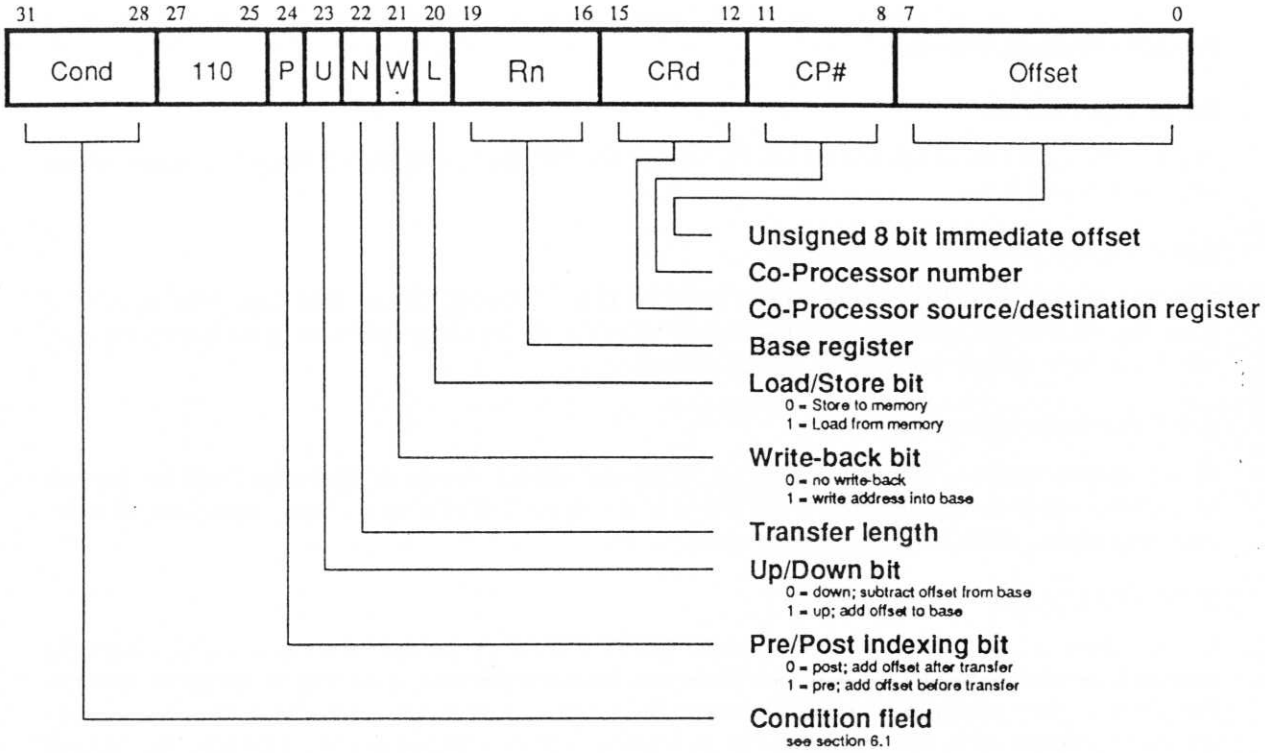
6.9 Co-Processor data transfers



The instruction is only executed if the condition is true. The various conditions are defined in section 6.1.

This class of instruction is used to transfer one or more words of data between the Co-Processor and main memory. ARM is responsible for supplying the memory address, and the Co-Processor supplies or accepts the data and controls the number of words transferred.

6.9.1 The Co-Processor fields

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

The CRd field and the N bit contain information for the Co-Processor which may be interpreted in different ways by different Co-Processors, 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.

6.9.2 Addressing modes

ARM 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 and specify word offsets here, whereas they are 12 bits and specify byte offsets for single data transfers.

An 8 bit unsigned immediate offset is scaled to words (ie shifted left 2 bits) and added to (U=1) or subtracted from (U=0) a base register (Rn), 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 when the address is pre-indexed, or by default if the address is post-indexed), or the old value of the base may be preserved (W=0 when the address is pre-indexed, or an offset of zero when the address is 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.

6.9.3 Use of R15

If Rn is R15, the value used will be the PC without the PSR flags, with the PC being the address of this instruction plus 8 bytes.

6.9.4 Forcing address translation

The W bit may be used in non-user mode programs (when the post-indexed addressing form is used) to force the $\overline{\text{TRANS}}$ pin LOW for the transfer cycle, allowing the operating system to generate user addresses when a suitable memory management system is present.

6.9.5 Address exceptions

If the address used for the first transfer is illegal the address exception mechanism will be invoked. Instructions which transfer multiple words will only trap if the first address is illegal; subsequent addresses will wrap around inside the 26 bit address space.

6.9.6 Data aborts

If the address is legal but the memory manager generates an abort the data abort trap will be taken. The writeback of the modified base will take place, but all other processor state will be preserved. The Co-Processor is partly responsible for ensuring restartability, and must either detect the abort or ensure that any actions consequent from this instruction can be repeated when the instruction is retried after the cause of the abort has been resolved.

6.9.7 Assembler syntax

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

LDC - load from memory to Co-Processor (L=1).

STC - store from Co-Processor to memory (L=0).

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

{cond} - two character condition mnemonic, see section 6.1.

CP# - the unique number of the required Co-Processor.

CRd is an expression evaluating to a valid Co-Processor register number.

<Address> can be:

* 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.

* A pre-indexed addressing specification:

  [Rn] offset of zero

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

* A post-indexed addressing specification:

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

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

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

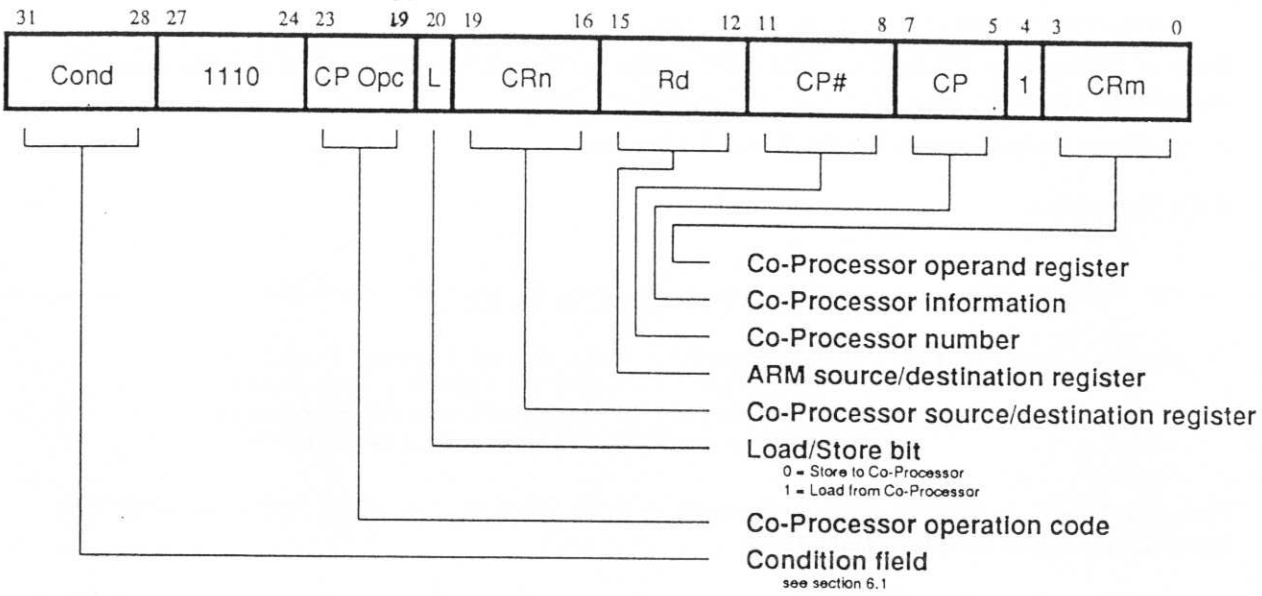
6.9.8 Examples

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

STCEQL 2,CR3,[R5,#24]! ; conditionally store CR3 of Co-Proc 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.

6.10 Co-Processor register transfers



The instruction is only executed if the condition is true. The various conditions are defined in section 6.1.

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

An important use of this instruction is to communicate control information directly from the Co-Processor into the ARM PSR flags. As an example, the result of a comparison of two floating point values within a Co-Processor can be moved to the PSR to control the subsequent flow of execution.

6.10.1 The Co-Processor fields

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

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

6.10.2 Transfers to R15

When a Co-Processor register transfer to ARM 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 PSR flags are unaffected by the transfer.

6.10.3 Transfers from R15

A Co-Processor register transfer from ARM with R15 as the source register will store the PC together with the PSR flags.

6.10.4 Assembler syntax

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

MCR - move from Co-Processor to ARM register (L=1).

MRC - move from ARM register to Co-Processor (L=0).

{cond} - two character condition mnemonic, see section 6.1.

CP# - the unique number of the required Co-Processor.

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

Rd is an expression evaluating to a valid ARM register number.

CRn and CRm are expressions evaluating to a valid Co-Processor register number.

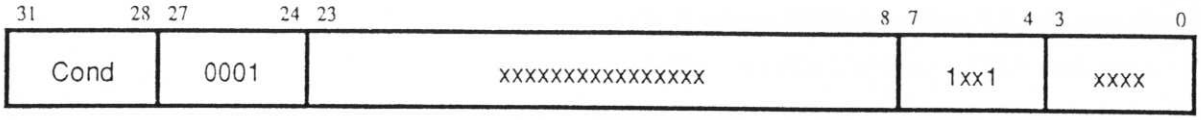
<expression2> - where present is evaluated to a constant and placed in the CP field.

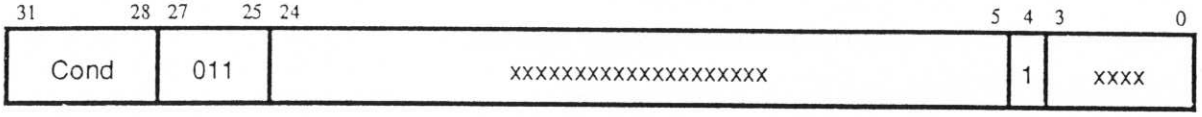
6.10.5 Examples

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

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

6.11 Undefined instructions





The instruction is only executed if the condition is true. The various conditions are defined in section 6.1.

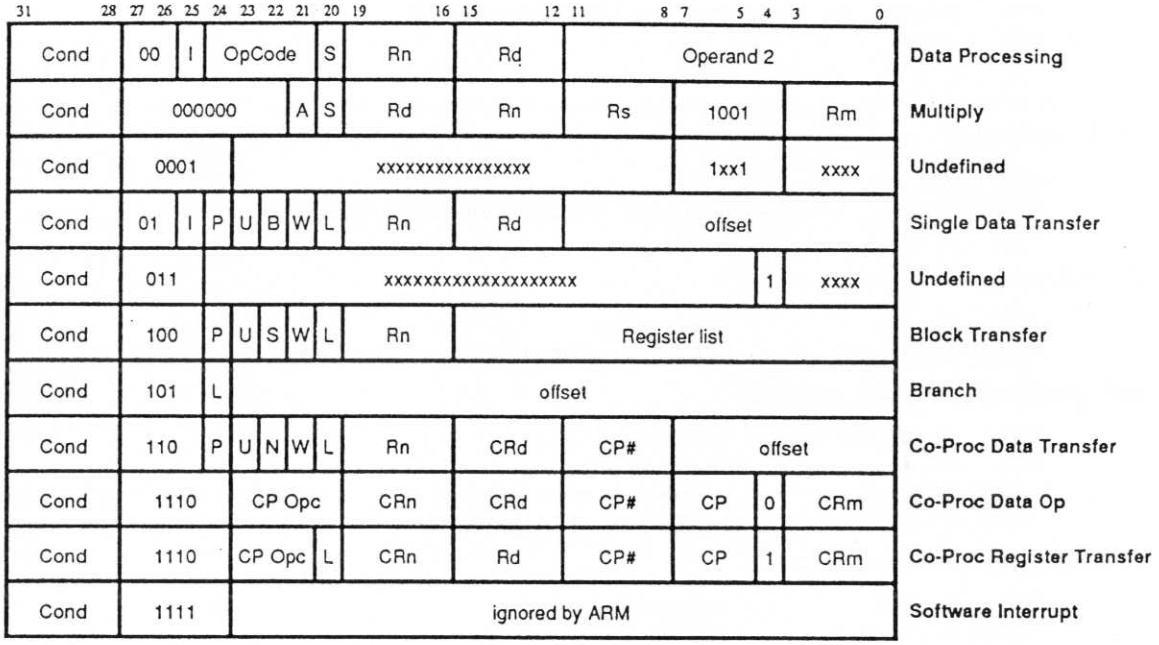
If the condition is true, the undefined instruction trap will be taken.

Note that the undefined instruction mechanism involves offering these instructions to any Co-Processors which may be present, and all Co-Processors must refuse to accept them by letting CPA float HIGH.

6.11.1 Assembler syntax

At present the assembler has no mnemonics for generating these instructions. If they are adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, these instructions should not be used.

6.12 Instruction set summary



(Note that 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 should be avoided, as their action may change in future ARM implementations.)

6.13 Instruction set examples

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

6.13.1 Using the conditional instructions

(1) using conditionals for logical OR

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

can be replaced by

```
CMP   Rn,#p
CMPNE Rm,#q  ;if condition not satisified 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
```

OUT OF DATE! (Works, but more efficient code possible)

6.13.2 Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition). Therefore BBC Basic 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. Then do this for all the newbits needed i.e. 32 of them. Luckily this can all be done in 5S cycles:

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

6.13.3 Multiplication by constant using the barrel shifter

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

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

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

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

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

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

(4) Multiplication by 6

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

(5) Multiply by 10 and add in extra number

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

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

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

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

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

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

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

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

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

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

rather than by:

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

6.13.4 Loading a word from an unknown alignment

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

6.13.5 Sign/zero extension of a half word

```
MOV   Ra,Ra,LSL #16   ;move to top
MOV   Ra,Ra,LSR #16   ;and back to bottom
                      ;use ASR to get
                      ;sign extended version
```

6.13.6 Return setting condition codes

```
BICS   PC,R14,#CFLAG  ;returns clearing C flag
                      ;from link register
ORRCCS PC,R14,#CFLAG  ;conditionally returns
                      ;setting C flag

; This code should not be used except in User mode
; since it will reset the interrupt enable flags to
; their value when R14 was set up.
; This generally applies to non-user mode programming,
; e.g. MOVS PC,R14. MOV PC,R14 is safer!
```

7. Memory Interface

ARM reads instructions and data from, and writes data to, its main memory via a 32 bit data bus. A separate 26 bit address bus specifies the memory location to be used for the transfer, and the $\overline{R}/W$ signal gives the direction of transfer (ARM to memory or memory to ARM). 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; they are in general much simpler than the DRAM interface described here.)

7.1 Cycle types

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

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

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

(4) Co-Processor register transfer. ARM wishes to use the data bus to communicate with a Co-Processor, but does not require any action by the memory system.

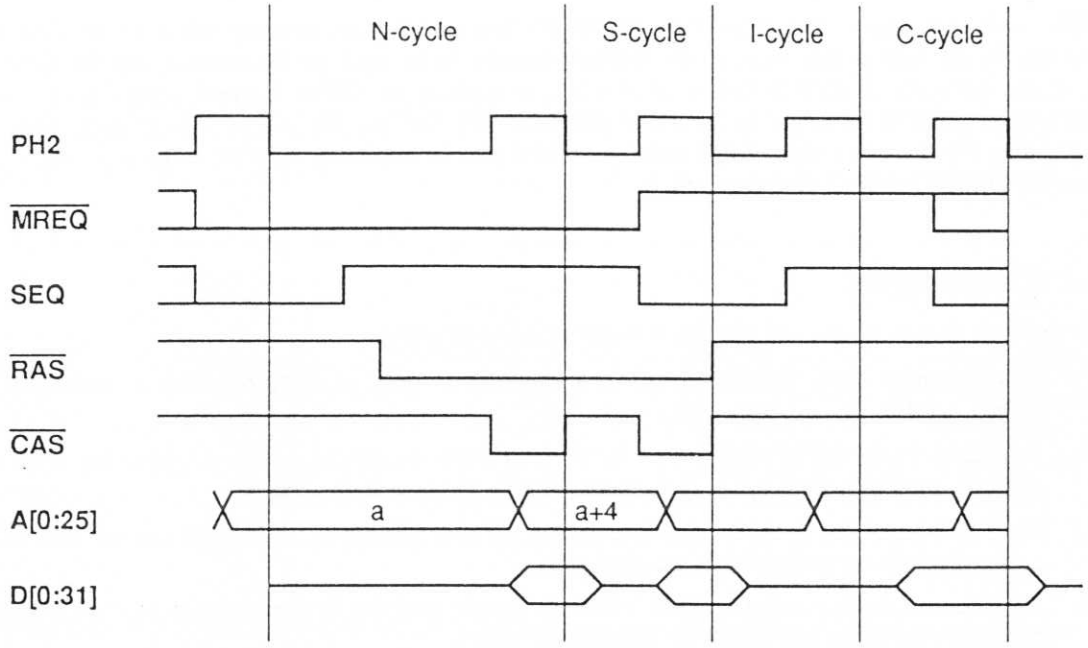
These four classes are distinguishable to the memory system by inspection of the $\overline{MREQ}$ and SEQ control lines (see table 1). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

$\overline{MREQ}$	SEQ	Cycle type	
0	0	Non-sequential cycle	(N-cycle)
0	1	Sequential cycle	(S-cycle)
1	0	Internal cycle	(I-cycle)
1	1	Co-Processor register transfer	(C-cycle)

Table 1: Memory cycle types

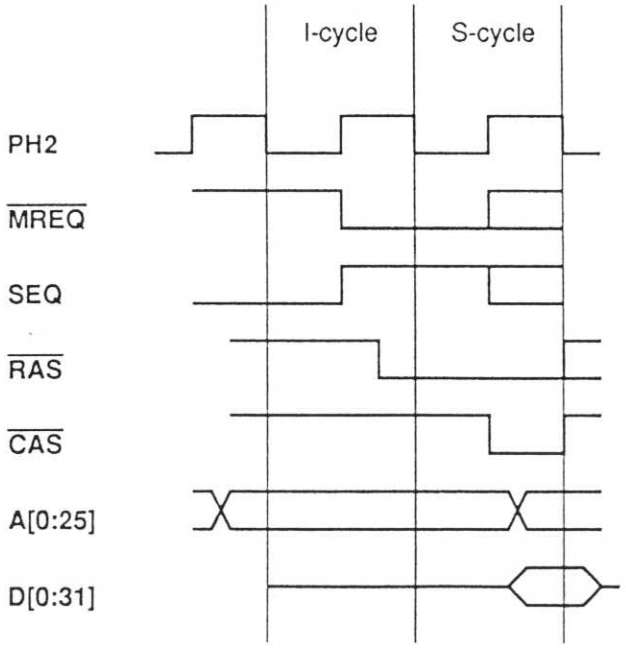
The following diagram shows the pipelining of the control signals, and suggests how the DRAM address strobes ($\overline{RAS}$ and $\overline{CAS}$) might be timed to use page mode for S-cycles. Note that the N-cycle is longer than the other cycles. This is to allow for the DRAM precharge and row access time, and is not an ARM

requirement.



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

When an S-cycle follows an I- or C-cycle, the address will be the same as that used in the I- or C-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to run at page mode speed whilst performing a full DRAM access:



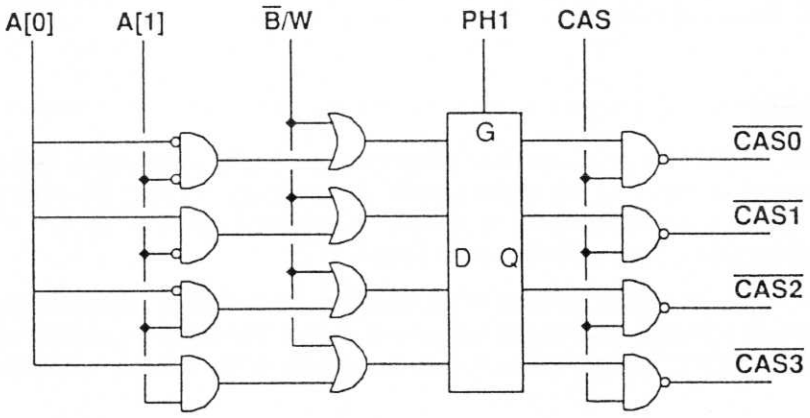
7.2 Byte addressing

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 $\overline{B}$/W 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 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. ARM will perform the byte extraction internally. Alternatively, the memory system may activate only the addressed byte of the memory. (This may be desirable in order to save power, or to enable the use of a common decoding system for both read and write cycles.)

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



$\overline{\text{CAS0}}$ drives the DRAM bank which is connected to **D[0:7]**, $\overline{\text{CAS1}}$ drives the bank connected to **D[8:15]**, 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.

7.3 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.

7.4 Memory management

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

* $\overline{\text{R}}$/W can be used by the memory manager to protect pages from being written to.
* $\overline{\text{OPC}}$ indicates that the word being fetched will be used as an instruction, so this signal can be used to control "execute only" pages.
* $\overline{\text{TRANS}}$ indicates whether the processor is in User or a non-user mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user. In the latter case, the T bit in LDR and STR instructions can be used to offer the supervisor the user's view of the memory.
* $\overline{\text{M}}$[0,1] can give the memory manager full information on the processor mode.

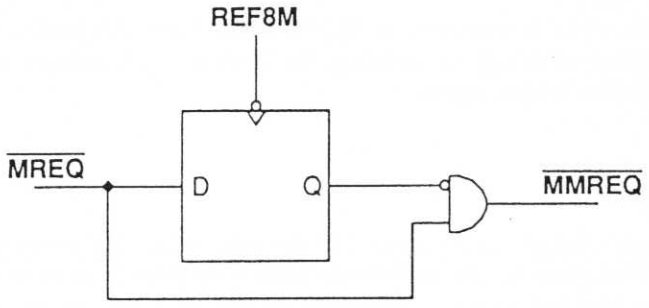
Address translation will normally only be necessary on an N-cycle, and this fact may be exploited to reduce power consumption in the memory manager and avoid the translation delay at other times.

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).

7.5 Use of MEMC

MEMC (VTI part number VL86C110) is an integrated memory controller for ARM which incorporates an address translation system and generates all the critical system timing signals. N-cycles run at up to 4 MHz (including translation), and S- and I-cycles at up to 8 MHz. It interfaces ARM to up to 4 MBytes of DRAM, and allows for video DMA, system ROM and IO devices.

MEMC was specified to support all the functionality of the prototype ARM devices, which did not have the multiply instruction or the Co-Processor interface of the present design. The following logic is required to modify the $\overline{\text{MREQ}}$ signal to MEMC in order to ensure correct operation of the multiply instruction:



This logic modifies $\overline{\text{MREQ}}$ from the ARM to produce $\overline{\text{MMREQ}}$, which is connected to the $\overline{\text{MREQ}}$ input on MEMC. **REF8M** is a MEMC output. Without this change, MEMC may merge the first instruction prefetch following a multiply with the last cycle of a DMA or refresh operation, which will cause the instruction to be fetched from the incorrect address.

(If Co-Processor capability is required, further logic must be added to modify the behaviour of MEMC. This is described in the next chapter.)

8. Co-Processor Interface

The functionality of the ARM instruction set may be extended by the addition of up to 16 external Co-Processors. When the Co-Processor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the Co-Processor will then increase the system performance in a software compatible way.

8.1 Interface signals

Three dedicated signals control the Co-Processor interface, $\overline{\text{CPI}}$, **CPA** and **CPB**. An external pull-up resistor is normally required on both **CPA** and **CPB**.

8.1.1 Co-Processor present/absent

ARM takes $\overline{\text{CPI}}$ LOW whenever it starts to execute a Co-Processor (or undefined) instruction. (This will not happen if the instruction fails to be executed because of the condition codes.) Each Co-Processor will have a copy of the instruction, and can inspect the CP# field to see which Co-Processor it is for. Every Co-Processor in a system must have a unique number, and if that number matches the contents of the CP# field, the Co-Processor should pull the **CPA** (Co-Processor absent) line LOW. If no Co-Processor has a number which matches the CP# field, **CPA** will float HIGH, and ARM will take the undefined instruction trap. Otherwise ARM observes the **CPA** line going LOW, and waits until the Co-Processor is not busy.

8.1.2 Busy-waiting

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

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

All three interface signals are sampled by both ARM and the Co-Processor(s) on the rising edge of **PH2**. If all three are LOW, the instruction is committed to execution, and if transfers are involved they will start on the next cycle. If $\overline{\text{CPI}}$ has gone HIGH after being LOW, and before the instruction is committed, ARM has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other Co-Processor instructions may come sooner, and the instruction should be discarded.

8.1.3 Pipeline following

In order to respond correctly when a Co-Processor instruction arises, each Co-Processor must have a copy of the instruction. All ARM instructions are fetched from memory via the main data bus, and Co-Processors are connected to this bus, so they can keep copies of all instructions as they go into the ARM pipeline. The $\overline{\text{OPC}}$ signal indicates when an instruction fetch is taking place, and **PH2** gives the timing of the transfer, so these may be used together to load an instruction pipeline within the Co-Processor.

8.2 Data transfer cycles

Once the Co-Processor has gone not-busy in a data transfer instruction, it must supply or accept data at the ARM bus rate (defined by **PH2**). 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 Co-Processor is responsible for determining the number of words to be transferred; ARM will continue to increment the address by one word per transfer until the Co-Processor tells it to stop. The termination condition is indicated by the Co-Processor releasing **CPA** and **CPB** to float HIGH.

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

8.3 Register transfer cycle

The Co-Processor register transfer cycle is the one case when ARM requires the data bus without requiring the memory to be active.

The memory system is informed that the bus is required by ARM taking both $\overline{\text{MREQ}}$ and **SEQ** HIGH. When the bus is free, **DBE** should be taken HIGH to allow ARM or the Co-Processor to drive the bus, and a **PH2** cycle times the transfer.

8.4 Privileged instructions

The Co-Processor may restrict certain instructions for use in supervisor mode only. To do this, the Co-Processor will have to track either the $\overline{\text{TRANS}}$ pin or the $\overline{\text{M}}$[0,1] pins.

As an example of the use of this facility, consider the case of a floating point Co-Processor (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.

8.5 Idempotency

A consequence of the implementation of the Co-Processor interface, with the interruptable busy-wait state, is that all instructions may be interrupted at any point up to the time when the Co-Processor 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 Co-Processor 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 Co-Processor which returns the integer result to an ARM register. The Co-Processor must stay busy while it performs the floating point to fixed point conversion, as ARM will expect to receive the integer value on the cycle immediately following that where it goes not-busy. The Co-Processor 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.

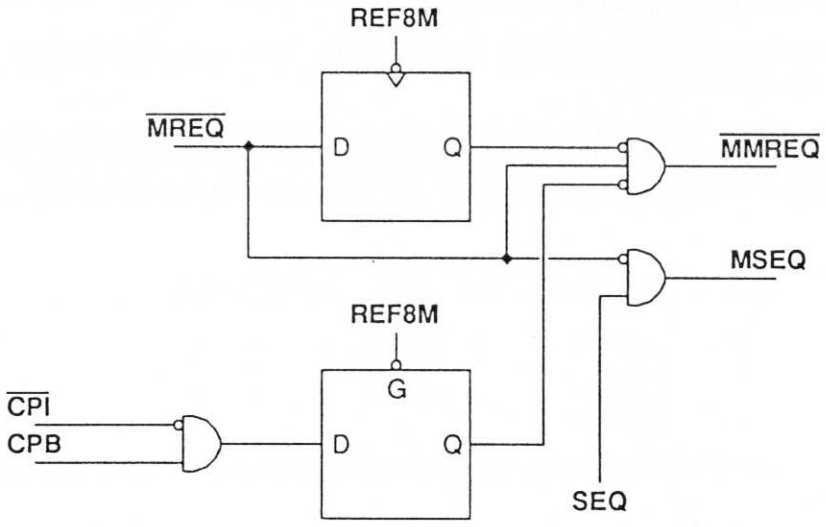
The Co-Processor data operation class of instruction is not generally subject to idempotency considerations, as the processing activity can take place after the Co-Processor goes not-busy. There is no need for ARM to be held up until the result is generated, because the result is confined to stay within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

8.6 Undefined instructions

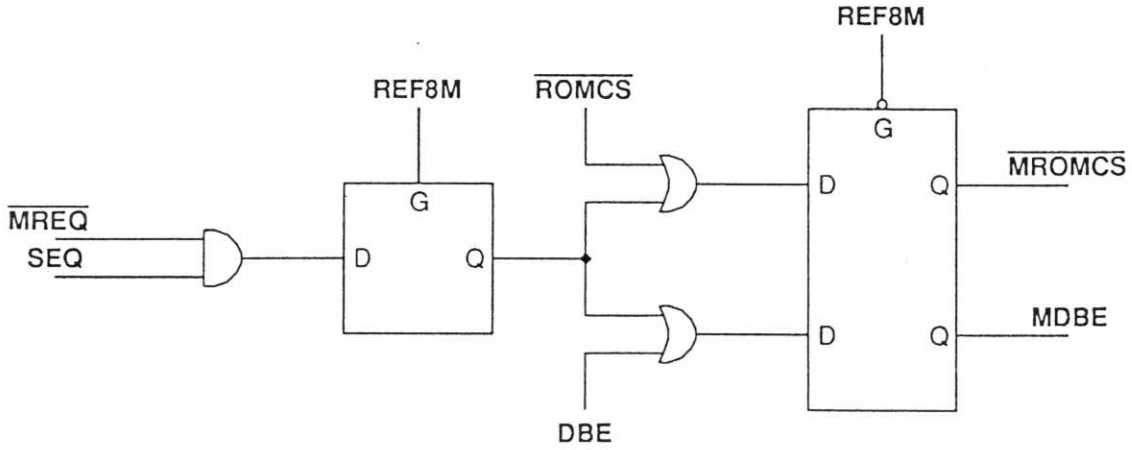
Undefined instructions are treated by ARM as Co-Processor instructions. All Co-Processors must be absent (ie let CPA float HIGH) when an undefined instruction is presented. ARM will then take the undefined instruction trap. Note that the Co-Processor need only look at bit 27 of the instruction to differentiate undefined instructions (which all have 0 in bit 27) from Co-Processor instructions (which all have 1 in bit 27).

8.7 Use of MEMC

The $\overline{\text{MREQ}}$ and SEQ signals from ARM must be modified before being presented to MEMC in order to ensure that the data bus is free for a register transfer should that be required. The following circuit performs the required operation, and incorporates the logic required to ensure that multiplies work:



In addition, ARM or the Co-Processor must be allowed to drive the data bus for the register transfer cycle, and the system ROM must be prevented from driving the bus at the same time:



(These two modifications can be fitted into one 16L8A PAL.)

9. Instruction Cycle Operations

In the following tables $\overline{\text{MREQ}}$ and SEQ (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the address of the next cycle. The address, $\overline{\text{B}}$/W, $\overline{\text{R}}$/W, and $\overline{\text{OPC}}$ (which appear up to half a cycle ahead) are shown in the cycle to which they apply.

9.1 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.

Cycle	address	$\overline{\text{b}}$/w	$\overline{\text{r}}$/w	data	seq	$\overline{\text{mreq}}$	$\overline{\text{opc}}$
1	pc+8	1	0	(pc+8)	0	0	0
2	alu	1	0	(alu)	1	0	0
3	alu+4	1	0	(alu+4)	1	0	0
	alu+8						

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

9.2 Data operations

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

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

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (ie will not request memory). This internal cycle is configured to merge with the next cycle into a single memory N-cycle when MEMC is used as the memory interface.

The PC may be any (or all!) of the register operands. When read onto the A bus it appears without the PSR bits, on the B bus it appears with them. Neither will affect external bus activity. When it is the destination, however, 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.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$
normal	1	pc+8	1	0	(pc+8)	1	0	0
		pc+12						
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	1	0	0
	3	alu+4	1	0	(alu+4)	1	0	0
		alu+8						
shift(Rs)	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	1	0	1
		pc+12						
shift(Rs), dest=pc	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	0	0	1
	3	alu	1	0	(alu)	1	0	0
	4	alu+4	1	0	(alu+4)	1	0	0
		alu+8						

9.3 Multiply and multiply accumulate

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

The datapath then cycles, adding the second operand to, subtracting it from, or just transmitting, the result register. The second operand is shifted in the Nth cycle by 2N or 2N+1 bits, under control of the Booth's logic. The first operand 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.

If the destination is the PC, all writing to it is prevented. The instruction will proceed as normal except that the PC will be unaffected. (If the S bit is set the PSR flags will be meaningless.)

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$
(Rs)=0,1	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	1	0	1
		pc+12						
(Rs)>1	1	pc+8	1	0	(pc+8)	0	1	0
	2	pc+12	1	0	-	0	1	1
	.	pc+12	1	0	-	0	1	1
	m	pc+12	1	0	-	0	1	1
	m+1	pc+12	1	0	-	1	0	1
		pc+12						

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

9.4 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.

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 base and destination modifications are prevented.

	Cycle	address	$\overline{b/w}$	$\overline{r/w}$	data	seq	$\overline{mreq}$	opc	$\overline{trans}$
normal	1	pc+8	1	0	(pc+8)	0	0	0	
	2	alu	b/w	0	(alu)	0	1	1	t
	3	pc+12	1	0	-	1	0	1	
		pc+12							
dest=pc	1	pc+8	1	0	(pc+8)	0	0	0	
	2	alu	b/w	0	(alu)	0	1	1	t
	3	pc+12	1	0	-	0	0	1	
	4	(alu)	1	0	((alu))	1	0	0	
	5	(alu)+4	1	0	((alu)+4)	1	0	0	
		(alu)+8							
base=pc,	1	pc+8	1	0	(pc+8)	0	0	0	
write-back	2	alu	b/w	0	(alu)	0	1	1	t
dest≠pc	3	pc'	1	0	-	0	0	1	
	4	pc'	1	0	(pc')	1	0	0	
	5	pc'+4	1	0	(pc'+4)	1	0	0	
		pc'+8							
base=pc,	1	pc+8	1	0	(pc+8)	0	0	0	
write-back	2	alu	b/w	0	(alu)	0	1	1	t
dest=pc	3	pc'	1	0	-	0	0	1	
	4	(alu)	1	0	((alu))	1	0	0	
	5	(alu)+4	1	0	((alu)+4)	1	0	0	
		(alu)+8							

(pc' is the PC value modified by write-back; t shows the cycle where the force translation option in the instruction may be used.)

9.5 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 PC will only be modified if it is the base and write-back occurs.

A data abort prevents the base write-back.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{trans}$
normal	1	pc+8	1	0	(pc+8)	0	0	0	
	2	alu pc+12	b/w	1	Rd	0	0	1	t
base=pc, write-back	1	pc+8	1	0	(pc+8)	0	0	0	
	2	alu	b/w	1	Rd	0	0	1	t
	3	pc'	1	0	(pc')	1	0	0	
	4	pc'+4 pc'+8	1	0	(pc'+4)	1	0	0	

9.6 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.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	1	Ra	0	0	1
n registers (n>1)	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	1	Ra	1	0	1
	.	alu+.	1	1	R.	1	0	1
	n	alu+.	1	1	R.	1	0	1
	n+1	alu+.	1	1	R.	0	0	1

9.7 Load multiple registers

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

The 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).

If the PC is the base, write-back is prevented.

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

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

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$
1 register	1	pc+8	1	0	(pc+8)	0	0	0
	2	alu	1	0	(alu)	0	1	1
	3	pc+12	1	0	-	1	0	1
		pc+12						
1 register	1	pc+8	1	0	(pc+8)	0	0	0
dest=pc	2	alu	1	0	pc'	0	1	1
	3	pc+12	1	0	-	0	0	1
	4	pc'	1	0	(pc')	1	0	0
	5	pc'+4	1	0	(pc'+4)	1	0	0
		pc'+8						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	0	(alu)	1	0	1
	.	alu+.	1	0	(alu+.)	1	0	1
	n	alu+.	1	0	(alu+.)	1	0	1
	n+1	alu+.	1	0	(alu+.)	0	1	1
	n+2	pc+12	1	0	-	1	0	1
		pc+12						
n registers	1	pc+8	1	0	(pc+8)	0	0	0
(n>1)	2	alu	1	0	(alu)	1	0	1
incl. pc	.	alu+.	1	0	(alu+.)	1	0	1
	n	alu+.	1	0	(alu+.)	1	0	1
	n+1	alu+.	1	0	pc'	0	1	1
	n+2	pc+12	1	0	-	0	0	1
	n+3	pc'	1	0	(pc')	1	0	0
	n+4	pc'+4	1	0	(pc'+4)	1	0	0
		pc'+8						

9.8 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 register 14.

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.

Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{trans}$
1	pc+8	1	0	(pc+8)	0	0	0	1
2	Xn	1	0	(Xn)	1	0	0	1
3	Xn+4	1	0	(Xn+4)	1	0	0	1
	Xn+8							

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

9.9 Co-Processor data operation

A Co-Processor data operation is a request from ARM for the Co-Processor to initiate some action. The action need not be completed for some time, but the Co-Processor must commit to doing it before pulling **CPB** LOW.

If the Co-Processor can never do the requested task, it should leave **CPA** and **CPB** to float HIGH. If it can do the task, but can't commit right now, it should pull **CPA** LOW but leave **CPB** HIGH until it can commit. ARM will busy-wait until **CPB** goes LOW.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{cpi}$	cpa	cpb
ready	1	pc+8	1	0	(pc+8)	1	0	0	0	0	0
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0

9.10 Co-Processor data transfer (from memory to Co-Processor)

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

ARM 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.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{cpi}$	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)	0	1	0	0	0	1
not ready	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	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)	1	0	1	1	0	0
ready	.	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+1	alu+.	1	0	(alu+.)	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
(m>1)	2	pc+8	1	0	-	0	1	1	0	0	1
not ready	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	0	(alu)	1	0	1	1	0	0
ready	.	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+m	alu+.	1	0	(alu+.)	1	0	1	1	0	0
	n+m+1	alu+.	1	0	(alu+.)	0	0	1	1	1	1
		pc+12									

9.11 Co-Processor data transfer (from Co-Processor to memory)

The ARM controls these instructions exactly as for memory to Co-Processor transfers, with the one exception that the $\overline{R}$/W line is inverted during the transfer cycle.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	opc	$\overline{cpi}$	cpa	cpb
1 register	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
ready	2	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
1 register	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
not ready	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	1	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	1	1	CPdata	1	0	1	1	0	0
ready	.	alu+.	1	1	CPdata	1	0	1	1	0	0
	n	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+1	alu+.	1	1	CPdata	0	0	1	1	1	1
		pc+12									
m registers	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
(m>1)	2	pc+8	1	0	-	0	1	1	0	0	1
not ready	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	0	0	1	0	0	0
	n+1	alu	1	1	CPdata	1	0	1	1	0	0
ready	.	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+m	alu+.	1	1	CPdata	1	0	1	1	0	0
	n+m+1	alu+.	1	1	CPdata	0	0	1	1	1	1
		pc+12									

9.12 Co-Processor register transfer (Load from Co-Processor)

Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and ARM 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 ARM register load instructions.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	opc	$\overline{cpi}$	cpa	cpb
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	0	CPdata	0	1	1	1	1	1
	3	pc+12	1	0	-	1	0	1	1	-	-
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	0	CPdata	0	1	1	1	1	1
	n+2	pc+12	1	0	-	1	0	1	1	-	-
		pc+12									

9.13 Co-Processor register transfer (Store to Co-Processor)

As for the load from Co-Processor, except that the last cycle is omitted.

	Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{cpi}$	cpa	cpb
ready	1	pc+8	1	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	1	1	Rd	1	0	1	1	1	1
		pc+12									
not ready	1	pc+8	1	0	(pc+8)	0	1	0	0	0	1
	2	pc+8	1	0	-	0	1	1	0	0	1
	.	pc+8	1	0	-	0	1	1	0	0	1
	n	pc+8	1	0	-	1	1	1	0	0	0
	n+1	pc+12	1	1	Rd	1	0	1	1	1	1
		pc+12									

9.14 Undefined instructions and Co-Processor absent

When a Co-Processor detects a Co-Processor instruction which it cannot perform, and this must include all undefined instructions, it must not drive CPA or CPB. These will float HIGH, causing the undefined instruction trap to be taken.

Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$	$\overline{cpi}$	cpa	cpb
1	pc+8	1	0	(pc+8)	0	1	0	0	1	1
2	pc+8	1	0	-	0	0	0	1	1	1
3	Xn	1	0	(Xn)	1	0	0	1	1	1
4	Xn+4	1	0	(Xn+4)	1	0	0	1	1	1
	Xn+8									

9.15 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.

Cycle	address	$\overline{b}$/w	$\overline{r}$/w	data	seq	$\overline{mreq}$	$\overline{opc}$
1	pc+8	1	0	(pc+8)	1	0	0
	pc+12						

9.16 Instruction speeds

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

If the condition is met the instructions take:

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

n is the number of words transferred.

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between 2^(2m-3) and 2^(2m-1)-1 inclusive takes m cycles for m>1. Multiplication by 0 or 1 takes 1 cycle. The maximum value m can take is 16.

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

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

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

10. DC Parameters

10.1 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	deg.C	1

NOTE:

(1) These are stress ratings only. Exceeding the absolute maximum ratings may permanantly damage the device. Operating the device at absolute maximum ratings for extended periods may affect device reliability.

10.2 DC Operating Conditions

Symbol	Parameter	Min	Typ	Max	Units	Note
VDD	Supply voltage	4.75	5.0	5.25	V	
Viht	IT input HIGH voltage	2.4		VDD	V	1,2
Vilt	IT input LOW voltage	0.0		0.8	V	1,2
Vihz	IOTZ input HIGH voltage	2.4		VDD	V	1,3
Vilz	IOTZ input LOW voltage	0.0		0.8	V	1,3
Vihk	ICk input HIGH voltage	VDD -0.3		VDD	V	1,4
Vilk	ICk input LOW voltage	0.0		0.3	V	1,4
Ta	Ambient operating temperature	0		70	deg.C	

NOTES:

(1) Voltages measured with respect to VSS.

(2) IT - TTL compatible inputs.

(3) IOTZ - Bi-directional 3-state inputs.

(4) ICk - Unbuffered clock inputs (PH1, ph2).

10.3 DC Characteristics

KEY

Mes - Values measured from a sample ARM
Nom - Nominal values derived from analogue simulations with VDD=5v, 100 deg.C.

Symbol	Parameter	Mes	Nom	Units	Note
IDD	Supply current	21		mA	
|Isc|	Output short circuit current	>22		mA	1
|Ilu|	D.C. latch-up current	>250		mA	2
|Iin|	input leakage current		10	uA	
Vohc	output HIGH voltage (CMOS)		4.6	V	3,6
Volc	output LOW voltage (CMOS)		0.29	V	3,7
Voht	output HIGH voltage (TTL)		3.8	V	4,8
Volt	output LOW voltage (TTL)		0.8	V	4,9
Vihtt	input HIGH voltage (TTL) threshold	1.85	2.1	V	5
Viltt	input LOW voltage (TTL) threshold	1.85	1.4	V	5
Cin	Input capacitance		5	pF	

NOTES:

(1) Not more than one output should be shorted to either rail at any time, and for no longer than 1 second.

(2) This value represents the D.C. current that the input/output pins can tolerate before the chip latches up.

(3) OC - CMOS compatible outputs.

OCZ - bi-directional 3-state outputs.

(4) IOTZ - TTL compatible bi-directional 3-state outputs.

(5) IT - TTL compatible inputs.

IOTZ - TTL compatible bi-directional 3-state inputs.

(6) Output current = 3mA

(7) Output current = -3mA

(8) Output current = 10mA

(9) Output current = -10mA

11. AC Parameters

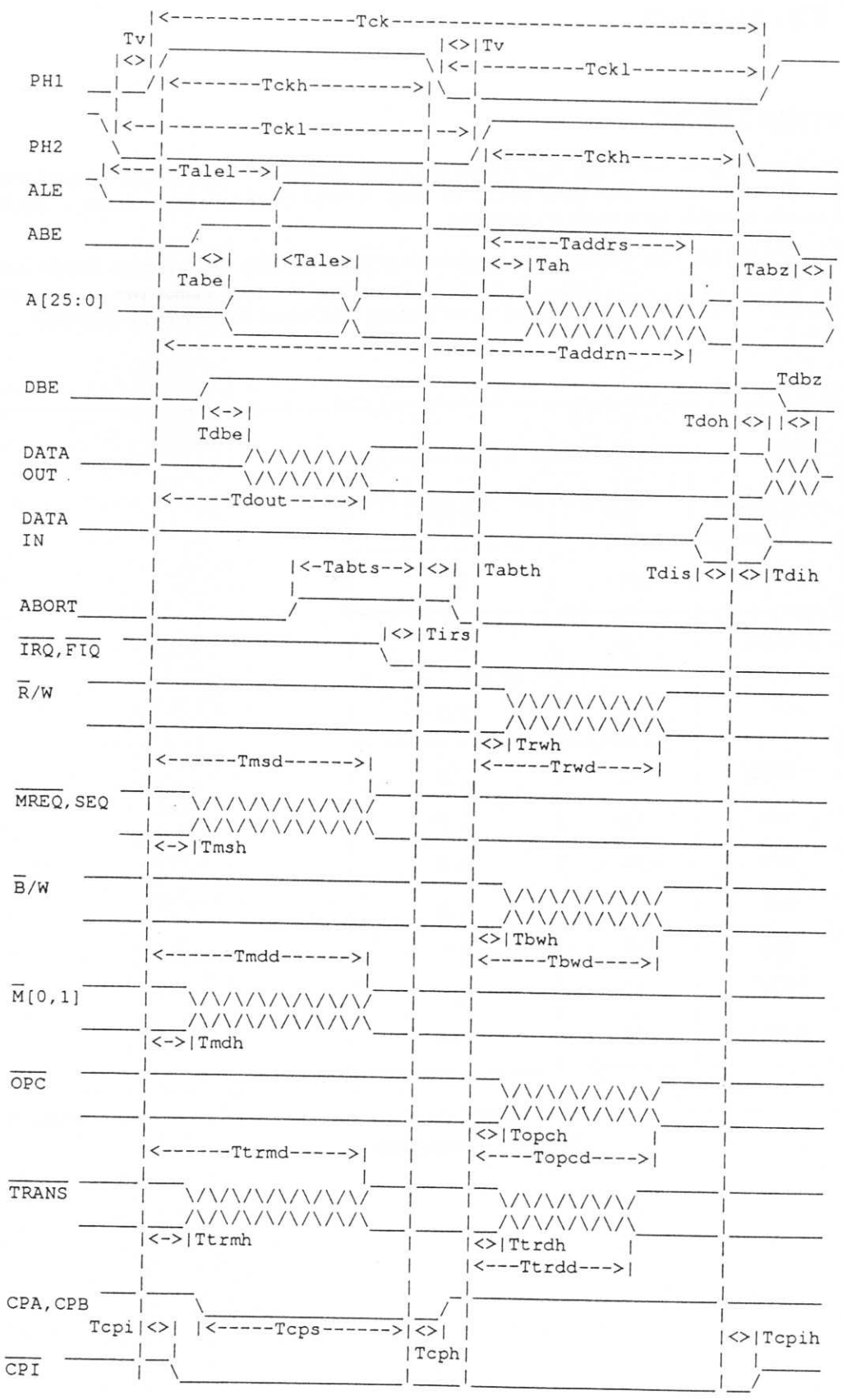
TEST CONDITIONS

The AC timing diagrams presented in this section assume that the outputs of ARM have been loaded with the capacitive loads shown in the 'Test Load' column of Table 2; these loads have been chosen as typical of the type of system in which ARM might be employed.

The output pads of ARM 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 increase in load capacitance necessary to increase the total output time by one nanosecond.

Output Signal	Test Load (pF)	Output derating (pF/ns)
D[0:31]	100	5
A[0:25]	50	5
$\overline{\text{CPI}}$	50	2.5
$\overline{\text{MREQ}}$	15	2.5
SEQ	15	2.5
$\overline{\text{R/W}}$	15	2.5
$\overline{\text{B/W}}$	15	2.5
$\overline{\text{OPC}}$	15	2.5
$\overline{\text{TRANS}}$	15	2.5
$\overline{\text{M}}$[0,1]	15	2.5

Table 2: AC test loads


Tck
Tv
PH1
Tckh
Tckl
PH2
Talel
ALE
ABE
Tale
Tabe
A[25:0]
Taddrs
Tah
Tabz
Taddrn
DBE
Tdbz
Tdbe
Tdoh
DATA OUT
Tdout
DATA IN
Tabts
Tabth
Tdis
Tdih
ABORT
IRQ, FIQ
Tirs
R/W
Trwh
Trwd
MREQ, SEQ
Tmsd
Tmsh
B/W
Tbwh
Tbwd
M[0,1]
Tmdd
Tmdh
OPC
Topch
Topcd
TRANS
Ttrmd
Ttrmh
Ttrdh
Ttrdd
CPA, CPB
Tcpi
Tcps
Tcph
Tcpih
CPI

Symbol	Parameter	Min	Mes	Max	Unit	Conditions
Tv	clock non-overlap	0	5			measured at 1 volt level
Tck	clock period	100		10000	ns	
Tckl	clock LOW time	45	50	10000	ns	
Tckh	clock HIGH time	45	50	10000	ns	
Tabe	address bus enable			30	ns	
Tabz	address bus disable			30	ns	
Tale	address latch open			30	ns	
Talel	ALE low time			10000	ns	Note 1
Taddrs	ph2 to address valid		20	35	ns	Note 2
Taddrn	ph1 to address valid			105	ns	
Tah	address hold time	5			ns	
Tdbe	data bus enable			45	ns	(TTL level)
Tdbz	data bus disable			45	ns	
Tdout	data out delay		25	55	ns	(TTL level)
Tdoh	data out hold	5			ns	
Tdis	data in setup	5	3		ns	
Tdih	data in hold	10	4		ns	
Tabts	abort setup time	40	22		ns	
Tabth	abort hold time	5	0			
Tirs	interrupt setup	10			ns	Note 3
Trwd	ph2 to $\overline{r}$/w valid		25	55	ns	Note 4
Trwh	r/w hold time	5				
Tmsd	ph1 to $\overline{mreq}$ & seq		20	55	ns	
Tmsh	mreq & seq hold time	5				
Tbwd	ph2 to $\overline{b}$/w valid		25	40	ns	
Tbwh	b/w hold time	5				
Tmdd	ph1 to $\overline{m}$[0,1] valid		15	30	ns	
Tmdh	m[0,1] hold time	5				
Topcd	ph2 to $\overline{opc}$ valid		25	35	ns	
Topch	opc hold time	5				
Ttrmd	ph1 to $\overline{trans}$ valid		20	35	ns	mode change
Ttrmh	trans hold time	5			ns	
Ttrdd	ph2 to $\overline{trans}$ valid			45	ns	Note 5
Trrdh	trans hold time	5				
Tcps	cpa, cpb setup	40			ns	
Tcph	cpa, cpb hold time	5			ns	
Tcpi	ph1 to $\overline{cpi}$ delay			25	ns	
Tcpih	cpi hold time	5			ns	

Notes:

(1) **ALE** controls a dynamic storage latch; this parameter is specified to ensure that the stored charge cannot leak sufficiently to generate intermediate logic levels in the associated logic.

(2) The **PH1** to address delay only applies to non-sequential cycles, when the address is being calculated in the ALU. For sequential cycles the address will be valid earlier, at the given time from **PH2**. (Taddrs applies to sequential and non-sequential cycles.)

(3) The interrupt and reset inputs may be asynchronous. This time will guarantee that the interrupt request is latched during this cycle.

(4) The worst case for $\overline{R}/W$ only arises when an address exception occurs on a data store operation. The address exception causes $\overline{R}/W$ to switch to read to prevent erroneous writing of memory.

(5) $\overline{\text{TRANS}}$ will only change during **PH2** as the result of a 'force translate' single data transfer operation whilst in a non-user mode. Otherwise it will change during **PH1** when a mode change to or from user mode takes place.

General notes on AC parameters:

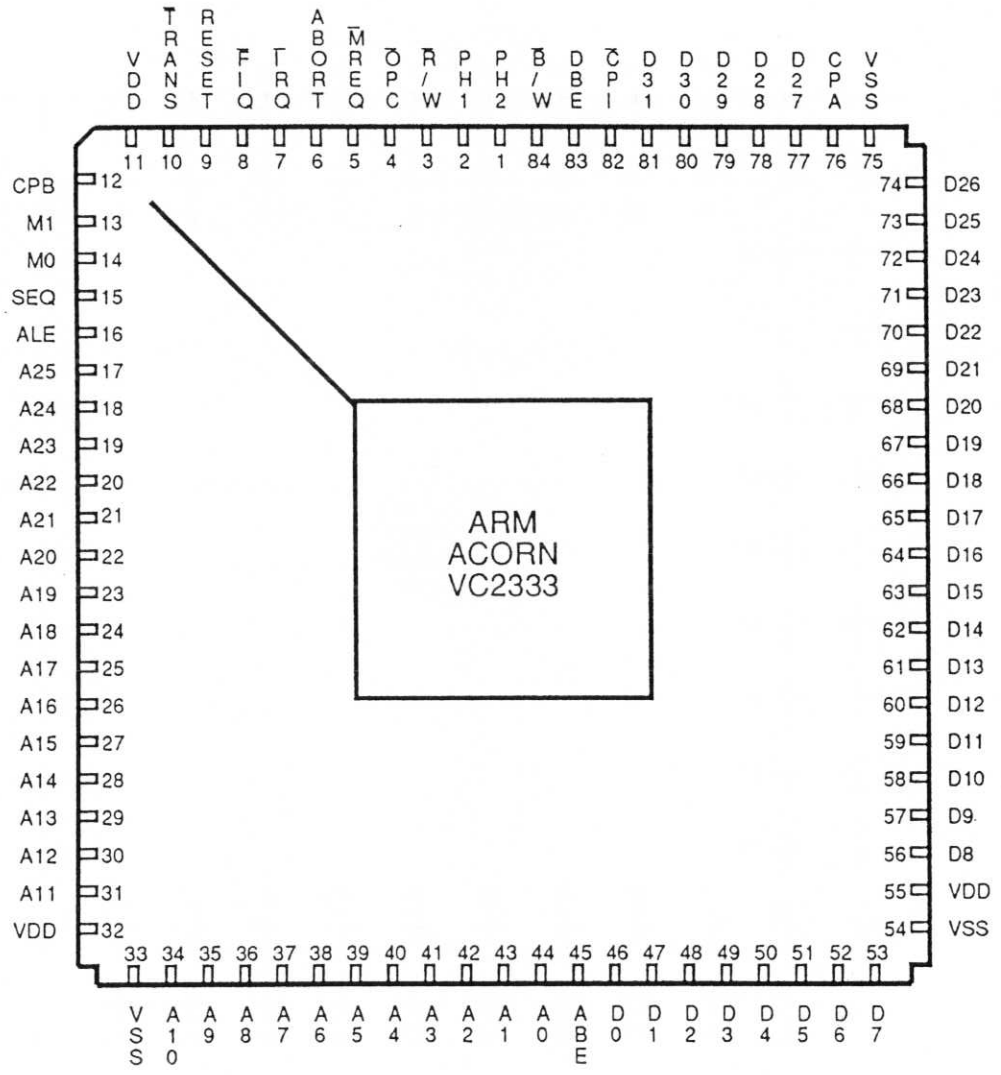
* The 'Min' and 'Max' times are not measured. The maximum delays are derived from SPICE models of the relevant logic functions, with VTI slow-slow transistor models, **VDD**=4.7 volts, **VSS**=0.1 volts, temperature 100 degrees Centigrade. The minimum hold times are calculated from the same models of the relevant paths, with the time in the table being the slow path time divided by four. All numbers have been rounded to the nearest 5 ns. All numbers are subject to change after device characterisation.

* The 'Mes' times were measured on a sample ARM at room temperature with **VDD**=5v.

* Output times are to CMOS levels except for the data bus, which is to TTL levels.

12. Packaging

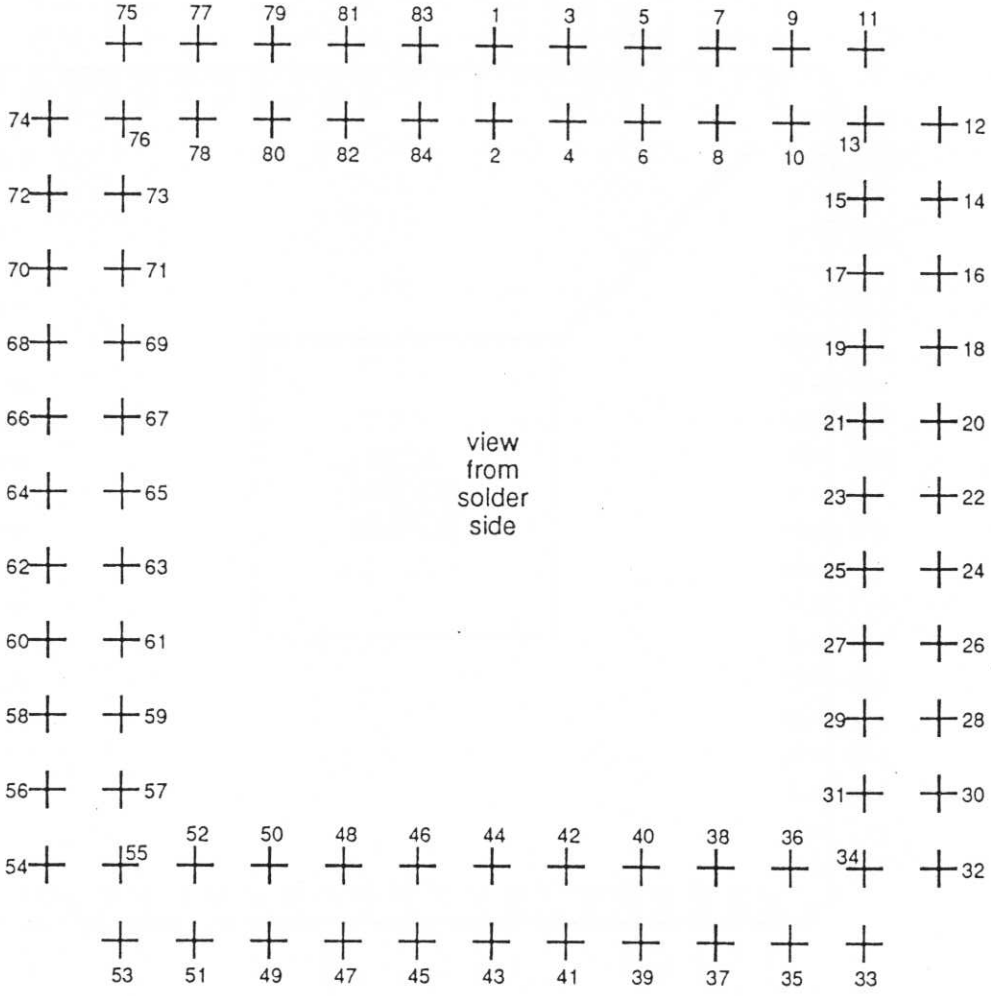
The ARM is packaged in an 84-pin JEDEC B ceramic leadless chip carrier, or a JEDEC C plastic leaded carrier (PLCC).



Suitable sockets for the devices are:

(i) AMP 55225-1 for the ceramic leadless chip carrier.

(ii) Burndy QILE84P-410T for the plastic leaded chip carrier (PLCC).



13. Compatibility with Prototype ARMs

13.1 Plug-in compatibility

This ARM chip has been designed to plug into boards built to accept the prototype ARM devices as far as possible.

The only pin changes between the prototype ARMs and the present devices are the additional pins to support the Co-Processor interface. $\overline{\text{CPI}}$ uses a previously unused pin, and so long as this is not connected no problem will arise. **CPA** and **CPB** use pins which were previously allocated to power and ground connections, and if so connected will cause all Co-Processors to appear absent, which is almost certainly the case for prototype ARM boards! **CPB** will, however, be connected to ground rather than 5v, which will cause the return address from the undefined instruction trap to be wrong. If the indefined instruction trap is to be used, for instance for floating point emulation, **CPB** should be tied high.

Signal timings are in general considerably faster than they were on the prototype devices, and this could cause problems in some circuits.

13.2 Bug fixes

Various problems on the prototype chips have been fixed.

The following bugs affected all programmers:

* RRX produced carry out equal to the inclusive OR of bits 0 and 31, thus giving the wrong answer when bit 31 = 1 and bit 0 = 0.
* ROR by a register which is 32*n, n <> 0, produced carry out equal to zero instead of bit 31.
* LDM of one register within the last 3 words of a page where the next page is inaccessible appeared to 'Data Abort' even though the load was legal.

The following bug affected only non-user mode programmers:

* Load multiple of a single register with PSR update requested using a mode specific register (eg R13) as base could writeback to the wrong register bank (most probably the user R13 was corrupted). Thus return from supervisor or interrupt code by LDMFD R13!,{PC}^ had to be avoided.

The following bug affected only hardware designers:

* An abort signalled during an internal cycle where ARM had not requested memory still caused the processor to abort. (This problem did not arise in MEMC based systems, since MEMC is prevented from producing an abort under such circumstances.)

13.3 Design differences

The following design changes affect all programmers:

* Multiply and multiply accumulate instructions have been added.
* The Co-Processor interface has been added, using what was previously undefined instruction space. Note that undefined instructions which fail on condition code will now be skipped, whereas previously they trapped.

* LDR, STR with the register offset shifted by a register specified amount have been removed, and are now undefined instructions.
* The data processing instruction space has been more fully decoded, and previously redundant codes are now used for multiply and undefined instructions.

The following changes affect only non-user mode programmers:

* Two extra FIQ registers have been added.
* LDM of user registers from supervisor mode has been added.

The following changes affect only hardware designers:

* Some cycle control signal states have been modified.
* The instruction pipeline now uses static logic, as do some other latches, in order to allow for the Co-Processor busy-wait state.
* The new chip will go faster, and all the control signal timings should change accordingly.
* The multiply and Co-Processor operations are incompatible with MEMC, and some external logic is required for correct functioning. (This is detailed in the chapters on the memory and Co-Processor interfaces.)