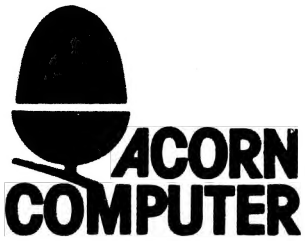
Acorn



Acorn RISC Machine
CPU Software Manual

Acorn RISC Machine CPU Software Manual

Issue 1.00

Acorn Computers Limited

The product described in this manual and products for use with it, are subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith. However, it is acknowledged that there may be errors or omissions in this manual. Revised versions of this manual, when available, can be obtained on request from Acorn Computers.

This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any errors or omission in, this manual, or any incorrect use of the product.

Written and produced by: Acorn Computers Limited, Cambridge.

First published October 1985

Published by Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge, CB1 4JN.

Contents

1. PROGRAMMERS' MODEL

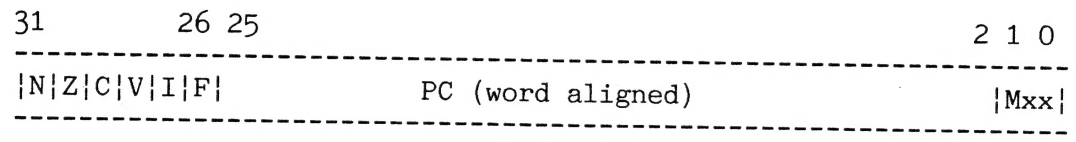
General

ARM has a 32 bit data bus and a 26 bit address bus. The data types the processor accesses are Bytes (8-bit B) and Words (32-bit W), where words must be aligned to a multiple of four bytes. Instructions are exactly one word, and data operations (e.g. ADD) are only performed on word quantities. Load and store operations can operate on either bytes and words.

Registers

The programmer sees a bank of sixteen 32-bit registers, R0 to R15; the only two special purpose registers are R14 and R15. R15 contains the Program Counter (PC), and the Processor Status Word (PSR), whilst R14 is the subroutine Link register (which receives a copy of R15 on a Branch and Link instruction). Special bits in the instructions allow the PC and PSR to be treated together or separately.

The format of the PC/PSR is as follows:



N (bit 31) : is Negative / signed less than flag
Z (bit 30) : is Zero flag
C (bit 29) : is Carry / not borrow / rotate extend flag
V (bit 28) : is oVerflow flag
I (bit 27) : is Interrupt ReQuest (IRQ) disable
F (bit 26) : is Fast Interrupt reQuest (FIQ) disable
PC (bits 25-2) is output as a word address (with two low-order zeros).
M1 and M0 (bit 0 and 1) : define current Mode and register bank

The processor has 25 registers. Depending on the Mode bits the programmer sees a subset of 16 of these:

Value of Mode Bits

	0	1	2	3
	Normal	FIQ	IRQ	SVC/Abort/Undefined
visible registers	R0	R0	R0	R0
	R1	R1	R1	R1
	¦	¦	¦	¦
	R9	R9	R9	R9
	R10	R10_FIQ	R10	R10
	R11	R11_FIQ	R11	R11
	R12	R12_FIQ	R12	R12
	R13	R13_FIQ	R13_IRQ	R13_SVC
	R14	R14_FIQ	R14_IRQ	R14_SVC
	R15	R15	R15	R15

User mode is the normal program execution state; registers R0-15 exist directly and in this mode only the N,Z,C and V bits of the PSR may be changed.

FIQ processing state (described in the Interrupts section) has five private registers mapped to R10-14 (R10_FIQ-R14_FIQ). Most FIQ programs (for data

transfer especially) will not need to save any other registers.

IRQ processing state has two private registers mapped to R13, R14 (R13_IRQ, R14_IRQ).

Supervisor mode (entered on SVC calls and other traps) has two private registers mapped to R13, R14 (R13_SVC, R14_SVC).

Two registers are enough for a private stack pointer and link register.

Because of the pipelined nature of the processor changes to the processor mode do not take effect immediately: there is a 1 clock cycle delay. This affects the use of Data Processing (see Section 2) instructions to change mode: after a TEQP instruction the programmer should not make an access to a multiply mapped register, but may access any of the other registers or use a no operation.

Interrupts

FIQ

The FIQ (Fast Interrupt reQuest) signal is designed to be used as a data transfer or channel process. Its effect may be masked out by setting the F flag in the PSR (but note that this is not possible from user mode). ARM checks for existence of FIQ at the end of instructions. When ARM is FIQed it will:

(a) save R15 in R14_FIQ;
(b) set M0, M1 to FIQ mode and set the F and I bits in the PC word;
(c) set PC to 28 (4*7).

Return from FIQ by SUBS PC,R14_FIQ,#4.

IRQ

The IRQ (Interrupt ReQuest) signal is a normal interrupt. It has a lower priority than FIQ, and is masked out when a FIQ sequence is entered. Its effect may be masked out at any time by setting the I bit in the PC (but note that this is not possible from user mode). ARM checks for existence of IRQ at the end of instructions. When successfully IRQed ARM will:

(a) save R15 in R14_IRQ;
(b) set M0, M1 to IRQ mode and set the I bit in the PC word;
(c) set PC to 24 (4*6).

Return from IRQ by SUBS PC,R14_IRQ,#4.

Address exception trap

When an address exception (a data transfer at an address above &3FFFFFF) is seen ARM will:

(a) complete the instruction if LDM/STM, or return to state just before execution (LDR/STR) - see data aborts
(b) save R15 in R14_SVC;
(c) set M0, M1 to SVC mode and set the I bit in the PC word;
(d) set PC to 20 (4*5).

Return from trap by subtracting 4 from R14_SVC and placing the result in R15

and PSR. This will return to the instruction after the one causing the trap.

LDM and STM will only cause address exceptions on the first data transfer of the instruction. A transaction which starts in the "legal" area and moves into the "illegal" area will not cause an address exception, and will attempt to access store addressed by the bottom 26 bits.

Abort

The Abort signal comes from a Memory Management system to complain about the current bus cycle. ARM checks for existence of Abort at the end of the first phase of each bus cycle. When successfully Aborted ARM will respond in one of two ways:

(i) if the abort occurred during an instruction prefetch, the prefetched instruction is marked as invalid; and when it comes to execution, it is reinterpreted as below.
(ii) if the abort occurred during a data access, the action depends on the instruction type. Data transfer instructions (e.g. LDR) are aborted as though the instruction had not executed. The LDM and STM instructions complete, and if writeback is set, the base is ALWAYS updated, even if the instruction would have overwritten it (i.e. LDM with base in list).

Then:

(a) save R15 in R14_SVC;
(b) set M0, M1 to SVC mode and set the I bit in the PC word;
(c) set PC to 12 (4*3) for prefetch abort, 16 (4*4) for data abort.

Continue after an instruction prefetch abort by subtracting 4 from R14_SVC and placing the result in R15 and PSR. A data access abort requires any auto-indexing to be reversed before returning to reexecute the offending instruction, the return being done by subtracting 8 from R14_SVC and placing the result in R15 and PSR.

Software interrupt

The software interrupt is used for getting into supervisor mode. ARM will:

(a) save R15 in R14_SVC;
(b) set M0, M1 to SVC mode and set the I bit in the PC word;
(c) set PC to 8 (4*2).

Return from SWI by transfering R14_SVC to R15 and PSR.

Undefined instruction trap

When an undefined instruction is seen ARM will:

(a) save R15 in R14_SVC;
(b) set M0, M1 to SVC mode and set the I bit in the PC word;
(c) set PC to 4 (4*1).

Return from trap by transfering R14_SVC to R15 and PSR.

Reset

When Reset goes low ARM will:

(a) stop the currently executing instruction and start executing no-ops
When Reset goes high again it will:
(b) save R15 in R14_SVC;
(c) set M0, M1 to SVC mode and set the F and I bits in the PC word;
(d) set PC to 0 (4*0).

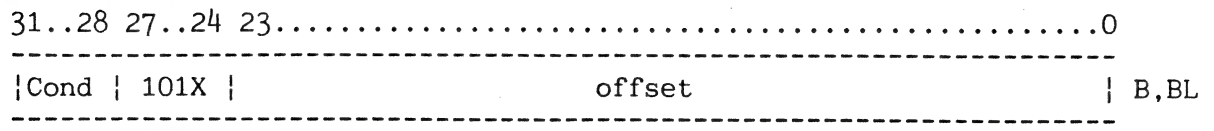
Vector Summary

0000000	Reset
0000004	Undefined instruction
0000008	Software interrupt
000000C	Abort (prefetch)
0000010	Abort (data)
0000014	Address exception
0000018	IRQ
000001C	FIQ

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine. The exception may be FIQ, where the routine might reside at 000001C onwards.

2. INSTRUCTION SET

Branch and Branch with Link



101X is 1010 for branch B
1011 for branch with link BL

Condition encoding:-

BEQ	BNE	BCS	BCC	BMI	BPL	BVS	BVC	BHI	BLS	BGE	BLT	BGT	BLE	BAL	BNV
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

All branches take a 24 bit word offset. The Branch with Link type writes the old PC and PSR into R14.

BAL -> always
BCC -> carry clear / unsigned lower than
BCS -> carry set / unsigned higher or same
BEQ -> equal (Z set)
BGE -> greater or equal (N set and V set or N clear and V clear)
BGT -> greater ((N set and V set or N clear and V clear) and Z clear)
BHI -> higher unsigned (C set and Z clear)
BLE -> less than or equal (N set and V clear or N clear and V set or Z set)
BLS -> lower or same unsigned (C clear or Z set)
BLT -> less than (N set and V clear or N clear and V set)
BMI -> negative (N set)
BNE -> not equal (Z clear)
BNV -> never
BPL -> positive (N clear)
BVC -> overflow clear
BVS -> overflow set

The branch offset must take account of the prefetch operation, which causes the PC to be 2 words ahead of the current instruction. For example:

```
    EAFFFFFE  here BAL here
```

In spite of the prefetching of instructions the value written into the link register is the address of the instruction following the branch and link instruction. Return by MOVS PC,R14 (MOV PC,R14) if the link register has not been saved or LDM Rn,{PC}^ (LDR PC,[Rn]) if the link register has been saved (instructions in () do not restore the status).

Assembler Syntax

B<L><cond> {expression}

<cond> is a two-char mnemonic as in the table above (EQ, NE, VS etc.). If absent then ALways will be used.

{expression} is the destination. The assembler calculates the offset.

Items in <> are optional.

Data Processing

```
31..28 27.25 24.....21   20  19..16 15..12 11..8 7...4 3...0
-------------------------------------------------------------
|Cond | 000 |      Opc      | Scc | Rn  | Rd  |   Shift   | Rm | R~R->R
-------------------------------------------------------------
|Cond | 001 |      Opc      | Scc | Rn  | Rd  | Shf |    Imm    | R~#->R
-------------------------------------------------------------
```

The instruction is only executed if the condition is true - see Branches.

Rd is the destination register.

Rm, Rn are source registers.

S2, below, is either Rm or the immediate field zero extended.

Shf is a 4 bit field defining a rotate right by Shf*2 of the 32 bit zero extended 8 bit Imm field.

Shift is an 8 bit field specifying the operation of the barrel shifter. The possible shifts are:

```
nnnnn000 logical left      by 0 to 31 bits              B31<..B0; B(32-N)->C
nnnnn010 logical right     by 1 to 32 bits <0 means 32> B31..>B0; B(N-1)->C
nnnnn100 arithmetic right  by 1 to 32 bits <0 means 32> B31..>B0; B(N-1)->C
nnnnn110 rotate right      by 1 to 31 bits           B0>B31..>B0; B(N-1)->C
00000110 rotate right one bit with extend                  C->B31, B0->C
 Rs x001 logical left      by Rs
 Rs x011 logical right     by Rs
 Rs x101 arithmetic right  by Rs
 Rs x111 rotate right      by Rs
```

Rs specified shifts require one additional execution cycle. Only the least significant byte of Rs is used, and signifies a shift of 0 to +255 bits. Logical left by 0 is a shift which does nothing to either the data or the carry and is used if the unmodified register only is required.

Scc is zero for no change in condition codes, 1 for a change

- when Rd is not R15, the condition codes are updated from the ALU flags.
- when Rd is R15, the PSR is overwritten by the corresponding bits in the ALU result, though some bits can only be changed in particular modes.

So normal moves to R15 are only 24 bits, moves with Scc set are 28 bits (full PC and user PSR: in non-user modes all 32 bits are used) while the result is used to set the PSR bits ONLY in CMP, CMN, TST, TEQ instructions when Rd is R15.

When Rm is R15 the value of the PC plus the PSR is presented to the ALU.
When Rn is R15 the PC is presented without the PSR, i.e. those bits are 0.

Mnemonic	Operation	Opc	Flags Affected
ADC	Rd:=Rn+SHIFT(S2)+C	5	N, Z, C, V
ADD	Rd:=Rn+SHIFT(S2)	4	N, Z, C, V
AND	Rd:=Rn AND SHIFT(S2)	0	N, Z, C
BIC	Rd:=Rn AND NOT(SHIFT(S2))	E	N, Z, C
CMN	SHIFT(S2)+Rn	B	N, Z, C, V
CMP	Rn-SHIFT(S2)	A	N, Z, C, V
EOR	Rd:=Rn EOR SHIFT(S2)	1	N, Z, C
MOV	Rd:=SHIFT(S2)	D	N, Z, C
MVN	Rd:=NOT SHIFT(S2)	F	N, Z, C
ORR	Rd:=Rn OR SHIFT(S2)	C	N, Z, C
RSB	Rd:=SHIFT(S2)-Rn	3	N, Z, C, V
RSC	Rd:=SHIFT(S2)-Rn-1+C	7	N, Z, C, V
SBC	Rd:=Rn-SHIFT(S2)-1+C	6	N, Z, C, V
SUB	Rd:=Rn-SHIFT(S2)	2	N, Z, C, V
TEQ	Rn EOR SHIFT(S2)	9	N, Z, C
TST	Rn AND SHIFT(S2)	8	N, Z, C

The condition codes N, Z, C, V are set by the ALU in arithmetic operations (ADD, ADC, SUB, SBC, RSB, RSC, CMP, RCP). The logical operations set N and Z from the ALU, C from the shifter, and V is unaffected by the instruction.

Assembler Syntax

opcode<cond><S><P> Rd<,Rn>,Rm<,shift>
or ,#expression

<cond> - two char mnemonic - see branch instructions.
<S> - set condition codes if S present (implied for CMP, CMN, TEQ, TST)
<P> - make Rd = R15 in instructions where Rd is not actually written to
Also sets Scc bit. Used for changing PSR.

e.g. TEQP R15,#0 ;Change to user mode (see description of Scc when Rd=15)

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

Rn not required in instructions with only two operands.

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

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

<shiftname>s are:

ASL, LSL, LSR, ASR, ROR.

Examples

```
ADDEQ R2,R4,R5
TEQS R4,#3
SUB R4,R5,R7,LSR R2
```

The First Devices ...

The first CPUs have two minor faults in the shift logic:

When using RRX shifts with S set, the carry out is the result of ORing the top and bottom bits of the source together, instead of just the bottom bit.

When using ROR with a register controlled shift and the register is greater than 31, the carry out will always be zero, instead of the bit MOD 32.

Single Data transfer

```
31..28 27..24   23   22    21   20   19..16 15..12 11.......4 3...0
-------------------------------------------------------------------
|Cond | 0100 | U/D | B/W | T  | L/S | Rn  | Rd  |     offset   |[Rn],off
-------------------------------------------------------------------
|Cond | 0101 | U/D | B/W | Wb | L/S | Rn  | Rd  |     offset   |[Rn,off]
-------------------------------------------------------------------
|Cond | 0110 | U/D | B/W | T  | L/S | Rn  | Rd  |  Shift  | Rm |[Rn],Rm
-------------------------------------------------------------------
|Cond | 0111 | U/D | B/W | Wb | L/S | Rn  | Rd  |  Shift  | Rm |[Rn,Rm]
-------------------------------------------------------------------
```

There are two forms of the instructions, either with a 12 bit binary offset in the instruction "off", or with a register (possibly shifted in some way). The offset may be added to (U/D=1) or subtracted from (U/D=0) the index register Rn.

[Rn],off and [Rn],Rm are Post-Indexed addressing modes: the address consists of Rn; the offset always then modifies and writes back the index register Rn.

[Rn,off] and [Rn,Rm] are Pre-Indexed addressing modes: the address consists of Rn modified by the offset; if Wb = 1 then the calculated address is written back to the index register Rn, otherwise the it is unchanged.

The Wb bit gives optional auto increment and decrement addressing modes.

L/S = 1 -> Rd becomes the operand i.e. LDR
L/S = 0 -> the operand becomes Rd i.e. STR

B/W: 1 transfer byte between register and any byte address
on load the byte will be zero extended to a word
B/W: 0 transfer word between register and any word aligned address.
If the address is not word aligned the result of the transfer
is not defined.

The 8 shift control bits are described in the data processing instructions.

T = 1 forces the translate output (the TRANS pin) to be active during the data transfer cycle, thereby allowing programs running in supervisor mode to load and save user memory areas.

These instructions will never affect the PSR, even when Rd or Rn is R15.

When using the PC as the base register one must remember that it contains an address 8 byte addresses further on than the start of the current instruction. For example

```
WORDDATA EQUATE &12345678
         ..........
         LDR    R12,WORDDATA-.-8(PC)
```

would load the contents of WORDDATA into R12 ("." is the current program counter run by the assembler).

Assembler Syntax

opcode<cond><B><T> Rd,Address

<cond> - two-character mnemonic - see branch instructions.
<B> - if B present then byte transfer, otherwise word transfer.
<T> - if T present the translate bit will be set.

Rd is an expression evaluating to a valid register number.

Address can be:

1. 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. If out of range, an error will be generated.

2. Pre-indexed

[Rn] offset of zero

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

[Rn,<+/->Rm<,shift>]<!> => offset of +/- contents of <index> register, shifted by <shift>. For possible shifts, see instruction format 1.

Rn and Rm are expressions evaluating to a valid register number.

NOTE if Rn is R15 then assembler will subtract 8 from the offset value to deal with ARM pipelining.

<!> => write back the base register if present.

3. Post-indexed

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

[Rn],<+/->Rm<,shift> => offset of +/- contents of <index> register shifted as in 2. above.

Examples

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

STR R1,[BASE,INDEX]! ;store R1 at BASE+INDEX (both register
                     ;contents) and write back address to BASE
```

```
STR R1,[BASE],INDEX   ;store R1 at BASE and writeback
                      ;BASE+INDEX to BASE

LDR R1,[BASE,#17]     ;load R1 from contents of BASE+17.
                      ;Don't write back

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

Block data transfer

```
31..28 27..24   23    22    21    20  19..16 15.....................0
-------------------------------------------------------------------
|Cond | 100X | U/D | PSR |  Wb | L/S |  Rn  |       operand         | LDM,STM
-------------------------------------------------------------------
```

LDM, STM move with push/pull

```
100X U/D is 1000 1 for up post-increment
            1001 1 for up pre-increment
            1000 0 for down post-decrement
            1001 0 for down pre-decrement
```

This instruction can load any registers or save any registers using any of the registers as the base, which may then be modified. Loading or saving is specified by the L/S bit, and the direction of movement from the base by the U/D bit. Note that the registers are always saved from lowest to highest, i.e. R0 will always be first (if in the list) and R15 (the PC and PSR) last. Also R0 will always be saved to/loaded from a lower memory address than R1, etc. The bits represent the registers, i.e. bit0 means R0 will be transferred. The PC is represented by bit 15, and the condition codes are saved with it on STM. When bit 15 = 1 in a LDM instruction, then the bits of the PSR which may be modified in the current mode are loaded (in addition to PC) if PSR = 1.

Pre-increment or decrement means that the address is incremented or decremented before each memory transfer, whereas post- modifies the address after each transfer. This refers to the notional stacking action of the instruction. The actual order of register transfer is not necessarily that which would be expected from normal stacking, but the end effect is the same. In fact in 'down' type instructions the address of the lowest register is calculated, then the registers transferred going up memory, with the base ending up pointing to the bottom again (if Wb is specified).

The Wb bit indicates whether the base register is to be updated, as before.

When writeback is specified, the base is written back during the second cycle of the instruction. During a STM, the first register is written out during the first cycle. A STM which includes storing the base, with the base as the first register in the list, will therefore store the unchanged value, whereas with the base anywhere else, will store the new value. A LDM will always overwrite the updated base, if the base is in the list.

When the base is the PC, the PSR bits will be used to form the address as well, so unless all interrupts are enabled and all flags are zero an address exception will occur. Also write back is never allowed when the base is the PC.

For STM the PSR bit is ignored in user mode. In other modes it may be used to force transfers from the user register bank. Note that when it is so used, write back will also be to the user bank, though the base will be fetched from the current bank. Therefore don't use write back when forcing user bank.

The action of the instructions when <operand> is zero is undefined and may affect registers or store.

Assembler Syntax

opcode<cond>FD|ED|FA|EA|IA|IB|DA|DB Rn<!>,Rlist<^>

<cond> - two character condition mnemonic, see next section.

FD|ED etc define pre/post indexing and up/down bit. It is assumed that these instructions will normally be used for stack operations. The F and E refer to a "full" or "empty" stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will go up and LDM down, if descending, vice-versa.

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

Rn is an expression evaluating to a valid register number.

Rlist can be either a list of registers enclosed in {} or an expression evaluating to the 16 bit operand.

<!> is writeback if present.

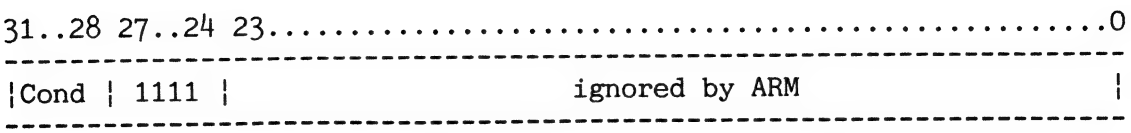
<^> => set PSR if present (note different meanings for PSR bit in LDM and STM).

Examples

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

STMIA BASE,{R0-R15}  ;save all regs
```

Supervisor Call



SWI -> SoftWare Interrupt

The PC word and PSR are saved in R14_SVC, with the PC adjusted to point to the word after the SWI instruction. The PC 24 bits are set to 8 (4*2) and M0, M1 to SVC mode and the processor continues (see the section on traps and vectors).

In Assembler, SWI<cond> Expression

Expression is ignored by ARM.

Refer to the "Brazil Kernel" manual for the IO control given by various SWI calls.

Examples

```
SWI   ReadC

SWI   WriteI+"k"
```

3. INSTRUCTION SUMMARY

Instruction Summary

31..28 27.25 24.....21 20 19..16 15..12 11..8 7..4 3...0

Cond	000	Opc		Scc	Rn	Rd	Shift		Rm	R~R->R
Cond	001	Opc		Scc	Rn	Rd	Shf	Imm		R~#->R
Cond	0100	U/D	B/W	T	L/S	Rn	Rd	offset		(LS),of
Cond	0101	U/D	B/W	Wb	L/S	Rn	Rd	offset		(LS,of)
Cond	0110	U/D	B/W	T	L/S	Rn	Rd	Shift	Rm	(LS),Rm
Cond	0111	U/D	B/W	Wb	L/S	Rn	Rd	Shift	Rm	(LS,Rm)
Cond	100X	U/D	PSW	Wb	L/S	Rn	operand			LDM,STM
Cond	101X	offset								B,BL
Cond	1111	ignored by ARM								SWI

Instructions of the form | Cond | 110X | and | Cond | 1110 | will cause Undefined Instruction traps. These codes are reserved for future internal or coprocessor expansion.

4. INSTRUCTION SPEEDS

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:

R~#->Rd, R~R->Rd	1 S	+1 S for SHIFT(Rs)	+1 S + 1 N if R15 written
LDR	2 S + 1 N	+1 S for SHIFT(Rs)	+1 S + 1 N if R15 written
STR	2 N	+1 S for SHIFT(Rs)	
LDM	(n+1)S + 1 N		+1 S + 1 N if R15 written
STM	(n-1)S + 2 N		
B,BL	2 S + 1 N		
SWI	2 S + 1 N		

n is the number of registers transferred in a LDM or STM.

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

S is a sequential cycle or a cycle which does not require memory at all.
N is a non-sequential cycle.

With the initial second processor product S cycles take 3/20uSec.
N cycles take 6/20uSec.
This corresponds to the 20MHz crystal divided by 3 or 6.

5. INSTRUCTION SET EXAMPLES

Examples of the instruction set

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 all save some), mostly they just save code.

Using the conditional instructions

(1) using conditionals for logical OR

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

can be replaced by

```
      CMP    Rn,#p
      CMPNE  Rm,#q   ;if condition not 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) unsigned 32 bit multiply

```
;enter with numbers in Ra, Rb
      MOV    Rm,#0            ;result register
Loop  MOVS   Ra,Ra,LSR #1
      ADDCS  Rm,Rm,Rb
      ADD    Rb,Rb,Rb
      BNE    Loop             ;stops when Ra becomes zero
                              ;Rm contains Ra*Rb
                              ;(Ra set to zero, Rb junk)
```

(4) 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
```

(5) 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:="."
```

(6) division and remainder

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

Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive or feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32 bit generator needs more than one feedback tap to be maximal length (i.e. 2^32-1 cycles before repetition). 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
```

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
```

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
```

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
```

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 mode to that
;when the R14 was set up. This generally applies
;to non-user mode programming e.g.
;MOVS PC,R14. MOV PC,R14 is safer!
```