ARM Instruction Formats and Timings

Last revised: 15th November 1995

This is not an official document from ARM Ltd; in fact other than a couple of nice people from ARM limited pointing out some of the corrections, they have no connection with this document at all. They do not guarantee to have found all the mistakes in this, so don't blame them when you find some more.

Corrections/amendments for this document would be most welcome. They should be reported to Robin Watts at the address below.

Throughout this document, a 'word' refers to 32 bits (thats 4 bytes) of memory. If you don't like this, tough.

This document is available in several forms. An index to them can be found it http://www.comlab.ox.ac.uk/oucl/users/robin.watts/ARMinstrs/ on the World Wide Web, or via anonymous FTP to ftp.comlab.ox.ac.uk in /tmp/Robin.Watts/ARMinstrs/README.

1. Processor Modes

ARM processors have a user mode and a number of privileged supervisor modes. These are used as follows:

- IRQ — Entered when an Interrupt Request (IRQ) is triggered.
- FIQ — Entered when a Fast Interrupt Request (FIQ) is triggered.
- SVC — Entered when a Software Interrupt (SWI) is executed.
- Undef — Entered when an Undefined instruction is executed (Not ARM 2 and 3, where SVC mode is entered).
- Abt — Entered when a memory access attempt is aborted by the memory manager (e.g. MEMC or MMU), usually because an attempt is made to access non-existent memory or to access memory from an insufficiently privileged mode (Not ARM 2 and 3, where SVC mode is entered).

In each case the appropriate hardware vector is also called.

2. Registers

The ARM 2 and 3 have 27 32 bit processor registers, 16 of which are visible at any given time (which sixteen varies according to the processor mode). These are referred to as R0-R15.

The ARM 6 and later have 31 32 bit processor registers, again 16 of which are visible at any given time.

R15 has special significance. On the ARM 2 and 3, 24 bits are used as the program counter, and the remaining 8 bits are used to hold processor mode, status flags and interrupt modes. R15 is therefore often referred to as PC.

R15 = PC = NZCVIFpp pppppppp pppppppp ppppMM

Bits 0-1 and 26-31 are known as the PSR (processor status register). Bits 2-25 give the address (in words) of the instruction currently being fetched into the execution pipeline (see below). Thus instructions are only ever executed from word aligned addresses.

M	Current processor mode
0	User Mode
1	Fast interrupt processing mode (FIQ mode)
2	Interrupt processing mode (IRQ mode)
3	Supervisor mode (SVC mode)

Name	Meaning
N	Negative flag
Z	Zero flag
C	Carry flag
V	oVerflow flag
I	Interrupt request disable
F	Fast interrupt request disable

R14, R14_FIQ, R14_IRQ, and R14_SVC are sometimes known as ‘link’ registers due to their behaviour during the branch with link instructions.

The ARM 6 and later processor cores support a 32 bit address space. Such processors can operate in both 26 bit and 32 bit PC modes. In 26 bit PC mode, R15 acts as on previous processors, and hence code can only be run in the lowest 64MBytes of the address space. In 32 bit PC mode, all 32 bits of R15 are used as the program counter. Separate status registers are used to store the processor mode and status flags. These are defined as follows:

NZCVxxxx xxxxxxxx xxxxxxxx IFxMMMMM

Note that the bottom two bits of R15 are always zero in 32-bit modes — i.e. you can still only get word-aligned instructions. Any attempts to write non-zeros to these bits will be ignored.

The following modes are currently defined:

M	Name	Meaning
00000	usr_26	26 bit PC User Mode
00001	fiq_26	26 bit PC FIQ Mode
00010	irq_26	26 bit PC IRQ Mode
00011	svc_26	26 bit PC SVC Mode
10000	usr_32	32 bit PC User Mode
10001	fiq_32	32 bit PC FIQ Mode
10010	irq_32	32 bit PC IRQ Mode
10011	svc_32	32 bit PC SVC Mode
10111	abt_32	32 bit PC Abt Mode
11011	und_32	32 bit PC Und Mode

Extrapolating from the above table, it might be expected that the following two modes are also defined:

M	Name	Meaning
00111	abt_26	26 bit PC Abt Mode
01011	und_26	26 bit PC Und Mode

These are in fact undefined (and if you **do** write 00111 or 01011 to the mode bits, the resulting chip state won’t be what you might expect — i.e. it won’t be a 26-bit privileged mode with the appropriate R13 and R14 swapped in).

The following table shows which registers are available in which processor modes:

<table>
<tr><td>Mode</td><td colspan="4">Registers available</td></tr>
<tr><td>USR</td><td colspan="3">R0 — R14</td><td>R15</td></tr>
<tr><td>FIQ</td><td>R0 — R7</td><td colspan="2">R8_FIQ — R14_FIQ</td><td>R15</td></tr>
<tr><td>IRQ</td><td colspan="2">R0 — R12</td><td>R13_IRQ — R14_IRQ</td><td>R15</td></tr>
<tr><td>SVC</td><td colspan="2">R0 — R12</td><td>R13_SVC — R14_SVC</td><td>R15</td></tr>
<tr><td>ABT</td><td colspan="2">R0 — R12</td><td>R13_ABT — R14_ABT</td><td>R15 (ARM 6 and later only)</td></tr>
<tr><td>UND</td><td colspan="2">R0 — R12</td><td>R13_UND — R14_UND</td><td>R15 (ARM 6 and later only)</td></tr>
</table>

There are six status registers on the ARM6 and later processors. One is the current processor status register (CPSR) and holds information about the current state of the processor. The other five are the saved processor status registers (SPSRs): there is one of these for each privileged mode, to hold information about the state the processor must be returned to when exception handling in that mode is complete.

These registers are set and read using the MSR and MRS instructions respectively.

3. Pipeline

Rather than being a microcoded processor, the ARM is (in keeping with its RISCness) entirely hardwired.

To speed execution the ARM 2 and 3 have 3 stage pipelines. The first stage holds the instruction being fetched from memory. The second starts the decoding, and the third is where it is actually executed. Due to this, the program counter is always 2 instructions beyond the currently executing instruction. (This must be taken account of when calculating offsets for branch instructions).

Because of this pipeline, 2 instruction cycles are lost on a branch (as the pipeline must refill). It is therefore often preferable to make use of conditional instructions to avoid wasting cycles. For example:

```
     ...
     CMP R0,#0
     BEQ over
     MOV R1,#1
     MOV R2,#2
over
     ...
```

can be more efficiently written as:

```
     ...
     CMP R0,#0
     MOVNE R1,#1
     MOVNE R2,#2
     ...
```

4. Timings

ARM instructions are timed in a mixture of S, N, I and C cycles.

An S-cycle is a cycle in which the ARM accesses a sequential memory location.

An N-cycle is a cycle in which the ARM accesses a non-sequential memory location.

An I-cycle is a cycle in which the ARM doesn’t try to access a memory location or to transfer a word to or from a coprocessor.

A C-cycle is a cycle in which a word is transferred between the ARM and a coprocessor on either the data bus (for uncached ARMs) or the coprocessor bus (for cached ARMs).

The different types of cycle must all be at least as long as the ARM’s clock rating. The memory system can stretch them: with a typical DRAM system, this results in:

- N-cycles being twice the minimum length (essentially because DRAMs require a longer access protocol when the memory access is non-sequential).
- S-cycles usually being the minimum length, but occasionally being stretched to N-cycle length (when you’ve just moved sequentially from the last word of one memory "row" to the first of the next one[1]).

---

[1] Memory controllers tend to use this simple strategy: if an N-cycle is requested, treat the access as not being in the same row; if an S-cycle is requested, treat the access as being in the same row unless it is effectively the last word in the row (which can be detected quickly). The net result is that **some** S-cycles will last the same time as an N-cycle; if I remember correctly, on an Archimedes these are S-cycle accesses to an address which is divisible by 16. The practical consequences of this for Archimedes code are: (a) that about 1 in 4 S-cycles becomes an N-cycle, since for this purpose, all addresses are word addresses and so divisible by 4; (b) that it is

- I- and C-cycles always being the minimum length.

With a typical SRAM system, all four types of cycle are typically the minimum length.

On the 8MHz ARM2 used in the Acorn Archimedes A440/1, an S (sequential) cycle is 125ns and an N (non-sequential) cycle is 250ns. It should be noted that these timings are **not** attributes of the ARM, but of the memory system. E.g. an 8MHz ARM2 can be connected to a static RAM system which gives a 125ns N cycle. The fact that the processor is rated at 8MHz simply means that it isn't guaranteed to work if you make any of the types of cycle shorter than 125ns in length.

Cached processors: All the information given is in terms of the clock cycles seen by the ARM. These do not occur at a constant rate: the cache control logic changes the source of the clock cycles presented to the ARM when cache misses occur.

Generally, a cached ARM has two clock inputs: the "fast clock" FCLK and the "memory clock" MCLK. When operating normally from cache, the ARM is clocked at FCLK speed and all types of cycle are the minimum length: cache is effectively a type of SRAM from this point of view. When a cache miss occurs, the ARM's clock is synchronised to MCLK, then the cache line fill takes place at MCLK speed (taking either N+3S or N+7S depending on the length of cache lines in the processor involved), then the ARM's clock is resynchronised back to FCLK.

While the memory access is taking place, the ARM is being clocked: however, an input called NWAIT is used to cause the ARM cycles involved not to do anything until the correct word arrives from memory, and usually not to do anything while the remaining words arrive (to avoid getting further memory requests while the cache is still busy with the cache line refill). The situation is also complicated by the fact that the cached ARM can be configured either for FCLK and MCLK to be synchronous to each other (so FCLK is an exact multiple of MCLK, and every MCLK clock cycle starts at just about the same time as an FCLK cycle) or asynchronous (in which case FCLK and MCLK cycles can have any relationship to each other).

All in all, the situation is therefore quite complicated. An approximation to the behaviour is that when a cache line miss occurs, the cycle involved takes the cache line refill time (i.e. N+3S or N+7S) in MCLK cycles, with N-cycles and S-cycles probably being stretched as described above for DRAM, plus a few more cycles to allow for the resynchronisation periods. For any more details, you really need to get a datasheet for the processor involved.

5. Instructions

Each ARM instruction is 32 bits wide, and are explained in more detail below. For each instruction class we give the instruction bitmap, and an example of the syntax used by a typical assembler.

It should of course be noted that the mnemonic syntax is not fixed; it is a property of the assembler, not the ARM machine code.

5.1. Condition Code

The top nibble of every instruction is a condition code, so every single ARM instruction can be run conditionally.

occasionally worth taking care to align code carefully to avoid this effect and get some extra performance.)

Instruction Bitmap	Cond No	Cond Code	Executes if
`0000xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	0	EQ(Equal)	Z
`0001xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	1	NE(Not Equal)	˜Z
`0010xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	2	CS(Carry Set)	C
`0011xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	3	CC(Carry Clear)	˜C
`0100xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	4	MI(MInus)	N
`0101xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	5	PL(PLus)	˜N
`0110xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	6	VS(oVerflow Set)	V
`0111xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	7	VC(oVerflow Clear)	˜V
`1000xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	8	HI(HIgher)	C and ˜Z
`1001xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	9	LS(Lower or Same)	˜C or Z
`1010xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	A	GE(Greater or equal)	N = V
`1011xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	B	LT(Less Than)	N = ˜V
`1100xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	C	GT(Greater Than)	(N = V) and ˜Z
`1101xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	D	LE(Less or equal)	(N = ˜V) or Z
`1110xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	E	AL(Always)	True
`1111xxxx xxxxxxxx xxxxxxxx xxxxxxxx`	F	NV(Never)	False

In most assemblers, the condition code is inserted immediately after the mnemonic stub; omitting a condition code defaults to AL being used.

HS (Higher or Same) and LO (LOwer) can be used as synonyms for CS and CC (respectively) in some assemblers.

The conditions GT, GE, LT, LE refer to signed comparisons whereas HS, HI, LS, LO refer to unsigned.

EORing a condition code with 1 gives the opposite condition code.

NB: ARM have deprecated the use of the NV condition code — you are now supposed to use MOV R0,R0 as a noop rather than MOVNV R0,R0 as was previously recommended. Future processors may have the NV condition code reused to do other things.

Instructions with false conditions execute in 1S cycle, and no time penalty is incurred by making an instruction conditional.

5.2. Data Processing Instructions

```
xxxx000a aaaSnnnn ddddcccc ctttmmmm  Register form
xxxx001a aaaSnnnn ddddrrrr bbbbbbbb  Immediate form
```

Typical Assembler Syntax:

```
MOV     Rd, #0
ADDEQS  Rd, Rn, Rm, ASL Rc
ANDEQ   Rd, Rn, Rm
TEQP    Pn, #&80000000
CMP     Rn, Rm
```

Combine contents of Rn with Op2, under operation a, placing the results in Rd.

If the register form is used, then Op2 is set to be the contents of Rm shifted according to t as below. If the immediate form is used, then Op2 = #b, ROR #2r.

t	Assembler		Interpretation
000	LSL #c		Logical Shift Left
001	LSL Rc		Logical Shift Left
010	LSR #c	for c != 0	Logical Shift Right
	LSR #32	for c = 0	
011	LSR Rc		Logical Shift Right
100	ASR #c	for c != 0	Arithmetic Shift Right
	ASR #32	for c = 0	
101	ASR Rc		Arithmetic Shift Right
110	ROR #c	for c != 0	Rotate Right.
	RRX	for c = 0	Rotate Right one bit with extend.
111	ROR Rc		Rotate Right

In the register form, Rc is signified by bits 8-11; bit 7 must be clear if Rc is used. (If you code a 1 instead, you'll get a multiply, a SWP or something unallocated instead of a data processing instruction.)

Also, only the bottom byte of Rc is used — If Rc = 256, then the shifts will be by zero.

"MOV[S] Ra,Rb,RLX" can be done by ADC[S] Ra,Rb,Rb, with RLX meaning Rotate Left one bit with extend.

Most assemblers allow ASL to be used as a synonym for LSL. Since opinions differ on what an arithmetic left shift is, LSL is the preferred term.

By setting the S bit in a MOV, MVN or logical instruction, (in either the register or immediate form) the carry flag is set to be the last bit shifted out.

If no shift is done, the carry flag will be unaffected.

If there is a choice of forms for an immediate (e.g. #1 could be represented as 1 ROR #0, 4 ROR #2, 16 ROR #4 or 64 ROR #6), the assembler is expected to use the one involving a zero rotation, if available. So MOVS Rn,#const will leave the carry flag unaffected if 0 <= const <= 255, but will change it otherwise.

aaaa	Assembler	Meaning	P-Code
0000	AND	Boolean And	Rd = Rn AND Op2
0001	EOR	Boolean Eor	Rd = Rn EOR Op2
0010	SUB	Subtract	Rd = Rn - Op2
0011	RSB	Reverse Subtract	Rd = Op2 - Rn
0100	ADD	Addition	Rd = Rn + Op2
0101	ADC	Add with Carry	Rd = Rn + Op2 + C
0110	SBC	Subtract with carry	Rd = Rn - Op2 - (1-C)
0111	RSC	Reverse sub w/carry	Rd = Op2 - Rn - (1-C)
1000	TST	Test bit	Rn AND Op2
1001	TEQ	Test equality	Rn EOR Op2
1010	CMP	Compare	Rn - Op2
1011	CMN	Compare Negative	Rn + Op2
1100	ORR	Boolean Or	Rd = Rn OR Op2
1101	MOV	Move value	Rd = Op2
1110	BIC	Bit clear	Rd = Rn AND NOT Op2
1111	MVN	Move Not	Rd = NOT Op2

Note that MVN and CMN are not as related as they first appear; MVN uses straight bitwise negation, setting Rn to the 1's complement of Op2. CMN compares Rn with the 2's complement of Op2.

These instructions fall broadly into 4 subsets:

MOV, MVN

Rn is ignored, and should be 0000. If the S bit is set, N and Z are set on the result, and if the shifter is used, C is set to be the last bit shifted out. V is unaffected.

CMN, CMP, TEQ, TST

Rd is not set by the instruction, and should be 0000. The S bit must be set (most assemblers do this automatically; if it weren't set, the instruction would be MRS, MSR, or an unallocated one.)

The arithmetic operations (CMN, CMP) set N, Z on result, and C and V from the ALU.

The logical operations (TEQ, TST) set N and Z on the result, C from the shifter if it is used (in which case it becomes the last bit shifted out), and V is unaffected.

As a special case (for ARMs >= 6, this only applies to 26 bit code), the dddd field being 1111 causes flags (in user mode), or the entire 26 bit PSR (in privileged modes) to be set from the corresponding bits of the result. This is indicated by a P suffix to the instruction — CMNP, CMPP, TEQP, TSTP. This is most commonly used to change mode via TEQP PC,#(new mode number). In 32 bit modes, MSR should be used instead (as TEQP etc will not work).

ADC, ADD, RSB, RSC, SBC, SUB

If the S bit is set, then N and Z are set on result, and C and V are set from the ALU.

AND, BIC, EOR, ORR

If the S bit is set, then N and Z are set on result, C is set from the shifter if used (in which case it becomes the last bit shifted out) and V is unaffected.

ADD and SUB can be used to make registers point to data in a position independent way, eg. ADD R0,PC,#24. This is so useful that some assemblers have a special directive called ADR which generates the appropriate ADD or SUB automatically. (ADR R0, fred typically puts the address of fred into R0, assuming fred is within range).

In 26-bit modes, special cases occur when R15 is one of the registers being used:

- If Rn = R15 then the value used is R15 with all the PSR bits masked out.
- If Op2 involves R15, then all 32 bits are used.

In 32-bit modes, all the bits of R15 are used.

In 26-bit modes, if Rd = R15 then:

- If the S bit is not set, only the 24 bits of the PC are set.
- If the S bit is set, both the PC and PSR are overwritten (though the Mode, I and F bits will not be altered unless we are in a non-user mode.)

For 32-bit modes, if Rd=15, all the bits of the PC will be overwritten, except the two least significant bits, which are always zero. If the S bit is not set, that is all that happens; if the S bit is set, the SPSR for the current mode is copied to the CPSR. You should not execute a data processing instruction with the PC as destination and the S bit set in 32-bit user mode, since user mode does not have an SPSR. (By the way, you won't break the processor by doing so — it's just that the results of doing so aren't defined, and may differ between processors.)

These instructions take the following number of cycles to execute: 1S + (1S if register controlled shift used) + (1S + 1N if PC changed)

5.3. Branch Instructions

```
xxxx101L oooooooo oooooooo oooooooo
```

Typical Assembler Syntax:

```
BEQ  address
BLNE subroutine
```

These instructions are used to force a jump to a new address, given as an offset in words from the value of the PC as this instruction is executed.

Due to the pipeline, the PC is always 2 instructions (8 bytes) ahead of the address at which this instruction was stored, so a branch with offset = (sign extended version of bits 0-23):

destination address = current address + 8 + (4 * offset)

In 26-bit modes, the top 6 bits of the destination address are cleared.

If the L flag is set, then the current contents of PC are copied into R14 before the branch is taken. Thus R14 holds the address of the instruction after the branch, and the called routine can return with MOV PC,R14.

In 26-bit modes, using MOVS PC,R14, to return from a branch with link, the PSR flags can be restored automatically on return. The behaviour of MOVS PC,R14 is different in 32-bit modes, and only suitable for return from an exception.

Both branch and branch with links, take 2S+1N cycles to execute.

5.4. Multiplication

```
xxxx0000 00ASdddd nnnnssss 1001mmmm
```

Typical Assembler Syntax:

```
MULEQS Rd, Rm, Rs
MLA    Rd, Rm, Rs, Rn
```

These instructions multiply the values of 2 registers, and optionally add a third, placing the result in another register.

If the S bit is set, the N and Z flags are set on the result, C is undefined, and V is unaffected.

If the A bit is set, then the effect of the operation is Rd = Rm.Rs + Rn otherwise, Rd = Rm.Rs.

The destination register shall not be the same as the operand register Rm. R15 shall not be used as an operand or as the destination register.

These instructions take 1S + 16I cycles to execute in the worst case, and may be less depending on arguement values. The exact time depends on the value of Rs, according to the following table:

Range of Rs	Number of cycles
&0 — &1	1S + 1I
&2 — &7	1S + 2I
&8 — &1F	1S + 3I
&20 — &7F	1S + 4I
&80 — &1FF	1S + 5I
&200 — &7FF	1S + 6I
&800 — &1FFF	1S + 7I
&2000 — &7FFF	1S + 8I
&8000 — &1FFFF	1S + 9I
&20000 — &7FFFF	1S + 10I
&80000 — &1FFFFF	1S + 11I
&200000 — &7FFFFF	1S + 12I
&800000 — &1FFFFFF	1S + 13I
&2000000 — &7FFFFFF	1S + 14I
&8000000 — &1FFFFFFF	1S + 15I
&20000000 — &FFFFFFFF	1S + 16I

These multiplication timings don't apply to ARM7DM. ARM7DM timings are given by the following table.

Range of Rs	MUL	MLA/ SMULL	SMLAL	UMULL	UMLAL
&0 — &FF	1S+1I	1S+2I	1S+3I	1S+2I	1S+3I
&100 — &FFFF	1S+2I	1S+3I	1S+4I	1S+3I	1S+4I
&10000 — &FFFFFF	1S+3I	1S+4I	1S+5I	1S+4I	1S+5I
&1000000 — &FEFFFFFF	1S+4I	1S+5I	1S+6I	1S+5I	1S+6I
&FF000000 — &FFFEFFFF	1S+3I	1S+4I	1S+5I	1S+5I	1S+6I
&FFFF0000 — &FFFFFEFF	1S+2I	1S+3I	1S+4I	1S+5I	1S+6I
&FFFFFF00 — &FFFFFFFF	1S+1I	1S+2I	1S+3I	1S+5I	1S+6I

5.5. Long Multiplication (ARM7DM)

```
xxxx0000 1UAShhhh llllssss 1001mmmm
```

Typical Assembler Syntax:

```
UMULL  Rl,Rh,Rm,Rs
UMLAL  Rl,Rh,Rm,Rs
SMULL  Rl,Rh,Rm,Rs
SMLAL  Rl,Rh,Rm,Rs
```

These instructions multiply the values of registers Rm and Rs to obtain a 64-bit product.

When the U bit is clear the multiply is unsigned (UMULL or UMLAL), otherwise signed (SMULL, SMLAL). When the A bit is clear the result is stored with its least significant half in Rl and its most significant half in Rh. When A is set, the result is instead added to the contents of Rh,Rl.

The program counter, R15 should not be used. Rh, Rl and Rm should be different.

If the S bit is set, the N and Z flags are set on the 64-bit result, C and V are undefined.

Timings for these can be found above in the multiplication section.

5.6. Single Data Transfer

```
xxxx010P UBWLnnnn ddddoooo oooooooo  Immediate form
xxxx011P UBWLnnnn ddddcccc ctt0mmmm  Register form
```

Typical Assembler Syntax:

```
LDR  Rd, [Rn, Rm, ASL#1]!
STR  Rd, [Rn],#2
LDRT Rd, [Rn]
LDRB Rd, [Rn]
```

These instructions load/store a word of memory from/to a register. The first register used in specifying the address is termed the base register.

If the L bit is set, then a load is performed. If not, a store.

If the P bit is set, then Pre-indexed addressing is used, otherwise post-indexed addressing is used.

If the U bit is set, then the offset given is added to the base register — otherwise it is subtracted.

If the B bit is set, then a byte of memory is transferred, otherwise a word is transferred. This is signified to assemblers by postfixing the mnemonic stub with a ‘B’.

The interpretation of the W bit depends on the addressing mode used:

- For pre-indexed addressing, W being set forces the writing back of the final address used for the address translation into the base register. (i.e. A side effect of the transfer is Rn := Rn +/- offset. This is signified to assemblers by postfixing the instruction with !.)
- For post-indexed addressing, the address is always written back, and the bit being set indicates that an address translation should be forced before the transfer takes place. This is signified to assmeblers by postfixing the mnemonic stub with ‘T’.

An address translation causes the chip to tell the memory system that this is a user mode transfer, regardless of whether the chip is in a user mode or a privileged mode at the time. This is useful e.g. when writing emulators: suppose for instance that a user mode program executes an STF instruction to an area of memory that may not be written by user mode code. If this is executed by an FPA, it will abort. If it is executed by the FPE, it should also abort. But the FPE runs in a privileged mode, so if it were to use normal stores, they wouldn’t abort. To make aborts work properly, it instead uses normal stores if it was called from a privileged mode, but STRTs if it was called from a user mode.

If the immediate form of the instruction is used, the o field gives a 12-bit offset. If the register form is used, then it is decoded as for the data processing instructions, with the restriction that shifts by register amounts are not allowed.

If R15 is used as Rd, the PSR is not modified. The PC should not be used in Op2.

Other restrictions:

- Don’t use writeback or post-indexing when the base register is the PC.
- Don’t use the PC as Rd for an LDRB or STRB.
- When using post-indexing with a register offset, don’t make Rn and Rm the same register (doing so makes recovery from aborts impossible).

A load takes 1S + 1N + 1I + (1S + 1N if PC changed) cycles, and a store takes 2N cycles.

5.7. Block Data Transfer

```
xxxx100P USWLnnnn llllllll llllllll
```

Typical Assembler Syntax:

```
LDMFD   Rn!, {R0-R4, R8, R12}
STMEQIA Rn,   {R0-R3}
STMIB   Rn,   {R0-R3}^
```

These instructions are used to load/store large numbers of registers from/to memory at a time. The memory addresses used are either increasing or decreasing in memory from a value held in a base register, Rn, (which may itself be stored), and the final address can be written back into the base. These instructions are ideal for implementing stacks, and storing/restoring the contents of registers on entry/exit from a subroutine.

The U bit indicates whether the address will be modified by +4 (set), or -4 (clear) for each register.

The W bit always indicates writeback.

If set, the L bit indicates a load operation should be performed. If clear, a save.

The P bit is used indicate whether to increment/decrement the base before or after each load/store (see the table below).

Bit l is set if Rl is to be loaded/stored by this operation.

Assemblers typically follow the mnemonic stub with a condition code, and then a two letter code to indicate the settings of the U and W bits.

Stub	Meaning	P	U
DA	Decrement Rn After each store/load	0	0
DB	Decrement Rn Before each store/load	1	0
IA	Increment Rn After each store/load	0	1
IB	Increment Rn Before each store/load	1	1

Synonyms for these exist which are clearer when implementing stacks:

Stub	Meaning
EA	Empty Ascending stack
ED	Empty Decending stack
FA	Full Ascending stack
FD	Full Decending stack

In an empty stack, the stack pointer points to the next empty position. In a full one the stack pointer points to the topmost full position. Ascending stacks grow towards high locations, and descending stacks grow towards low locations.

The registers are always stored so that the lowest numbered register is at the lowest address in memory. This can affect stacking and unstacking code. For instance, if I want to push R1-R4 on to a stack, then load them back two at a time, to get them back to the same registers, I need to do something like:

```
STMFD R13!,{R1,R2,R3,R4}  ;Puts R1 low in memory, i.e. at end of stack
LDMFD R13!,{R1,R2}
LDMFD R13!,{R3,R4}
```

for a descending stack, but something like:

```
STMFA R13!,{R1,R2,R3,R4}  ;Puts R4 high in memory, i.e. at end of stack
LDMFA R13!,{R3,R4}
LDMFA R13!,{R1,R2}
```

for an ascending stack.

The codes are synonyms as follows:

Code	Load	Store
EA	DB	IA
ED	IB	DA
FA	DA	IB
FD	IA	DB

The S bit controls two special functions, both of which are indicated to the assembler by putting "^" at the end of the instruction:

- If the S bit is set, the instruction is LDM and R15 is in the register list, then:

* In 26-bit privileged modes, all 32 bits of R15 will be loaded.

* In 26-bit user mode, the 4 flags and 24 PC bits of R15 will be loaded. Bits 27, 26, 1 and 0 of the loaded value will be ignored.

* In 32-bit modes, all 32 bits of R15 will be loaded, though note that the two bottom bits are always zero, so any ones loaded to them will be ignored. In addition, the SPSR of the current mode will be transferred to the CPSR; since user mode does not have an SPSR, this type of instruction should not be used in 32-bit user mode.
- If the S bit is set and either the instruction is STM or R15 is not in the register list, then the user mode registers will be transferred rather than those for the current mode. This type of instruction should not be used in user mode.

Special cases occur when the base register is used in the list of registers to be transferred.

- The base register can always be loaded without any problems. However, don't specify writeback if the base register is being loaded — you can't end up with both a written-back value and a loaded value in the base register!
- The base register can be stored with no complications as long as writeback is not used.
- Storing a list of registers including the base register using writeback will write the value of the base register before writeback to memory only if the base register is the first in the list. Otherwise, the value which is used is not defined.

Further special cases occur if the program counter is present in the list of registers to load and save.

- The PSR is always saved with the PC (in 26 bit modes) (and the PC will always be 12 bytes further on, rather than the usual 8 (in all modes)).
- On a load, only the bits of the PSR that are alterable in the current mode can be affected, and then only if the S bit is set.

The PC should not be used as the base register.

A block data load, takes nS + 1N + 1I + (1S + 1N if PC changed) cycles, and a block data store takes (n-1)S + 2N cycles, where "n" is the number of words being transferred.

5.8. Software interrupt

```
xxxx1111 yyyyyyyy yyyyyyyy yyyyyyyy
```

Typical Assembler Syntax:

```
        SWI   "OS_WriteI"
        SWINE &400C0
```

On encountering a software interrupt, the ARM switches into SVC mode, saves the current value of R15 into R14_SVC, and jumps to location 8 in memory, where it assumes it will find a SWI handling routine to decode the lower 24 bits of the SWI just executed, and do whatever the SWI number concerned means on that particular operating system.

An operating system written on the ARM will typically use SWIs to provide miscellaneous routines for programmers.

A SWI takes 2S + 1N cycles to execute (plus whatever time is required to decode the SWI number and execute the appropriate routines).

5.9. Co-processor data operations

```
xxxx1110 oooonnnn ddddpppp qqq0mmmm
```

Typical Assembler Syntax:

```
CDP p, o, CRd, CRn, CRm, q
CDP p, o, CRd, CRn, CRm
```

This instruction is passed on to co-processor p, telling it to perform operation o, on co-processor registers CRn and CRm, and place the result into Crd.

qqq may supply additional information about the operation concerned.

The exact meaning of these instructions depends on the particular co-processor in use; The above is only a recommended usage for the bits (and indeed the FPA doesn't conform to it). The only part which is obligatory is that pppp must be the coprocessor number: the coprocessor designer is free to allocate oooo, nnnn, dddd, qqq and mmmm as desired.

If the coprocessor uses the bits in a different way than the recommended one, assembler macros will probably be needed to translate the instruction syntax that makes sense to people into the correct CDP instruction. For commonly used coprocessors such as the FPA, many assemblers have the extra mnemonics built in and do this translation automatically. (For example, assembling MUFEZ F0,F1,#10 as its equivalent CDP 1,1,CR0,CR9,CR15,3.)

Currently defined co-processor numbers include:

1 and 2	Floating Point unit
15	Cache Controller

If a call to a coprocessor is made and the coprocessor does not respond (normally becuase it isn't there!), the undefined instruction vector is called (exactly as for one of the undefined instructions given later). This is used to transparently provide FP support on machines without an FPA.

These instructions take 1S + bI cycles to execute, where b is the number of cycles that the coprocessor causes the ARM to busy-wait before it accepts the instruction: again, this is under the coprocessor's control.

5.10. Co-processor data transfer and register transfers

```
xxxx110P UNWLnnnn DDDDpppp oooooooo LDC/STC
xxxx1110 oooLNNNN ddddpppp qqq1MMMM MRC/MCR
```

Again these depend on the particular co-processor p in use.

N and D signify co-processor register numbers, n and d are ARM processor numbers. o is the co-processor operation to use. M signifies bits the coprocessor is free to use as it wants.

The first form, denotes LDC if L=1, STC otherwise. The instruction behaves like LDR or STR respectively, in each case with an immediate offset, with the following exceptions.

- The offset is 4*(oooooooo), not a general 12-bit constant.
- If P=0 (post-indexing) is specified, W must be 1, and W being 1 just indicates that writeback is required, not that the memory system should be told that this is a user mode transfer. Instructions with P=0 and W=0 are reserved for future expansion.
- One or more coprocessor registers are loaded or stored. The coprocessor determines how many and which registers are to be loaded or stored from the DDDD and N bits: all the ARM does is transfer a word to or from the indicated address, then another to or from the indicated address + 4, then one to or from the indicated address + 8, etc., until the coprocessor tells it to stop.
- By convention, DDDD denotes the (first) coprocessor register to load or store and N denotes the length in some way, with N=1 indicating a "long" form. Coprocessor designers are free to ignore this...
- The assembler syntax is along the lines of:

```
LDC    p,CRd,[Rn,#20]   ;short form (N=0), pre-indexed
```

```
STCL   p,CRd,[Rn,#-32]! ;long form (N=1), pre-indexed with writeback
LDCNEL p,CRd,[Rn],#-100 ;long form (N=1), post-indexed
```

The second form denotes, MRC, if L=1, MCR otherwise. MRC transfers a coprocessor register to an ARM register, MCR the other way around (the letters may seem the wrong way around, but remember that destinations are usually written on the left in ARM assembler).

MCR transfers the contents of ARM register Rd to the coprocessor. The coprocessor is free to do whatever it wants with it based on the values of the ooo, dddd, qqq and MMMM fields, though as usual there is a "standard" interpretation: write it to coprocessor register CRN, using operation ooo, with possible additional control provided by CRM and qqq. The assembler syntax is:

```
MCR   p,o,Rd,CRN,CRM,q
```

Rd should not be R15 for an MCR instruction.

MRC transfers a single word from the coprocessor and puts it in ARM register Rd. The coprocessor is free to generate this word in any way it likes using the same fields as for MCR, with the standard interpretation that it comes from CRN using operation ooo, with possible additional control provided by CRM and qqq. The assembler syntax is:

```
MRC   p,o,Rd,CRN,CRM,q
```

If Rd is R15 for an MRC instruction, the top 4 bits of the word transferred are used to set the flags; the remaining 28 bits are discarded. (This is the mechanism used e.g. by floating point comparison instructions.)

LDC and STC take (n-1)S + 2N + bI cycles to execute, MRC takes 1S+bI+1C cycles, and MCR takes 1S + (b+1)I + 1C cycles, where b is the number of cycles that the coprocessor causes the ARM to busy-wait before it accepts the instruction: again, this is under the coprocessor's control, and n is the number of words being transferred (Note this is under the coprocessor's control, not the ARM's)

5.11. Single Data Swap (ARM 3 and later including ARM 2aS)

```
xxxx0001 0B00nnnn dddd0000 1001mmmm
```

Typical Assembler Syntax:

```
SWP Rd, Rm, [Rn]
```

These instructions load a word of memory (address given by register Rn) to a register Rd and store the contents of register Rm to the same address. Rm and Rd may be the same register, in which case the contents of this register and of the memory location are swapped. The load and store operations are locked together by setting the LOCK pin high during the operation to indicate to the memory manager that they should be allowed to complete without interruption.

If the B bit is set, then a byte of memory is transferred, otherwise a word is transferred.

None of Rd, Rn, and Rm may be R15.

This instruction takes 1S + 2N + 1I cycles to execute.

5.12. Status Register transfer (ARM 6 and later)

```
xxxx0001 0s10aaaa 11110000 0000mmmm  MSR  Register form
xxxx0011 0s10aaaa 1111rrrr bbbbbbbb  MSR  Immediate form
xxxx0001 0s001111 dddd0000 00000000  MRS
```

Typical Assembler Syntax:

```
MSR   SPSR_all, Rm          ;aaaa = 1001
MSR   CPSR_flg, #&F0000000  ;aaaa = 1000
MSRNE CPSR_ctl, Rm          ;aaaa = 0001
MRS   Rd, CPSR
```

The s bit, when set means access the SPSR of the current privileged mode, rather than the CPSR. This bit must only be set when executing the command in a privileged mode.

MSR is used for transfering a register or constant to a status register.

The aaaa bits can take the following values:

Value	Meaning
0001	Set the control bits of the PSR concerned.
1000	Set the flag bits of the PSR concerned.
1001	Set the control and flag bits of the PSR concerned (i.e. all the bits at present).

Other values of aaaa are reserved for future expansion.

In the register form, the source register is Rm. In the immediate form, the source is #b, ROR #2r.

R15 should not be specified as the source register of an MRS instruction.

MRS is used for transfering processor status to a register.

The d bits store the destination register number; Rd must not be R15.

N.B. The instruction encodings correspond to the data processing instructions with opcodes 10xx (i.e. the test instructions) and the S bit clear.

These instruction always execute in 1-S cycle.

5.13. Undefined instructions

```
xxxx0001 yyyyyyyy yyyyyyyy 1yy1yyyy ARM 2 only
xxxx011y yyyyyyyy yyyyyyyy yyy1yyyy
```

These instructions are currently undefined. On encountering an undefined instruction, the ARM switches to SVC mode (on ARM 3 and below) or Undef mode (on ARM 6 and above), puts the old value of R15 into R14_SVC (or R14_UND) and jumps to location, where it expects to find code to decode the undefined instruction and behave accordingly.

Notes:

- These instructions are documented as "undefined" because they enter the undefined instruction processor trap in this way. Plenty of other instructions are undefined in the looser sense that nothing says what they do. For instance, bit patterns of the form:

  xxxx0000 01xxxxxx xxxxxxxx 1001xxxx

  are related to data processing instructions, multiplies, long multiplies and SWPs, but are none of these because:

* Data processing instructions with bit 25 = 0 and bit 4 = 1 have register controlled shifts, and so must have bit 7 = 0.
* Multiply instructions have bits 23:22 = 00.
* Long multiply instructions have bits 23:22 = 1U.
* SWPs have bit 24 = 1.

What these instructions do simply isn't defined, whereas the ones listed above are actually **defined** to enter the undefined instruction trap, at least until some future use is found for them.

- Note that the "ARM2 only" undefined instructions include those that became SWP instructions on ARM3/ARM2as and later.

6. Credits

This document was originally written by Robin Watts, with considerable consultation with Steven Singer. It was then later updated by Mark Smith to include more information on ARMs later than 2.

David Seal provided a huge list of corrections and amendments, and unwittingly provided the basis for the timing information in a posting to usenet.

Various corrections were also submitted/posted by Olly Betts, Clive Jones, Alain Noullez, John Veness, Sverker Wiberg and Mark Wooding.

Thanks to everyone that helped (and if I have missed you here, please let me know.)

Just because I have included peoples addresses here, please do not take this as an invitation to mail them any questions you may have!

Olly Betts	olly@mantis.co.uk
Paul Hankin	pdh13@cus.cam.ac.uk
Robert Harley	robert@edu.caltech.cs
Clive Jones	Clive.Jones@armltd.co.uk
Alain Noullez	anoullez@zig.inria.fr
David Seal	<address withheld by request>
Steven Singer	s.singer@ph.surrey.ac.uk
Mark Smith	ee91mds2@brunel.ac.uk
John Veness	john@uk.ac.ox.drl
Robin Watts	Robin.Watts@comlab.ox.ac.uk
Sverker Wiberg	sverkerw@Student.csd.UU.SE
Mark Wooding	csuov@csv.warwick.ac.uk

For those not on the internet, messages can be sent by snail mail to:

Robin Watts
St Catherines College,
Oxford,
OX1 3UJ