5.0 Memory Interface

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

5.1 Cycle types

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

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

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

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

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

These four classes are distinguishable to the memory system by inspection of the **nMREQ** and **SEQ** control lines (see *Table 6: Memory Cycle Types*). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

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

Table 6: Memory Cycle Types

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

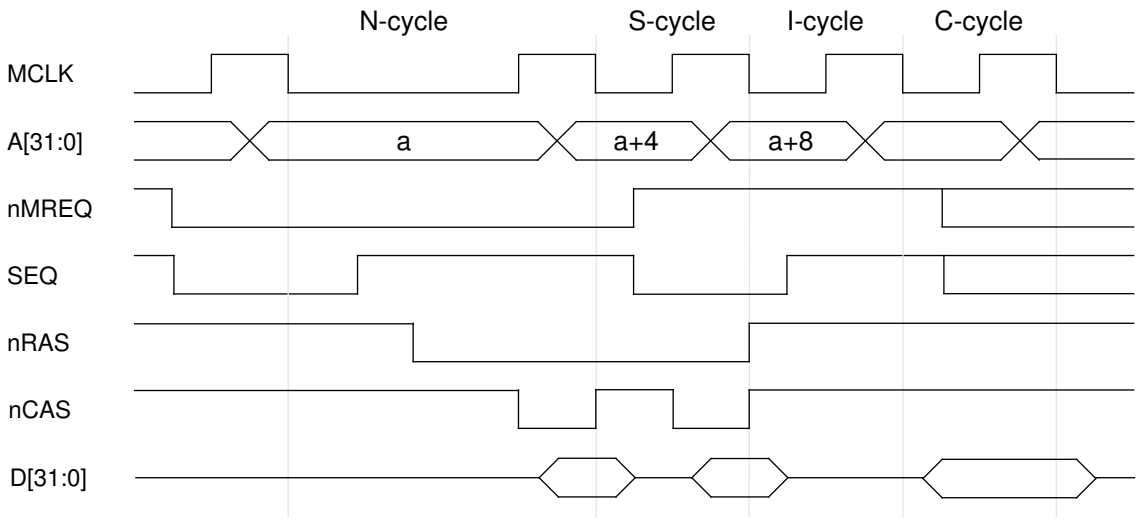

Figure 29: ARM Memory Cycle Timing

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

The processor clock must be stretched to match the full access. When an S-cycle follows an I- or C-cycle, the address will be the same as that used in the I- or C-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to run at page mode speed whilst performing a full DRAM access. This is shown in *Figure 30: Memory Cycle Optimization*.

5.2 Byte addressing

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

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

When a byte is requested in a read transfer (LDRB), the memory system can safely ignore that the request is for a byte quantity and present the whole word.

ARM60 will perform the byte extraction internally. Alternatively, the memory system may activate only the addressed byte of the memory. This may be desirable in order to save power, or to enable the use of a common decoding system for both read and write cycles.

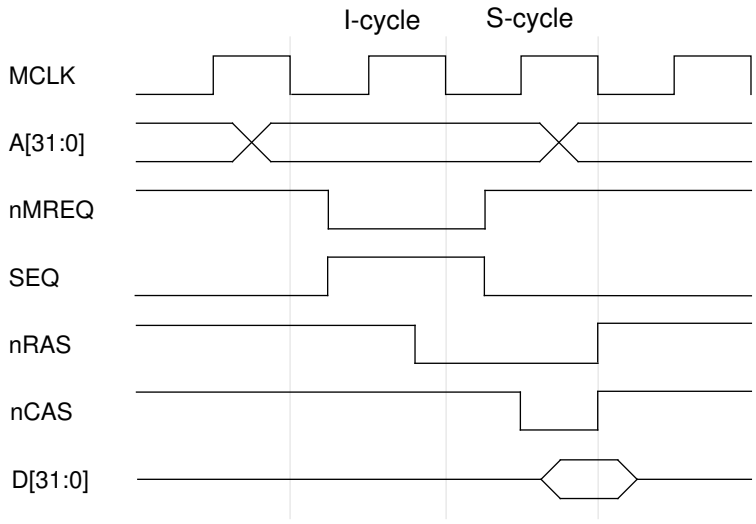

Figure 30: Memory Cycle Optimization

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

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

When the processor is configured for Little Endian operation byte 0 of the memory system should be connected to data lines 7 through 0 (**D[7:0]**) and strobed by **nCAS0**. **nCAS1** drives the bank connected to data lines 15 though 8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time critical signal.

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

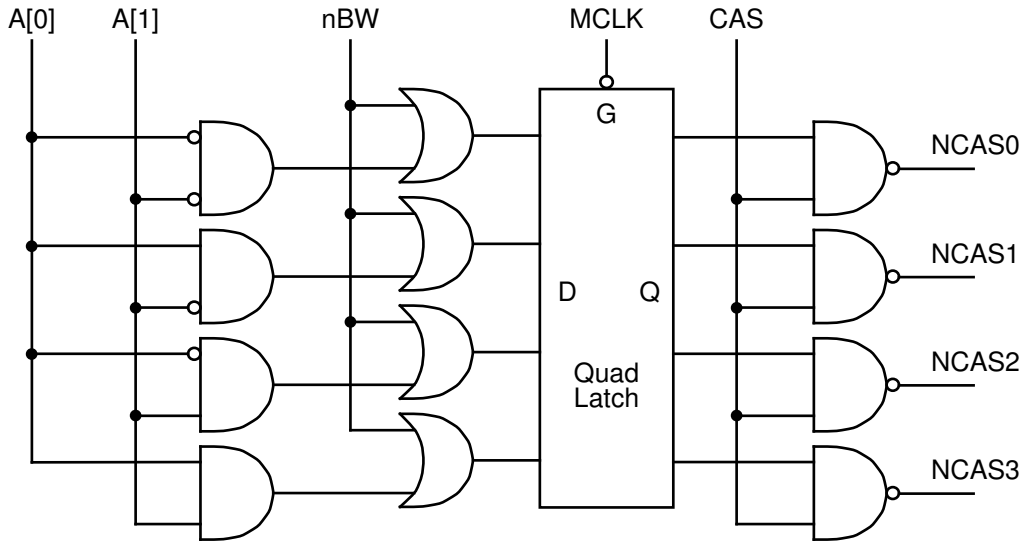

Figure 31: Decoding Byte Accesses to Memory

5.3 Address timing

Normally the processor address changes during phase 2 to the value which the memory system should use during the following cycle. This gives maximum time for driving the address to large memory arrays, and for address translation where required. Dynamic memories usually latch the address on chip, and if the latch is timed correctly they will work even though the address changes before the access has completed.

Static RAMs and ROMs will not work under such circumstances, as they require the address to be stable until after the access has completed. Therefore, for use with such devices, the address transition must be delayed until after the end of phase 2. An on-chip address latch, controlled by **ALE**, allows the address timing to be modified in this way. In a system with a mixture of static and dynamic memories (which for these purposes means a mixture of devices with and without address latches), the use of **ALE** may change dynamically from one cycle to the next, at the discretion of the memory system.

5.4 Memory management

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

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

(2) **nTRANS** indicates whether the processor is in user or a privileged mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user.

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

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

5.5 Locked operations

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

5.6 Stretching access times

All memory timing is defined by **MCLK**, and long access times can be accommodated by stretching this clock. It is usual to stretch the LOW period of **MCLK**, as this allows the memory manager to abort the operation if the access is eventually unsuccessful (**ABORT** must be setup prior to the rising edge of **MCLK** if **LATEABT** is LOW configuring ARM60 for early aborts).

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

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

6.0 Coprocessor Interface

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

6.1 Interface signals

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

6.1.1 Coprocessor present/absent

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

6.1.2 Busy-waiting

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

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

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

6.1.3 Pipeline following

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

6.2 Data transfer cycles

Once the coprocessor has gone not-busy in a data transfer instruction, it must supply or accept data at the ARM60 bus rate (defined by **MCLK**). It can deduce the direction of transfer by inspection of the L bit in the instruction, but must only drive the bus when permitted to by **DBE** being HIGH. The coprocessor is responsible for determining the number of words to be transferred; ARM60 will continue to increment the address by one word per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor driving **CPA** and **CPB** HIGH.

There is no limit in principle to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case ARM60 interrupt latency, as the instruction is not 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.

6.3 Register transfer cycle

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

6.4 Privileged instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the **nTRANS** output.

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

6.5 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, ie must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to an ARM60 register. The coprocessor must stay busy while it performs the floating point to fixed point conversion, as ARM60 will expect to receive the integer value on the cycle immediately following that

where it goes not-busy. The coprocessor must therefore preserve the original floating point value and not corrupt it during the conversion, because it will be required again if an interrupt arises during the busy period.

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

6.6 Undefined instructions

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

7.0 Instruction Cycle Operations

In the following tables **nMREQ** and **SEQ** (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the type of the next cycle. The address, **nBW**, **nRW**, and **nOPC** (which appear up to half a cycle ahead) are shown in the cycle to which they apply.

7.1 Branch and branch with link

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

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

The third cycle performs a fetch from the destination + 4, refilling the instruction pipeline, and if the branch is with link R14 is modified (4 is subtracted from it) to simplify return from SUB PC,R14,#4 to MOV PC,R14. This makes the STM..{R14} LDM..{PC} type of subroutine work correctly. The cycle timings are shown below in *Table 7: Branch Instruction Cycle Operations*

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
1	pc+8	1	0	(pc + 8)	0	0	0
2	alu	1	0	(alu)	0	1	0
3	alu+4	1	0	(alu + 4)	0	1	0
	alu+8						

Table 7: Branch Instruction Cycle Operations

pc is the address of the branch instruction

alu is an address calculated by ARM60

(alu) are the contents of that address, etc

7.2 Data Operations

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

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

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

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

PSR Transfer operations exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register. The cycle timings are shown below *Table 8: Data Operation Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
normal	1	pc+8	1	0	(pc+8)	0	1	0
		pc+12						
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	0	1	0
	3	alu+4	1	0	(alu+4)	0	1	0
		alu+8						
shift(Rs)	1	pc+8	1	0	(pc+8)	1	0	0
	2	pc+12	1	0	-	0	1	1
		pc+12						
shift(Rs)	1	pc+8	1	0	(pc+8)	1	0	0
dest=pc	2	pc+12	1	0	-	0	0	1
	3	alu	1	0	(alu)	0	1	0
	4	alu+4	1	0	(alu+4)	0	1	0
		alu+8						

Table 8: Data Operation Instruction Cycle Operations

7.3 Multiply and multiply accumulate

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

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

All cycles except the first are internal. The cycle timings are shown below in *Table 9: Multiply Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
(Rs)=0,1	1	pc+8	1	0	(pc+8)	1	0	0
	2	pc+12	1	0	-	0	1	1
		pc+12			(pc+8)			
(Rs)>1	1	pc+8	1	0	(pc+8)	1	0	0
	2	pc+12	1	0	-	1	0	1
	•	pc+12	1	0	-	1	0	1
	m	pc+12	1	0	-	1	0	1
	m+1	pc+12	1	0	-	0	1	1
		pc+12						

Table 9: Multiply Instruction Cycle Operations

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

7.4 Load register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in *Table 10: Load Register Instruction Cycle Operations*.

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

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

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
normal	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	b/w	0	(alu)	1	0	1
	3	pc+12	1	0	-	0	1	1
		pc+12						
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	b/w	0	pc’	1	0	1
	3	pc+12	1	0	-	0	0	1
	4	pc’	1	0	(pc’)	0	1	0
	5	pc’+4	1	0	(pc’+4)	0	1	0
		pc’+8						

Table 10: Load Register Instruction Cycle Operations

7.5 Store register

The first cycle of a store register is similar to the first cycle of load register. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle. The cycle timings are shown below in *Table 11: Store Register Instruction Cycle Operations*. The base write-back is prevented during a Data Abort if the processor is configured for Early Abort. The write-back is not prevented if Late Abort is configured.

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
1	pc+8	1	0	(pc+8)	0	0	0
2	alu	b/w	1	Rd	0	0	1
	pc+12						

Table 11: Store Register Instruction Cycle Operations

7.6 Load multiple registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, whilst performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown in *Table 12: Load Multiple Registers Instruction Cycle Operations*.

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

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

When the PC is in the list of registers to be loaded the current instruction pipeline must be invalidated.

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

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	1	0	1
	3	pc+12	1	0	-	0	1	1
		pc+12						
1 register	1	pc+8	1	0	(pc+8)	0	0	0
dest=pc	2	alu	1	0	pc’	1	0	1
	3	pc+12	1	0	-	0	0	1
	4	pc’	1	0	(pc’)	0	1	0
	5	pc’+4	1	0	(pc’+4)	0	1	0
		pc’+8						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	0	(alu)	0	1	1
	•	alu+•	1	0	(alu+•)	0	1	1
	n	alu+•	1	0	(alu+•)	0	1	1
	n+1	alu+•	1	0	(alu+•)	1	0	1
	n+2	pc+12	1	0	-	0	1	1
		pc+12						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>10)	2	alu	1	0	(alu)	0	1	1
incl pc	•	alu+•	1	0	(alu+•)	0	1	1
	n	alu+•	1	0	(alu+•)	0	1	1
	n+1	alu+•	1	0	pc’	1	0	1
	n+2	pc+12	1	0	-	0	0	1
	n+3	pc’	1	0	(pc’)	0	1	0
	n+4	pc’+4	1	0	(pc’+4)	0	1	0
		pc’+8						

Table 12: Load Multiple Registers Instruction Cycle Operations

7.7 Store multiple registers

Store multiple proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers to contend with. The cycle timings are shown in *Table 13: Store Multiple Registers Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	1	Ra	0	0	1
		pc+12						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	1	Ra	0	1	1
	•	alu+•	1	1	R•	0	1	1
	n	alu+•	1	1	R•	0	1	1
	n+1	alu+•	1	1	R•	0	0	1
		pc+12						

Table 13: Store Multiple Registers Instruction Cycle Operations

7.8 Data swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in *Table 14: Data Swap Instruction Cycle Operations*.

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

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

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

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	LOCK
1	pc+8	1	0	(pc+8)	0	0	0	0
2	Rn	b/w	0	(Rn)	0	0	1	1
3	Rn	b/w	1	Rm	1	0	1	1
4	pc+12	1	0	-	0	1	1	0
	pc+12							

Table 14: Data Swap Instruction Cycle Operations

7.9 Software interrupt and exception entry

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

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

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in *Table 15: Software Interrupt Instruction Cycle Operations*.

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
1	pc+8	1	0	(pc+8)	0	0	0	C
2	Xn	1	0	(Xn)	0	1	0	1
3	Xn+4	1	0	(Xn+4)	0	1	0	1
	Xn+8							

Table 15: Software Interrupt Instruction Cycle Operations

where C represents the current mode-dependent value.

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

7.10 Coprocessor data operation

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

If the coprocessor can never do the requested task, it should leave **CPA** and **CPB** HIGH. If it can do the task, but can't commit right now, it should drive **CPA** LOW but leave **CPB** HIGH until it can commit. ARM60 will busy-wait until **CPB** goes LOW. The cycle timings are shown in *Table 16: Coprocessor Data Operation Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	1	0	-	1	0	1	0	0	1
	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
		pc+12									

Table 16: Coprocessor Data Operation Instruction Cycle Operations

7.11 Coprocessor data transfer (from memory to coprocessor)

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

ARM60 spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address base during the transfer cycles. The cycle timings are shown in *Table 17: Coprocessor Data Transfer Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	1	0	(alu)	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
not ready	2	pc+8	1	0	-	1	0	1	0	0	1
	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	0	(alu)	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	1	0	(alu)	0	1	1	1	0	0
ready	•	alu+•	1	0	(alu+•)	0	1	1	1	0	0
	n	alu+•	1	0	(alu+•)	0	1	1	1	0	0
	n+1	alu+•	1	0	(alu+•)	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
(m>1)	2	pc+8	1	0	-	1	0	1	0	0	1
not ready	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	0	(alu)	0	1	1	1	0	0
	•	alu+•	1	0	(alu+•)	0	1	1	1	0	0
	n+m	alu+•	1	0	(alu+•)	0	1	1	1	0	0
	n+m+1	alu+•	1	0	(alu+•)	0	0	1	1	1	1
		pc+12									

Table 17: Coprocessor Data Transfer Instruction Cycle Operations

7.12 Coprocessor data transfer (from coprocessor to memory)

The ARM60 controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the nRW line is inverted during the transfer cycle. The cycle timings are show in *Table 18: Coprocessor Data Transfer Instruction Cycle Operations*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
not ready	2	pc+8	1	0	-	1	0	1	0	0	1
	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	1	1	CPdata	0	1	1	1	0	0
ready	•	alu+•	1	1	CPdata	0	1	1	1	0	0
	n	alu+•	1	1	CPdata	0	1	1	1	0	0
	n+1	alu+•	1	1	CPdata	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
(m>1)	2	pc+8	1	0	-	1	0	1	0	0	1
not ready	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	0	1	1	1	0	0
	•	alu+•	1	1	CPdata	0	1	1	1	0	0
	n+m	alu+•	1	1	CPdata	0	1	1	1	0	0
	n+m+1	alu+•	1	1	CPdata	0	0	1	1	1	1
		pc+12									

Table 18: Coprocessor Data Transfer Instruction Cycle Operations

7.13 Coprocessor register transfer (Load from coprocessor)

Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and ARM60 puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all ARM60 register load instructions. The cycle timings are shown in *Table 19: Coprocessor register transfer (Load from coprocessor)*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	0	CPdata	1	0	1	1	1	1
	3	pc+12	1	0	-	0	1	1	1	-	-
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	1	0	CPdata	1	0	1	0	0	1
	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	0	CPdata	1	0	1	1	1	1
	n+2	pc+12	1	0	-	0	1	1	1	-	-
		pc+12									

Table 19: Coprocessor register transfer (Load from coprocessor)

7.14 Coprocessor register transfer (Store to coprocessor)

As for the load from coprocessor, except that the last cycle is omitted. The cycle timings are shown below in *Table 20: Coprocessor register transfer (Store to coprocessor)*.

	Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	1	Rd	0	0	1	1	1	1
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	1	0	-	1	0	1	0	0	1
	•	pc+8	1	0	-	1	0	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	1	Rd	0	0	1	1	1	1
		pc+12									

Table 20: Coprocessor register transfer (Store to coprocessor)

7.15 Undefined instructions and coprocessor absent

When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive **CPA** or **CPB** LOW. These will remain HIGH, causing the undefined instruction trap to be taken. Cycle timings are shown in *Table 21: Undefined Instruction Cycle Operations*.

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB	nTRANS
1	pc+8	1	0	(pc+8)	1	0	0	0	1	1	Old
2	pc+8	1	0	-	0	0	0	1	1	1	Old
3	Xn	1	0	(Xn)	0	1	0	1	1	1	1
4	Xn+4	1	0	(Xn+4)	0	1	0	1	1	1	1
	Xn+8										

Table 21: Undefined Instruction Cycle Operations

7.16 Unexecuted instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see *Table 22: Unexecuted Instruction Cycle Operations*).

Cycle	Address	nBW	nRW	Data	nMREQ	SEQ	nOPC
1	pc+8	1	0	(pc+8)	0	1	0
	pc+12						

Table 22: Unexecuted Instruction Cycle Operations

7.17 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 23: ARM Instruction Speeds*. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

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

Table 23: ARM Instruction Speeds

n is the number of words transferred.

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

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

If the condition is not met all instructions take one S cycle. The cycle types (N, S, I and C) are defined in *Chapter 5.0 Memory Interface*.