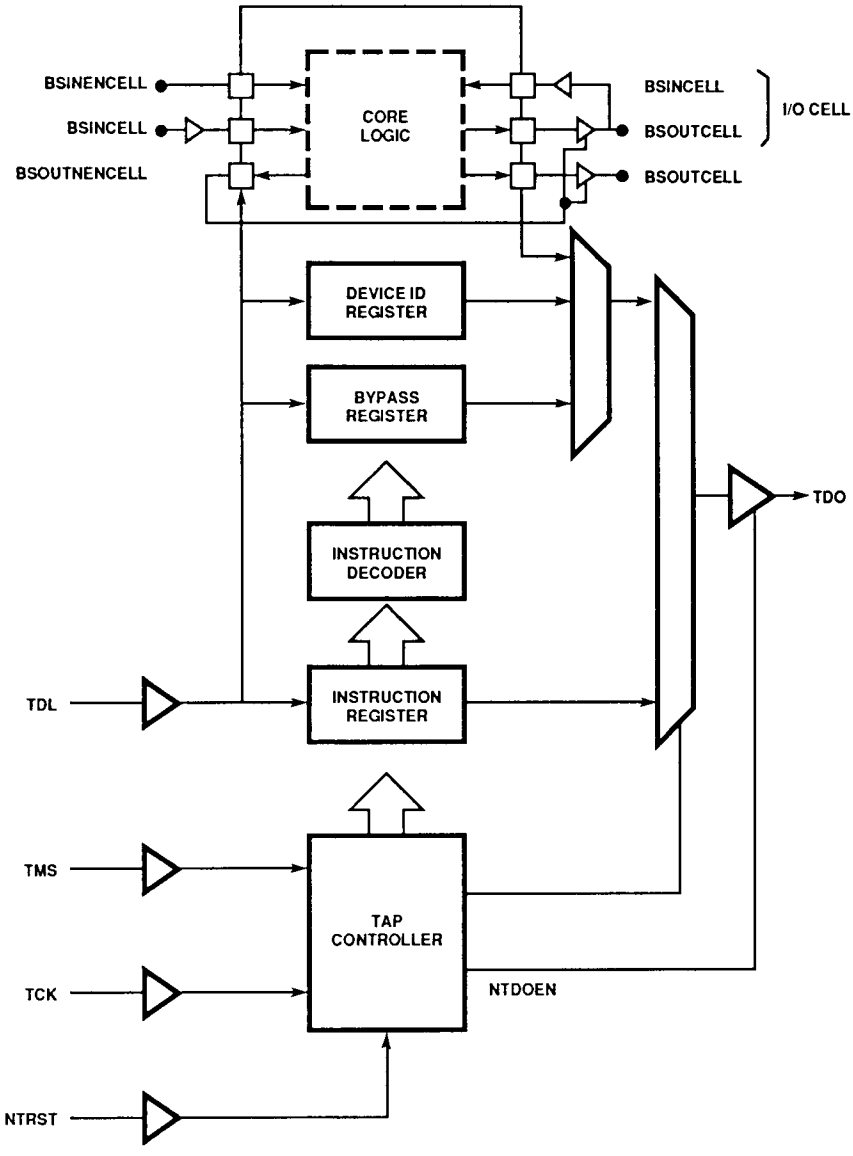
SAMPLE/PRELOAD (0011)

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.

BLOCK DIAGRAM



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, while 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 or EXTEST instructions (see the table below for appropriate guard values to be used for each boundary-scan cell).

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. The first EXTEST 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.

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, while 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.

Single-step operation is possible using the INTEST instruction.

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 at the end of this section) 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, while data is shifted in via the TDI pin into the ID register. In the *UPDATE-DR* state, the ID register is unaffected.

HI-Z (0111)
The HI-Z instruction connects a 1 bit shift register (the BYPASS register) between TDI and TDO.

When the HI-Z 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.

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 for this device at the end of this section) 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.

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 derived from the boundary-scan cells. 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.

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.

TEST DATA REGISTERS

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.

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

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 32 bit device identification code is loaded into the ID register from its parallel inputs during the *CAPTURE-DR* state.

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.

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 TEST mode (i.e. when either EXTEST or INTEST is the currently selected instruction), 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 3-stateable buses.

ARM600 DEVICE IDENTIFICATION REGISTER

MANUFACTURER'S IDENTIFICATION CODE:

000 0010 1011

PART NUMBER CODE:

0000 0000 0100 0011

VERSION CODE:

0000

No programmable supplementary identification code is provided.

The complete 32-bit device identification code for ARM600 is:

ARM600 DEVICE IDENTIFICAITON REGISTER

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1	0
VERSION	PART NUMBER	MANUFACTURER IDENTITY	
0000	0000 0000 0100 0011	0000 0100 001	1

IN HEXADECIMAL: 00043057

ARM600 BOUNDARY SCAN REGISTER

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 3-state outputs. All boundary-scan register cells at input pins can apply tests to the on-chip system logic. The EXTEST guard values specified in the table below should be clocked into the boundary-scan register (using the SAMPLE/PRELOAD instruction) before the EXTEST instruction is 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. These guard values should also be used when new EXTEST or INTEST ventors are clocked into the boundary-scan register.

BOUNDARY SCAN TABLE

No.	Cell Name	Pin	Type	Output Enable BS cell	INTEST Guard Value	EXTEST/CLAMP Guard Value
(From TDI)						
1	DIN10	D[10]	IN	–		
2	DOUT10	D[10]	OUT	NENDOUT		
3	DIN11	D[11]	IN	–		
4	DOUT11	D[11]	OUT	NENDOUT		
5	DIN12	D[12]	IN	–		
6	DOUT12	D[12]	OUT	NENDOUT		
7	DIN13	D[13]	IN	–		
8	DOUT13	D[13]	OUT	NENDOUT		
9	DIN14	D[14]	IN	–		
10	DOUT14	D[14]	OUT	NENDOUT		
11	DIN15	D[15]	IN	–		
12	DOUT15	D[15]	OUT	NENDOUT		
13	DIN16	D[16]	IN	–		
14	DOUT16	D[16]	OUT	NENDOUT		
15	DIN17	D[17]	IN	–		
16	DOUT17	D[17]	OUT	NENDOUT		
17	DIN18	D[18]	IN	–		
18	DOUT18	D[18]	OUT	NENDOUT		
19	NENDOUT	–	OUTEN0	–	1	
20	DIN19	D[19]	IN	–		
21	DOUT19	D[19]	OUT	NENDOUT		
22	DIN20	D[20]	IN	–		
23	DOUT20	D[20]	OUT	NENDOUT		
24	DIN21	D[21]	IN	–		
25	DOUT21	D[21]	OUT	NENDOUT		
26	DIN22	D[22]	IN	–		
27	DOUT22	D[22]	OUT	NENDOUT		
28	DIN23	D[23]	IN	–		
29	DOUT23	D[23]	OUT	NENDOUT		
30	DIN24	D[24]	IN	–		

BOUNDARY SCAN TABLE (Cont.)

No.	Cell Name	Pin	Type	Output Enable BS cell	INTEST Guard Value	EXTEST/CLAMP Guard Value
31	DOUT24	D[24]	OUT	NENDOUT		
32	DIN25	D[25]	IN	–		
33	DOUT25	D[25]	OUT	NENDOUT		
34	DIN26	D[26]	IN	–		
35	DOUT26	D[26]	OUT	NENDOUT		
36	DIN27	D[27]	IN	–		
37	DOUT27	D[27]	OUT	NENDOUT		
38	DIN28	D[28]	IN	–		
39	DOUT28	D[28]	OUT	NENDOUT		
40	DIN29	D[29]	IN	–		
41	DOUT29	D[29]	OUT	NENDOUT		
42	DIN30	D[30]	IN	–		
43	DOUT30	D[30]	OUT	NENDOUT		
44	DIN31	D[31]	IN	–		
45	DOUT32	D[32]	OUT	NENDOUT		
46	DBE	DBE	IN	–		
47	SEQ	SEQ	OUT	NMSE		
48	NMREQ	NMREQ	OUT	NMSE		
49	NMSE	MSE	INEN1	–	0	
50	CPDBE	CPDBE	IN	–		
51	NIRQ	NIRQ	IN	–		
52	NFIQ	NFIQ	IN	–		
53	A00	A[0]	OUT	NABE		
54	A01	A[1]	OUT	NABE		
55	A02	A[2]	OUT	NABE		
56	A03	A[3]	OUT	NABE		
57	A04	A[4]	OUT	NABE		
58	A05	A[5]	OUT	NABE		
59	A06	A[6]	OUT	NABE		
60	A07	A[7]	OUT	NABE		
61	A08	A[8]	OUT	NABE		
62	A09	A[9]	OUT	NABE		
53	A10	A[10]	OUT	NABE		
64	A11	A[11]	OUT	NABE		
65	A12	A[12]	OUT	NABE		
66	A13	A[13]	OUT	NABE		
67	A14	A[14]	OUT	NABE		
68	A15	A[15]	OUT	NABE		
69	A16	A[16]	OUT	NABE		
70	A17	A[17]	OUT	NABE		
71	A18	A[18]	OUT	NABE		

BOUNDARY SCAN TABLE (Cont.)

No.	Cell Name	Pin	Type	Output Enable BS cell	INTEST Guard Value	EXTEST/CLAMP Guard Value
72	A19	A[19]	OUT	NABE		
73	A20	A[20]	OUT	NABE		
74	A21	A[21]	OUT	NABE		
75	A22	A[22]	OUT	NABE		
76	A23	A[23]	OUT	NABE		
77	A24	A[24]	OUT	NABE		
78	A25	A[25]	OUT	NABE		
79	A26	A[26]	OUT	NABE		
80	A27	A[27]	OUT	NABE		
81	A28	A[28]	OUT	NABE		
82	A29	A[29]	OUT	NABE		
83	A30	A[30]	OUT	NABE		
84	A31	A[31]	OUT	NABE		
85	NABE	ABE	INEN1	–	0	
86	NRW	NRW	OUT	NCBE		
87	NBW	NBW	OUT	NCBE		
88	RLW	LOCK	OUT	NCBE		
89	NCBE	CBE	INEN1	–	0	
90	NRESET	NRESET	IN	–		
91	SNA	SNA	IN	–		
92	NWAIT	NWAIT	IN	–		
93	ABORT	ABORT	IN	–		
94	MCLK	MCLK	IN	–		0
95	FCLK	FCLK	IN	–		0
96	NTEST	NTEST	IN	–		1
97	NCPWTIN	NCPWT	IN	–		
98	NCPWTOUT	NCPWT	OUT	NCPE		
99	CPCLKIN	CPCLK	IN	–		
100	CPCLKOUT	CPCLK	OUT	NCPE		
101	NCPE	–	OUTEN0	–	1	
102	CPSPV	CPSPV	OUT	NCPE		
103	NCPI	NCPI	OUT	NCPE		
104	NOPC	NOPC	OUT	NCPE		
105	CPE	CPE	IN	–		
106	CPA	CPA	IN	–		
107	CPB	CPB	IN	–		
108	CPDIN31	CPD[31]	IN	–		
109	CPDOUT31	CPD[31]	OUT	NENCPDHI		
110	CPDIN30	CPD[30]	IN	–		
111	CPDOUT30	CPD[30]	OUT	NENCPDHI		
112	CPDIN29	CPD[29]	IN	–		
113	CPDOUT29	CPD[29]	OUT	NENCPDHI		

BOUNDARY SCAN TABLE (Cont.)

No.	Cell Name	Pin	Type	Output Enable BS cell	INTEST Guard Value	EXTEST/CLAMP Guard Value
114	CPDIN28	CPD[28]	IN	–		
115	CPDOUT28	CPD[28]	OUT	NENCPDHI		
116	CPDIN27	CPD[27]	IN	–		
117	CPDOUT27	CPD[27]	OUT	NENCPDHI		
118	CPDIN26	CPD[26]	IN	–		
119	CPDOUT26	CPD[26]	OUT	NENCPDHI		
120	NENCPDHI	–	OUTEN0	–	1	
121	CPDOUT25	CPD[25]	OUT	NENCPDHI		
122	CPDIN25	CPD[25]	IN	–		
123	CPDOUT24	CPD[24]	OUT	NENCPDHI		
124	CPDIN24	CPD[24]	IN	–		
125	CPDOUT23	CPD[23]	OUT	NENCPDHI		
126	CPDIN23	CPD[23]	IN	–		
127	CPDOUT22	CPD[22]	OUT	NENCPDHI		
128	CPDIN22	CPD[22]	IN	–		
129	CPDOUT21	CPD[21]	OUT	NENCPDHI		
130	CPDIN21	CPD[21]	IN	–		
131	CPDOUT20	CPD[20]	OUT	NENCPDHI		
132	CPDIN20	CPD[20]	IN	–		
133	CPDOUT19	CPD[19]	OUT	NENCPDHI		
134	CPDIN19	CPD[19]	IN	–		
135	CPDOUT18	CPD[18]	OUT	NENCPDHI		
136	CPDIN18	CPD[18]	IN	–		
137	CPDOUT17	CPD[17]	OUT	NENCPDHI		
138	CPDIN17	CPD[17]	IN	–		
139	CPDOUT16	CPD[16]	OUT	NENCPDHI		
140	CPDIN16	CPD[16]	IN	–		
141	CPDOUT15	CPD[15]	OUT	NENCPDHI		
142	CPDIN15	CPD[15]	IN	–		
143	CPDOUT14	CPD[14]	OUT	NENCPDHI		
144	CPDIN14	CPD[14]	IN	–		
145	CPDOUT13	CPD[13]	OUT	NENCPDHI		
146	CPDIN13	CPD[13]	IN	–		
147	CPDOUT12	CPD[12]	OUT	NENCPDHI		
148	CPDIN12	CPD[12]	IN	–		
149	CPDOUT11	CPD[11]	OUT	NENCPDHI		
150	CPDIN11	CPD[11]	IN	–		
151	CPDOUT10	CPD[10]	OUT	NENCPDHI		
152	CPDIN10	CPD[10]	IN	–		
153	CPDOUT9	CPD[9]	OUT	NENCPDHI		
154	CPDIN9	CPD[9]	IN	–		
155	CPDOUT8	CPD[8]	OUT	NENCPDHI		
156	CPDIN18	CPD[8]	IN	–		

BOUNDARY SCAN TABLE (Cont.)

No.	Cell Name	Pin	Type	Output Enable	INTEST	EXTEST/CLAMP
				BS cell	Guard Value	Guard Value
157	NENCPDLO	–	OUTEN0	–	1	
158	CPDIN7	CPD[7]	IN	–		
159	CPDOUT7	CPD[7]	OUT	NENCPDLO		
160	CPDIN6	CPD[6]	IN	–		
161	CPDOUT6	CPD[6]	OUT	NENCPDLO		
162	CPDIN5	CPD[5]	IN	–		
163	CPDOUT5	CPD[5]	OUT	NENCPDLO		
164	CPDIN4	CPD[4]	IN	–		
165	CPDOUT4	CPD[4]	OUT	NENCPDLO		
166	CPDIN3	CPD[3]	IN	–		
167	CPDOUT3	CPD[3]	OUT	NENCPDLO		
168	CPDIN2	CPD[2]	IN	–		
169	CPDOUT2	CPD[2]	OUT	NENCPDLO		
170	CPDIN1	CPD[1]	IN	–		
171	CPDOUT1	CPD[1]	OUT	NENCPDLO		
172	CPDIN0	CPD[0]	IN	–		
173	CPDOUT0	CPD[0]	OUT	NENCPDLO		
174	DIN0	D[0]	IN	–		
175	DOUT0	D[0]	OUT	NENDOUT		
176	DIN1	D[1]	IN	–		
177	DOUT1	D[1]	OUT	NENDOUT		
178	DIN2	D[2]	IN	–		
179	DOUT2	D[2]	OUT	NENDOUT		
180	DIN3	D[3]	IN	–		
181	DOUT3	D[3]	OUT	NENDOUT		
182	DIN4	D[4]	IN	–		
183	DOUT4	D[4]	OUT	NENDOUT		
184	DIN5	D[5]	IN	–		
185	DOUT5	D[5]	OUT	NENDOUT		
186	DIN6	D[6]	IN	–		
187	DOUT6	D[6]	OUT	NENDOUT		
188	DIN7	D[7]	IN	–		
189	DOUT7	D[7]	OUT	NENDOUT		
190	DIN8	D[8]	IN	–		
191	DOUT8	D[8]	OUT	NENDOUT		
192	DIN9	D[9]	IN	–		
193	DOUT9	D[9]	OUT	NENDOUT		
(TO TDO)						

Type Key

IN	Input Pad
OUT	Output Pad
INEN1	Input Enable Active High
OUTEN0	Output Enable Active Low

OUTPUT ENABLE BOUNDARY-SCAN CELLS

The boundary-scan register cells nENDOUT, nENCPDH, nENCPDLO, NABE, NCBE, NCPE, and NMSE control the output drivers of three-state outputs as shown in the table above. In the case of type OUTEN0 enable cells (e.g. Nendout), loading a 1 into the cell will place the associated drivers into the three-state state, while in the case of type INEN1 enable cells (e.g. Nabe), loading a 0 into the cell will three-state the associated drivers.

If the on-chip system logic causes the drivers controlled by nENDOUT, for example, to be three-state, i.e. by driving the signal nENDOUT HIGH, then a 1 will be observed on this cell if the SAMPLE/PRELOAD or INTEST instructions are active.

To put all ARM600 three-state outputs into their high impedance state, a logic 1 should be clocked into the output enable boundary-scan cells nENDOUT, nENCPDHI and nENCPDLO and a logic 0 should be clocked into NABE, NCBE, NCPE, and NMSE. Alternatively, the HI-Z instruction can be used.

SINGLE-STEP OPERATION

ARM600 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 serializing 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.

DC PARAMETERS

ABSOLUTE MAXIMUM RATINGS

Symbol	Parameter	Min	Max	Units	Note
VDD	Supply Voltage	VSS–0.3	VSS+7.0	V	1
VIP	Voltage Applied to Any Pin	VSS–0.3	VDD+0.3	V	1
TS	Storage Temperature	–40	125	°C	1

DC OPERATING CONDITIONS

Symbol	Parameter	Min	Typ	Max	Units	Note
VDD	Supply Voltage	4.75	5.0	5.25	V	
VIBC	IC Input HIGH Voltage	3.5		VDD	V	2,3
VILC	IC Input LOW Voltage	0.0		1.5	V	2,3
VIHT	IT/ITP Input HIGH Voltage	2.4		VDD	V	2,4,5
VILT	IT/ITP Input LOW Voltage	0.0		0.8	V	2,4,5
TA	Ambient Operating Temperature	0		70	°C	

DC CHARACTERISTICS, Note 6

Symbol	Parameter	Typ	Units	Note
IDD	Supply Current		nA	
ISC	Output Short Circuit Current		nA	
ILU	D.C. Latch-up Current		nA	
LIN	IT Input Leakage Current		uA	
LINP	ITPInput Leakage Current		uA	
LOH	Output HIGH Current (Vout=VSS+0.4V)		nA	
IOL	Output LOW Current (Vout=VSS+0.4V)		nA	
VIHTK	IC Input HIGH Voltage Threshold		V	
VILTK	IC Input LOW Voltage Threshold		V	
VIHTT	IT/ITP Input HIGH Voltage Threshold		V	
VILTT	IT/ITP Input LOW Voltage Threshold		V	
CIN	Input Capacitance		pF	

Note:

1. These are stress ratings only. Exceeding the absolute maximum ratings may permanently damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability.
2. Voltages measured with respect to VSS.
3. IC – CMOS-level inputs.
4. IT – TTL-level inputs (includes IT and ITOTZ pin types).
5. ITP – TTL-level inputs with pullups.
6. Nominal values shown are derived from transient analysis simulations.

AC PARAMETERS

TEST CONDITIONS

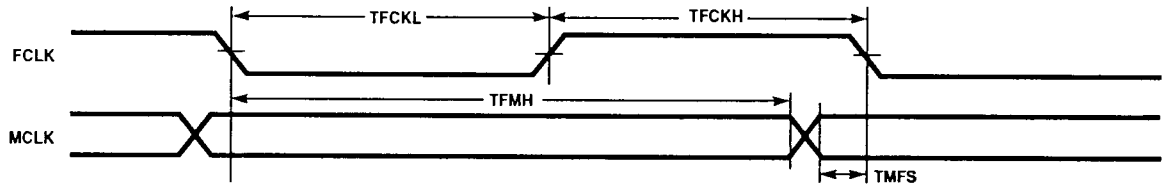
The AC timing diagrams presented in this section assume that the outputs of ARM600 have been loaded with the capacitive loads shown in the 'Test Load' column of the following table; these loads have been chosen as typical of the system in which ARM600 might be employed. The output pads of ARM600 are CMOS drivers which exhibit a propagation delay that increases linearly with the increase in load capacitance. An 'Output Derating' figure is given for each output pad, showing the approximate rate of increase of output time with increasing load capacitance.

AC TEST LOADS

Output Signal	Test Load (pf)	Output Derating (ns/pf)
A[25:0]	50	0.072
D[31:0]	50	0.072
NRW	50	0.072
NBW	50	0.072
LOCK	50	0.072
NMREQ	50	0.072
SEQ	50	0.072
CPCLK	30	0.072
CPSPV	30	0.072
NCPOPC	30	0.072
NCPI	30	0.072
NCPWT	30	0.072
CPD[31:0]	30	0.072

RELATIONSHIP BETWEEN FCLK & MCLK

Symbol	Parameter	Min	Max	Units	Note
TFCKL	FCLK LOW Time	20		ns	1
TFCKH	FCLK HIGH Time	20		ns	1
TFMH	FCLK – MCLK Hold Time	25		ns	2
TMFS	MCLK – FCLK Setup	0		ns	2



Notes:

1. FCLK timings measured at 50% of Vdd.
2. This parameter is only valid in synchronous mode (when the pin SNA is tied to Vdd). In asynchronous mode, there need be no relationship between FCLK and MCLK.

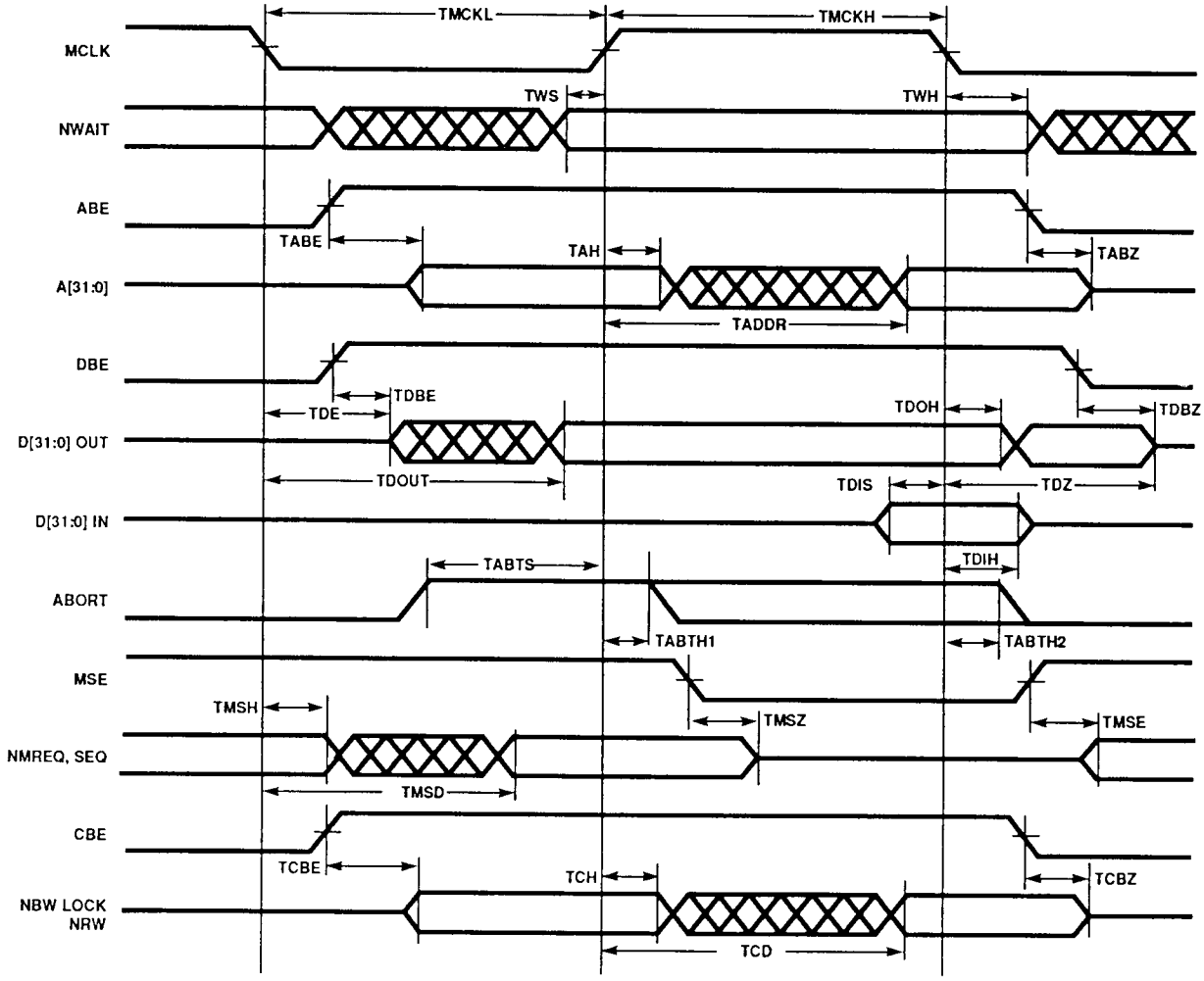
MAIN BUS SIGNALS

Symbol	Parameter	Min	Max	Units	Note
TMCKL	MCLK LOW Time	2/		ns	1
TMCKH	MCLK HIGH Time	20		ns	
TWS	NWAIT Setup to MCLK	10		ns	
TWH	NWAIT Hold From MCLK	5		ns	
TABE	Address Bus Enable		15	ns	2
TABZ	Address Bus Disable		10	ns	
TADDR	MCLK to Address Delay		15	ns	2
TAH	Address Hold Time	5		ns	2
TDBE	DBE to Data Enable		15	ns	2
TDE	MCLK to Data Enable		15	ns	2
TDBZ	DBE to Data Disable		10	ns	
TDZ	MCLK to Data Disable		10	ns	
TDOUT	Data Out Delay		20	ns	2
TDOH	Data Out Hold	5		ns	2
TDIS	Data in Setup	5		ns	
TDIH	Data in Hold	10		ns	
TABTS	ABORT Setup Time	20		ns	
TABTH1	ABORT Hold Time	5		ns	3
TABTH2	ABORT Hold Time	5		ns	3
TMSE	NMREQ & SEQ Enable		20	ns	
TMSZ	NMREQ & SEQ Disable		10	ns	
TMSD	NMREQ & SEQ Delay		35	ns	
TMSH	NMREQ & SEQ Hold	5		ns	
TCBE	Control Bus Enable		15	ns	
TCBZ	Control Bus Disable		10	ns	
TCD	MCLK to Control Valid		20	ns	
TCH	Control Hold Time	5		ns	

Notes:

1. MCLK timings measured between clock edges at 50% of Vdd.
2. The timings of these buses are measured to TTL levels.
3. TABTH1 is a requirement for ARM600. To ensure compatibility with future processors, designs should meet TABTH2. TABTH2 is not tested on ARM600.

MAIN BUS SIGNALS



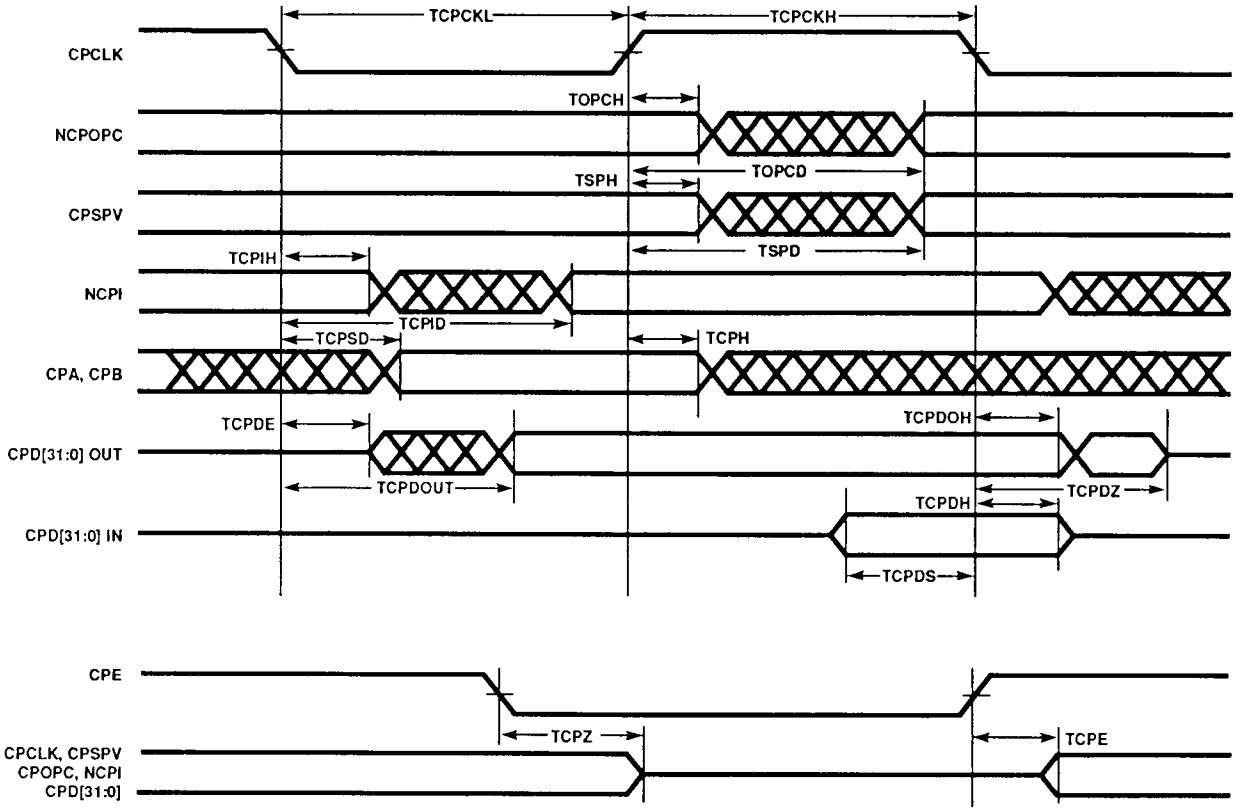
COPROCESSOR BUS SIGNALS, Note 3

Symbol	Parameter	Min	Max	Units	Note
TCPCKL	Clock LOW Time	18		ns	1
TCPCKH	Clock HIGH Time	18		ns	1
TOPCD	CPCLK to NCPOPC Valid	10	15	ns	
TOPCH	NCPOPC Hold Time	5		ns	
TSPD	CPCLK to CPSPV Valid		15	ns	
TSPH	CPSPV Hold Time	5		ns	
TCPI	CPCLK to NCPI Valid		15	ns	
TCPIH	CPI Hold Time	5		ns	
TCPS	CPA/CPB Setup	15		ns	
TCPSD	CPA/CPB Delay		0	ns	2
TCPH	CPA/CPB Hold	5		ns	
TCPDE	Data Out Enable	5	10	ns	
TCPDZ	Data Out Disable		5	ns	
TCPDOUT	Data Out Delay		15	ns	
TCPDOH	Data Out Hold	5		ns	
TCPDS	Data In Setup	10		ns	
TCPDH	Data In Hold	5		ns	
TCPE	Coproc Bus Enable		15	ns	
TCPZ	Coproc Bus Disable	5	10	ns	

Notes:

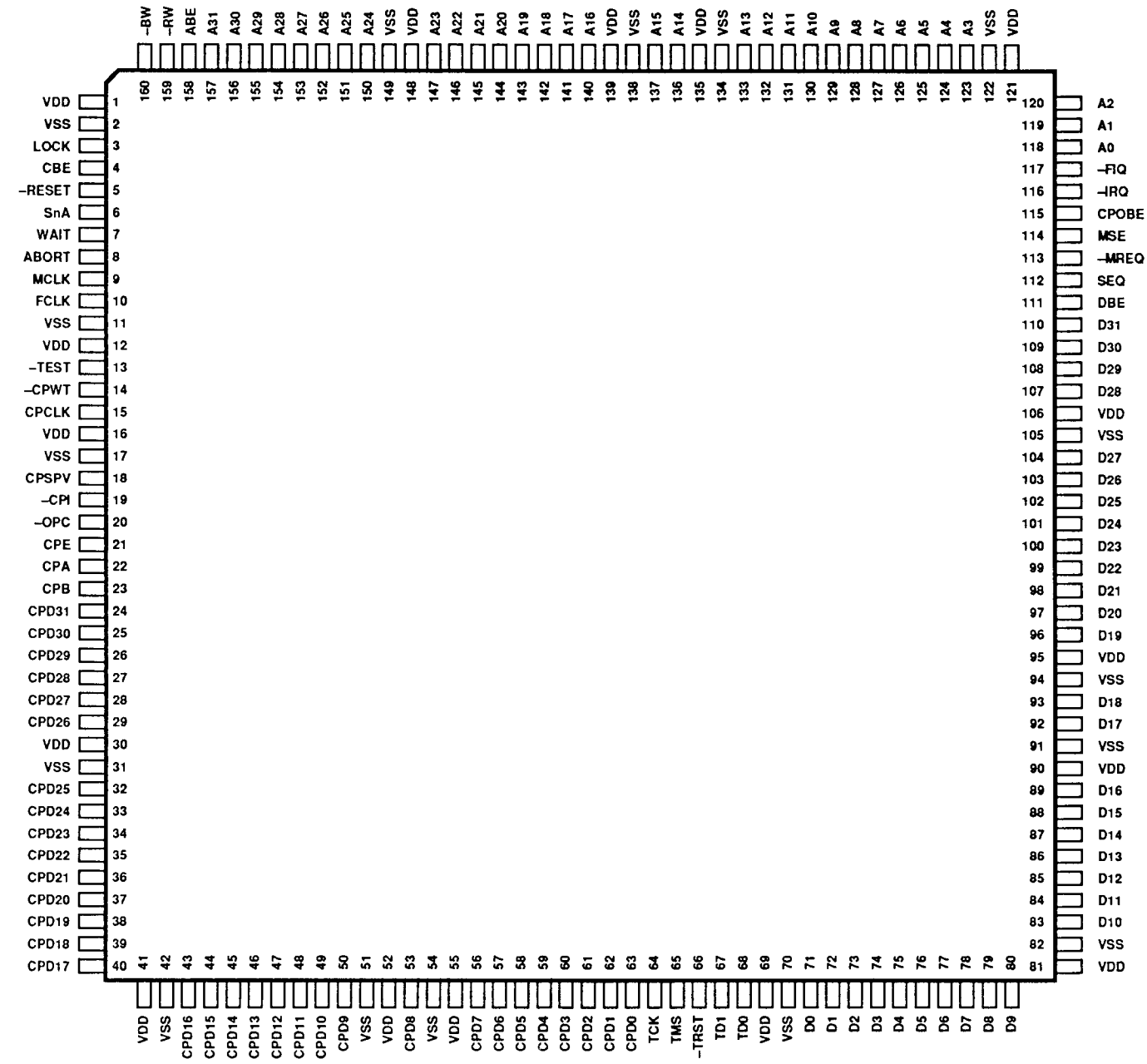
1. CPCLK is synthesized internally from FCLK or MCLK according to what ARM600 is doing. The minimum CPCLK will be generated from FCLK. This parameter assumes that FCLK has a minimum HIGH and LOW time of 20 ns.
2. This parameter only applies during a CPDT to/from external memory, when the NMREQ/SEQ output delay (TMSd) will be adversely affected if this limit is exceeded. Note that ARM600 will always be synchronized to MCLK under these conditions, (Tcpckl & Tcpckh will be correspondingly larger), so this parameter is seldom a limiting condition.
3. These timings assume that all these signals are equally loaded.

COPROCESSOR BUS SIGNALS



PIN DIAGRAM

ARM600 is packaged in a 160-pin Plastic Quad Flat Pack.



PINOUT

Pin No.	Pin Name	Direction
1	VDD	–
2	VSS	–
3	LOCK	O
4	CBE	I
5	NRESET	I
6	SNA	I
7	NWAIT	I
8	ABORT	I
9	MCLK	I
10	FCLK	I
11	VSS	–
12	VDD	–
13	NTEST	I
14	NCPWT	O
15	CPCLK	O
16	VDD	–
17	VSS	–
18	CPSPV	O
19	NCPI	O
20	NOPC	O
21	CPE	I
22	CPA	I
23	CPB	I
24	CPD[31]	I/O
25	[30]	
26	[29]	
27	[28]	
28	[27]	
29	CPD[26]	I/O
30	VDD	–
31	VSS	–
32	CPD[26]	I/O
33	[24]	
34	[23]	
35	[22]	
36	[21]	
37	[20]	
38	[19]	
39	[18]	
40	CPD[19]	I/O
41	VDD	–
42	VSS	–
43	CPD[16]	I/O
44	[15]	
45	[14]	

Pin No.	Pin Name	Direction
46	[13]	
47	[12]	
48	[11]	
49	[10]	
50	CPD[9]	I/O
51	VDD	–
52	VSS	
53	CPD[8]	I/O
54	VDD	–
55	VSS	–
56	CPD[7]	I/O
57	[6]	
58	[5]	
59	[4]	
60	[3]	
61	[2]	
62	[1]	
63	CPD[0]	I/O
64	TCK	I
65	TMS	I
66	NTRST	I
67	TDI	I
68	TDO	O
69	VDD	–
70	VSS	–
71	D[0]	I/O
72	[1]	
73	[2]	
74	[3]	
75	[4]	
76	[5]	
77	[6]	
78	[7]	
79	[8]	
80	D[9]	I/O
81	VDD	–
82	VSS	–
83	D[10]	I/O
84	[11]	
85	[12]	
86	[13]	
87	[14]	
88	[15]	
89	D[16]	I/O

Pin No.	Pin Name	Direction
90	VDD	–
91	VSS	–
92	D[17]	I/O
93	D[18]	I/O
94	VDD	–
95	VSS	–
96	D[19]	I/O
97	[20]	
98	[21]	
99	[22]	
100	[23]	
101	[24]	
102	[25]	
103	[26]	
104	D[27]	I/O
105	VSS	–
106	VDD	–
107	D[28]	I/O
108	[29]	
109	[30]	
110	D[31]	I/O
111	DBE	I
112	SEQ	O
113	NMREQ	O
114	MSE	I
115	CPDBE	I
116	NIRQ	I
117	NFIQ	I
118	A[0]	O
119	[1]	
120	A[2]	O
121	VDD	–
122	VSS	–
123	A[3]	O
124	[4]	
125	[5]	
126	[6]	
127	[7]	
128	[8]	
129	[9]	
130	[10]	
131	[11]	
132	[12]	
133	A[13]	O
134	VSS	–

PINOUT (Cont.)

Pin No.	Pin Name	Direction
135	VDD	–
136	A[14]	O
137	A[15]	O
138	VSS	–
139	VDD	–
140	A[16]	O
141	[17]	
142	[18]	
143	[19]	
144	[20]	
145	[21]	
146	[22]	
147	A[23]	O
148	VDD	–
149	VSS	–
150	A[24]	O
151	[25]	
152	[26]	
153	[27]	
154	[28]	
155	[29]	
156	[30]	
157	A[31]	O
158	ABE	I
159	NRW	O
160	NBW	O

VLSI CORPORATE OFFICES

CORPORATE HEADQUARTERS • VLSI PRODUCT DIVISIONS
VLSI Technology, Inc. • 1109 McKay Drive • San Jose, CA 95131 • 408-434-3100

PORTABLE SYSTEMS DIVISION AND DESKTOP SYSTEMS DIVISION
VLSI Technology, Inc. • 8375 South River Parkway • Tempe, AZ 85284 • 602-752-8574

VLSI SUBSIDIARY

COMPASS DESIGN AUTOMATION, INC.
1865 Lundy Avenue • San Jose, CA 95131 • 408-433-4880

VLSI SALES OFFICES AND TECH CENTERS

UNITED STATES

ARIZONA
Tempe, 602-752-6450
FAX 602-752-6001

CALIFORNIA
San Jose, 408-922-5200
FAX 408-922-5252

Encino, 818-609-9981
FAX 818-609-0535

Irvine, 714-250-4900
FAX 714-250-9041

FLORIDA
Pompano Beach, 305-971-0404
FAX 305-971-2086

GEORGIA
Duluth, 404-476-8574
FAX 404-476-3790

ILLINOIS
Hoffman Estates, 708-884-0500
FAX 708-884-9394

MARYLAND
Millersville, 410-987-8777
FAX 410-987-4489

MASSACHUSETTS
Wilmington, 508-658-9501
FAX 508-657-6420

NEW JERSEY
Plainsboro, 609-799-5700
FAX 609-799-5720

NORTH CAROLINA
Durham, 919-544-1891/92
FAX 919-544-6667

TEXAS
Richardson, 214-231-6716
FAX 214-669-1413

WASHINGTON
Bellevue, 206-453-5414
FAX 206-453-5229

INTERNATIONAL

FRANCE
Palaiseau Cedex, 1-69-19-71-00
FAX 1-69-19-71-01

GERMANY
Muenchen, 89-92795-0
FAX 89-92795-145

HONG KONG
Wanchai, 852-802-7755
FAX 852-802-7959

ITALY
Argrate Brianza, 39-6056791
FAX 39-6056808

JAPAN
Tokyo, 03-3239-5211
FAX 03-3239-5215

Chuo-ku, Osaka, 06-243-6041
FAX 06-243-6960

TAIWAN
Taipei, 886-2-325-4422
FAX 886-2-325-4411

UNITED KINGDOM
Milton Keynes, 09 08/66 75 95
FAX 09 08/67 00 27

VLSI SALES OFFICES

UNITED STATES

ALABAMA
Huntsville, 205-539-5513
FAX 205-536-8622

FLORIDA
Clearwater, 813-538-0681
FAX 813-538-8379

MINNESOTA
St. Louis Park, 612-545-1490
FAX 612-545-3489

NEW YORK
E. Rochester, 716-586-0670
FAX 716-586-0672
Fishkill, 914-897-8574
FAX 914-897-2363

OHIO
Cleveland, 216-292-8235
FAX 216-464-7609

OREGON
Portland, 503-244-9882
FAX 503-245-0375

TEXAS
Austin, 512-343-8191
FAX 512-343-2759

INTERNATIONAL

JAPAN
Tokyo, 03-3262-0850
FAX 03-3262-0881

SINGAPORE
Singapore, 65-742-2314
FAX 65-741-1979

VLSI AUTHORIZED DESIGN CENTERS

UNITED STATES

COLORADO
SIS MICROELECTRONICS, INC.
Longmont, 303-776-1667

ILLINOIS
ASIC DESIGNS
Naperville, 708-717-5841

MAINE
QUADIC SYSTEMS, INC.
South Portland, 207-871-8244

PENNSYLVANIA
INTEGRATED CIRCUIT SYSTEMS, INC.
Valley Forge, 215-666-1900

INTERNATIONAL

EIRE AND U.K.
PA TECHNOLOGY
Herts, 76-261222

SYMBIONICS
Cambridge, 223-421025

FRANCE
CETIA
Toulon Cedex, 9-42-12005

SOREP
Chateaubourg, 99-623955

INDIA
ARCUS TECHNOLOGY LTD.
Bangalore, 91-812-217307

JAPAN
ADC CORPORATION
Tokyo, 03-3492-1251

LSI SYSTEMS, INC.
Kanagawa, 0462-29-3220

NIPPON STEEL CORPORATION
Tokyo, 03-5566-2141

TOMEN ELECTRONICS
Tokyo, 03-3506-3650

PALTEK CORPORATION
Tokyo, 03-3707-5455

KOREA
ANAM VLSI DESIGN CENTER
Seoul, 82-2-553-2106

NORWAY
NORKRETS AS
Oslo, 47-2360677/8

VLSI SALES REPRESENTATIVES

UNITED STATES

ARIZONA
LUSCOMBE ENGINEERING
Scottsdale, 602-949-9333

CALIFORNIA
EMERGING TECHNOLOGY
San Jose, 408-263-9366
Cameron Park, 916-676-4387

COLORADO
LUSCOMBE ENGINEERING
Longmont, 303-772-3342

IDAHO
EMERGING TECHNOLOGY
Boise, 208-378-4680

IOWA
SELTEC SALES
Cedar Rapids, 319-364-7660

KANSAS
ELECTRI-REP
Overland Park, 913-649-2168

MISSOURI
ELECTRI-REP
St. Louis, 314-993-4421

NEW YORK
ADVANCED COMPONENTS
Syracuse, 315-437-6700

OHIO
APPLIED DATA MANAGEMENT
Cincinnati, 513-579-8108

OREGON
ELECTRA TECHNICAL SALES
Beaverton, 503-643-5074

UTAH
LUSCOMBE ENGINEERING
Salt Lake City, 801-565-9885

WASHINGTON
ELECTRA TECHNICAL SALES
Kirkland, 206-821-7442

INTERNATIONAL

AUSTRALIA
GEORGE BROWN GROUP
Adelaide, 61-8-352-2222
Brisbane, 61-7-252-3876
Melbourne, 61-3-878-8111
Newcastle, 61-49-69-6399
Perth, 61-9-362-1044
Sydney, 61-2-638-1888

AUSTRIA
THOMAS NEUROTH
Wien, 0043-222-82 56 45

BRAZIL
UNIAO DIGITAL COMERCIAL
Sao Paulo, 55-11 533-0967

CANADA
DAVETEK MARKETING, INC.
British Columbia, 604-430-3680
Alberta, 403-250-2034

INTELATECH, INC.
Mississauga, 416-629-0082

HONG KONG
LESTINA INTERNATIONAL, LTD
Tsimshatsui, 852-7351736

ISRAEL
RDT ELECTRONICS ENG. LTD
Tel-Aviv, 23-4832119

SINGAPORE
DYNAMIC SYSTEMS PTE, LTD
Singapore, 65-742-1986

TAIWAN
PRINCETON TECH CORP.
Taipei, 886-2-917-8856

WEIKENG INDUSTRIAL CO.
Taipei, 886-2-776-3998

THAILAND
TRON ELECTRONICS CO. LTD
Bangkok, 66 2 260-3913

VLSI DISTRIBUTORS

UNITED STATES

(represented by **Arrow/Schweber** except where noted)

ALABAMA
Huntsville, 205-837-6955

ARIZONA
Phoenix, 602-431-0030

CALIFORNIA
Los Angeles, 818-880-9686
Orange County, 714-838-5422
San Diego, 619-565-4800
San Jose, 408-441-9700

COLORADO
Denver, 303-799-0258

CONNECTICUT
Wallingford, 203-265-7741

FLORIDA
Lake Mary, 407-333-9300
Deerfield Beach, 305-429-8200

GEORGIA
Atlanta, 404-497-1300

ILLINOIS
Chicago, 708-250-0500

INDIANA
Indianapolis, 317-299-2071

IOWA
Cedar Rapids, 319-395-7230

KANSAS
Kansas City, 913-541-9542

MARYLAND
Baltimore, 301-596-7800

MASSACHUSETTS
Boston, 508-658-0900

MICHIGAN
Detroit, 313-462-2290

MINNESOTA
Minnesota, 612-941-5280

MISSOURI
St. Louis, 314-567-6888

NEW JERSEY
Philadelphia, 609-596-8000
Pine Brook, 201-227-7880

NEW YORK
Rochester, 716-427-0300
Happuage, 516-231-1000

NORTH CAROLINA
Raleigh, 919-876-3132

OHIO
Dayton, 513-435-5563
Cleveland, 216-248-3990

OKLAHOMA
Tulsa, 918-252-7537

OREGON
ALMAC/ARROW
Portland, 503-629-8090

PENNSYLVANIA
Pittsburgh, 412-963-6807

TEXAS
Austin, 512-835-4180
Dallas, 214-380-6464
Houston, 713-530-4700

UTAH
Salt Lake City, 801-973-6913

WASHINGTON
ALMAC/ARROW
Seattle, 206-643-9992
Spokane, 509-924-9500

WISCONSIN
Milwaukee, 414-792-0150

INTERNATIONAL

BELGIUM/LUXEMBURG
MICROTRON
Mechelen, 215-212223

CANADA
ARROW/SCHWEBER
Montreal, 514-421-7411
Ottawa, 613-226-6903
Toronto, 416-670-7769
Vancouver, 604-421-2333

SEMAD
Calgary, 403-252-5664
Markham, 416-475-8500
Montreal, 514-694-0860
Ottawa, 613-727-8325
British Columbia, 604-420-9889

DENMARK
DELCO
Lynge, 45 42-189533

ENGLAND
HAWKE COMPONENTS
Bramley, NR Basingstoke
256-880800

KUDOS-THAME LTD
Berks, 734-351010

FINLAND
COMDAX
Helsinki, 80-670277

FRANCE
ASAP s.a.
Montigny-le-Bretonneux,
1-30438233

GERMANY
DATA MODUL GmbH
Muenchen, 089-560170

ELECTRONIC 2000 AG
Muenchen, 089-420010

EAST GERMANY
ZENTRUM MIKROELEKTRONIK
Dresden, 0037-51-588464

ITALY
INTER-REP S.P.A.
Torino, 11-2165901

JAPAN
ASAHI GLASS CO. LTD
Tokyo, 03-3218-5854

TOKYO ELECTRON, LTD
Tokyo, 03-3340-8111

NETHERLANDS
TME
Aa Heeswijk-Dinther,
4139-8895

PUERTO RICO
ISLA CARIBE ELECTRO SALES
Guaynabo, 809-720-4430

SWEDEN
TRACO AB
Farsta, 8-930000

SPAIN AND PORTUGAL
SEMICONDUCTORES s.a.
Barcelona, 3-21723 40

SWITZERLAND
DECTRO SWISS ELECTRONIC DESIGN AG
Zuerich, 0041-1-3868-600

6/92

130770-001 2.5K 0692