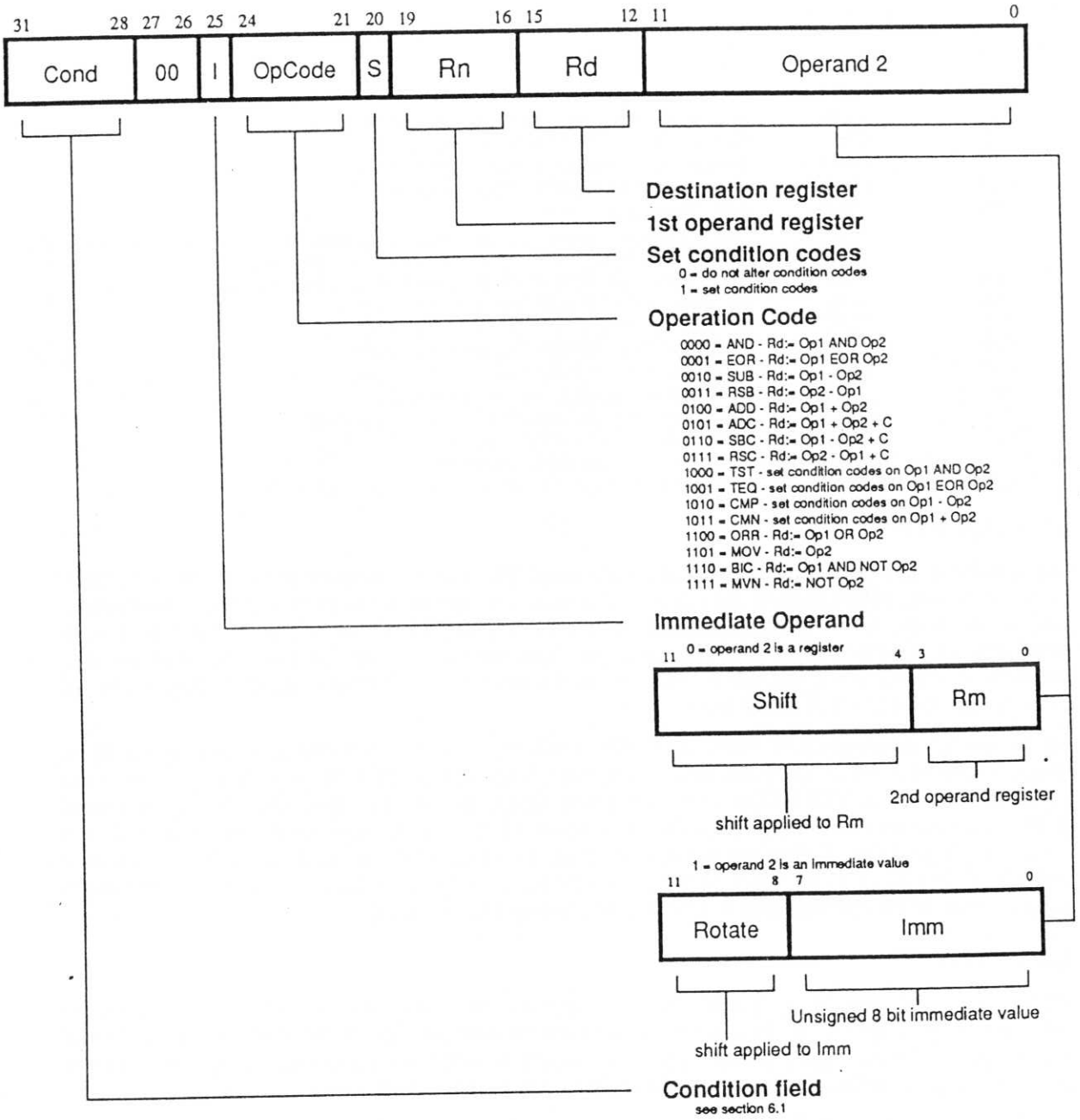
6.3 Data processing



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

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands. The first operand is always a register (Rn). The second operand may be a shifted register (Rm) or a rotated 8 bit immediate value (Imm) according to the value of the I bit in the instruction. The condition codes in the PSR may be preserved or updated as a result of this instruction, according to the value of the S bit in the instruction. Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result, and therefore should always have the S bit set. (The assembler treats TST, TEQ, CMP and CMN as TSTS, TEQS, CMPS and CMNS by default.)

6.3.1 Operations

The operations supported are:

Assembler Mnemonic	OpCode	Action
AND	0000	Bit-wise logical AND of operands
EOR	0001	Bit-wise logical EOR of operands
SUB	0010	Subtract operand 2 from operand 1
RSB	0011	Subtract operand 1 from operand 2
ADD	0100	Add operands
ADC	0101	Add operands plus carry (PSR C flag)
SBC	0110	Subtract operand 2 from operand 1 plus carry
RSC	0111	Subtract operand 1 from operand 2 plus carry
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	Bit-wise logical OR of operands
MOV	1101	Move operand 2 (operand 1 is ignored)
BIC	1110	Bit clear (bit-wise logical AND of operand 1 and NOT operand 2)
MVN	1111	Move NOT operand 2 (operand 1 is ignored)

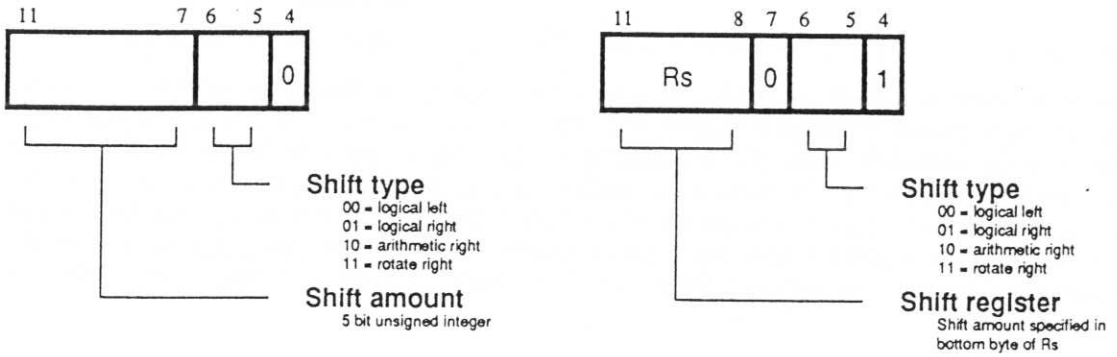
6.3.2 PSR flags

The operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result. If the S bit is set (and Rd is not R15, see below) the V flag in the PSR will be unaffected, the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0), the Z flag will be set if and only if the result is all zeroes, and the N flag will be set to the logical value of bit 31 of the result.

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32 bit integer (either unsigned or 2's complement signed, the two are equivalent). If the S bit is set (and Rd is not R15) the V flag in the PSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed. The C flag will be set to the carry out of bit 31 of the ALU, the Z flag will be set if and only if the result was zero, and the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

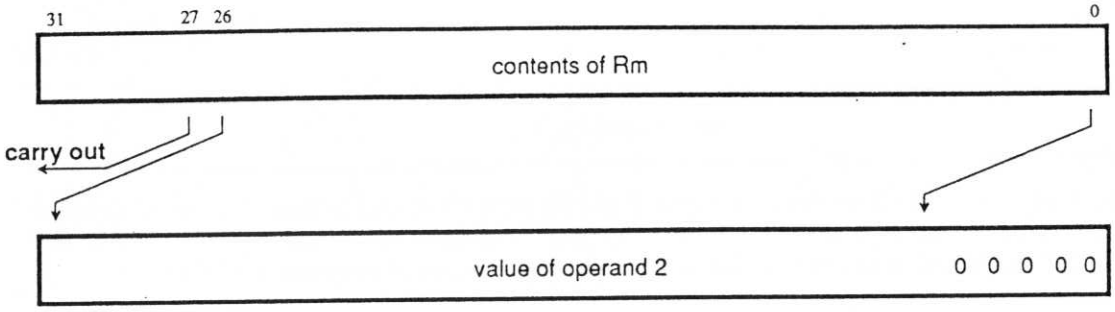
6.3.3 Shifts

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



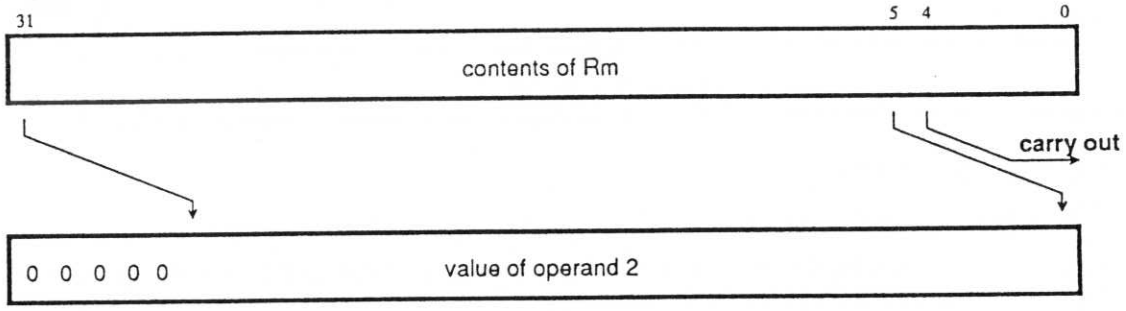
Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeroes, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomed the shifter carry output which may be latched into the C bit of the PSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is:



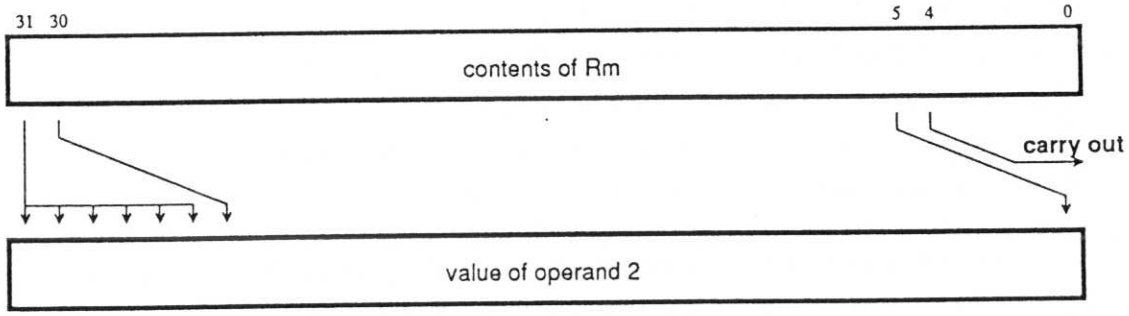
Note that LSL #0 is a special case, where the shifter carry out is the old value of the PSR C flag. The contents of Rm are used directly as the second operand.

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



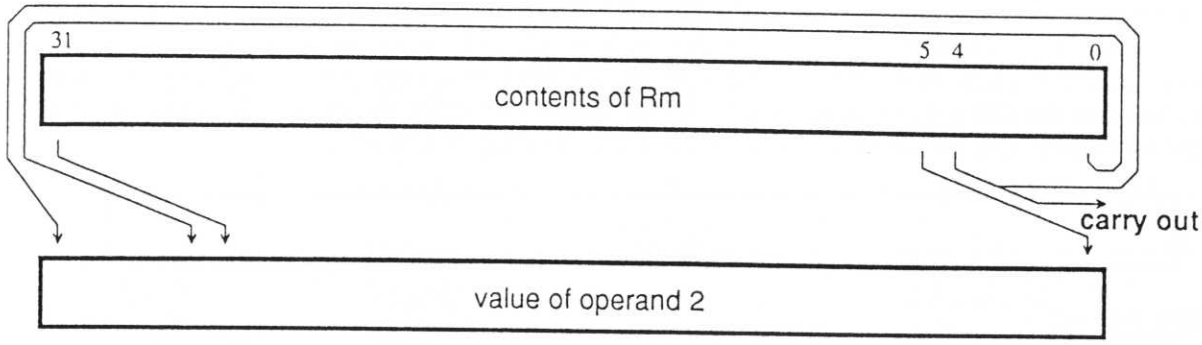
The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeroes. This preserves the sign in 2's complement notation. For example, ASR #5:

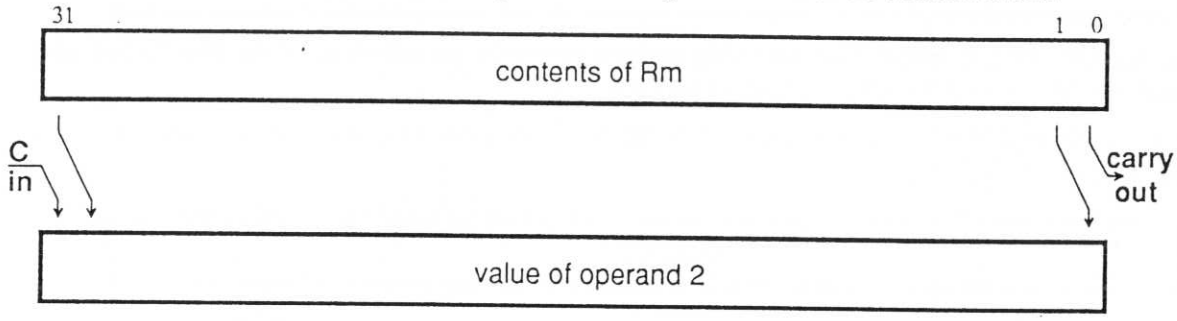


The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeroes, according to the value of bit 31 of Rm.

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeroes used to fill the high end in logical right operations. For example, ROR #5:



The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the PSR C flag to the most significant end of the contents of Rm:



Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount.

If this byte is zero, the unchanged contents of Rm will be used as the second operand, and the old value of the PSR C flag will be passed on as the shifter carry output.

If the byte has a value between 1 and 31, the shifted result will exactly match that of an instruction specified shift with the same value and shift operation.

If the value in the byte is 32 or more, the result will be a logical extension of the shifting processes described above:

* LSL by 32 has result zero, carry out equal to bit 0 of Rm.
* LSL by more than 32 has result zero, carry out zero.
* LSR by 32 has result zero, carry out equal to bit 31 of Rm.
* LSR by more than 32 has result zero, carry out zero.
* ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.
* ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.
* ROR by n where n is greater than 32 will give the same result and carry out as ROR by n-32; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Note that the zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or an undefined instruction.

6.3.4 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. The immediate value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2. Another example is that the 8 bit constant may be aligned with the PSR flags (bits 0, 1, and 26 to 31). All the flags can thereby be initialised in one TEQP instruction (see section 6.2.5).

6.3.5 Writing to R15

When Rd is a register other than R15, the condition code flags in the PSR may be updated from the ALU flags as described above. When Rd is R15 and the S flag in the instruction is set, the PSR is overwritten by the corresponding bits in the ALU result, so bit 31 of the result goes to the N flag, bit 30 to the Z flag, bit 29 to the C flag and bit 28 to the V flag. In user mode the other flags (I, F, M1, M0) are protected from direct change, but in non-user modes these will also be affected, accepting copies of bits 27, 26, 1 and 0 of the result respectively.

When one of these instructions is used to change the processor mode (which is only possible in a non-user mode), the following instruction should not access a banked register (R8-R14) during its first cycle. A no-op should be inserted if the next instruction must access a banked register. Accesses to the unbanked registers (R0-R7 and R15) are safe.

If the S flag is clear when Rd is R15, only the 24 PC bits of R15 will be written. Conversely, if the instruction is of a type which does not normally produce a result (CMP, CMN, TST, TEQ) but Rd is R15 and the S bit is set, the result will be used in this case to update those PSR flags which are not protected by virtue of the processor mode.

6.3.6 Using R15 as an operand

If R15 is used as an operand in a data processing instruction it can present different values depending on which operand position it occupies. It will always contain the value of the PC. It may or may not contain the values of the PSR flags as they were at the completion of the previous instruction.

When R15 appears in the Rm position it will give the value of the PC together with the PSR flags to the barrel shifter.

When R15 appears in either of the Rn or Rs positions it will give the value of the PC alone, with the PSR bits replaced by zeroes.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount, the PC will be 8 bytes ahead when used as Rs, and 12 bytes ahead when used as Rn or Rm.

6.3.7 Assembler syntax

* MOV,MVN - single operand instructions

  <opcode>{cond}{S} Rd,<Op2>

* CMP,CMN,TEQ,TST - instructions which do not produce a result.

  <opcode>{cond}{P} Rn,<Op2>

* AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC

  <opcode>{cond}{S} Rd,Rn,<Op2>

where <Op2> is **Rm{,<shift>}** or **,<#expression>**

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

{S} - set condition codes if S present (implied for CMP, CMN, TEQ, TST).

{P} - make Rd = R15 in instructions where Rd is not specified, otherwise Rd will default to R0. (Used for changing the PSR directly from the ALU result.)

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

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

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

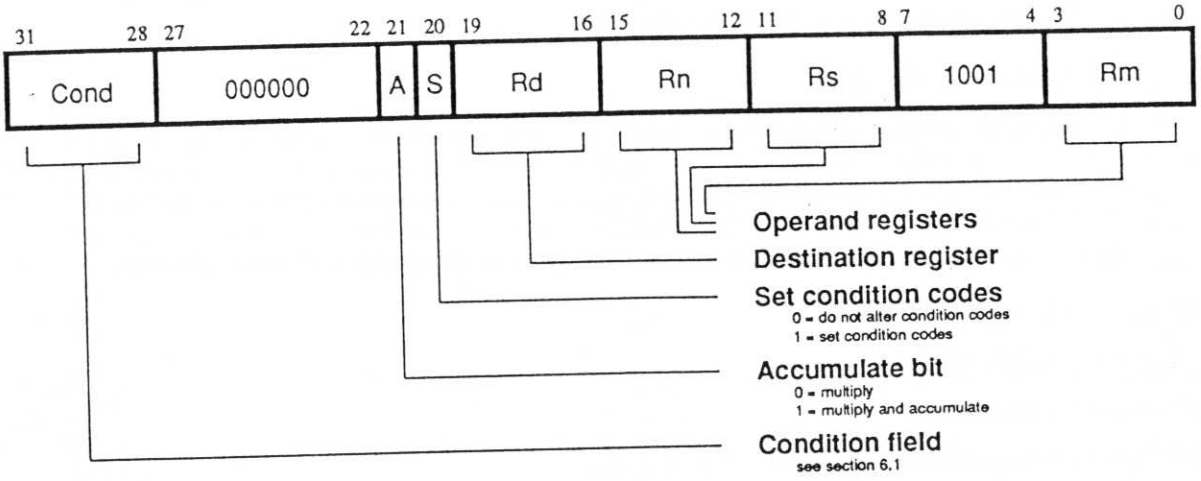
<shiftname>s are: ASL, LSL, LSR, ASR, ROR.

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

6.3.8 Examples

```
ADDEQ R2,R4,R5          ; if the Z flag is set make R2:=R4+R5

TEQS R4,#3              ; test R4 for equality with 3
                        ; (the S is in fact redundant as the
                        ;  assembler inserts it automatically)

SUB R4,R5,R7,LSR R2     ; logical right shift R7 by the number in
                        ; the bottom byte of R2, subtract the result
                        ; from R5, and put the answer into R4


                        ; assume non-user mode here
TEQP R15,#0             ; Change to user mode and clear N,Z,C,V,I,F
                        ; NB R15 is here in the Rn position, so it
                        ; comes without the PSR flags
MOVNV R0,R0             ; no-op to avoid mode change hazard
MOV PC,R14              ; return from subroutine (R14 is a banked register)

MOVS PC,R14             ; return from subroutine and restore the PSR
```

6.4 Multiply and multiply-accumulate (MUL, MLA)



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

The multiply and multiply-accumulate instructions use a 2 bit Booth's algorithm to perform integer multiplication. They give the least significant 32 bits of the product of two 32 bit operands, and may be used to synthesize higher precision multiplications.

The multiply form of the instruction gives Rd:=Rm*Rs. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives Rd:=Rm*Rs+Rn, which can save an explicit ADD instruction in some circumstances.

Both forms of the instruction work on operands which may be considered as signed (2's complement) or unsigned integers.

6.4.1 Operand restrictions

Due to the way the Booth's algorithm has been implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the Rm operand register, as Rd is used to hold intermediate values and Rm is used repeatedly during the multiply. A MUL will give a zero result if Rm=Rd, and a MLA will give a meaningless result.

The destination register (Rd) should also not be R15. R15 is protected from modification by these instructions, so the instruction will have no effect, except that it will put meaningless values in the PSR flags if the S bit is set.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

6.4.2 PSR flags

Setting the PSR flags is optional, and is controlled by the S bit in the instruction. The N and Z flags are set correctly on the result (N is equal to bit 31 of the result, Z is set if and only if the result is zero), the V flag is unaffected by the instruction (as for logical data processing instructions), and the C flag is set to a meaningless value.

6.4.3 Writing to R15

As mentioned above, R15 must not be used as the destination register (Rd). If it is so used, the instruction will have no effect except possibly to scramble the PSR flags.

6.4.4 Using R15 as an operand

R15 may be used as one or more of the operands, though the result will rarely be useful. When used as Rs the PC bits will be used without the PSR flags, and the PC value will be 8 bytes on from the address of the multiply instruction. When used as Rn, the PC bits will be used along with the PSR flags, and the PC will again be 8 bits on from the address of the instruction. When used as Rm, the PC bits will be used together with the PSR flags, but the PC will be the address of the instruction plus 12 bytes in this case.

6.4.5 Assembler syntax

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

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

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

{S} - set condition codes if S present.

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

(Rd must not be R15 and must not be the same as Rm.)

6.4.6 Examples

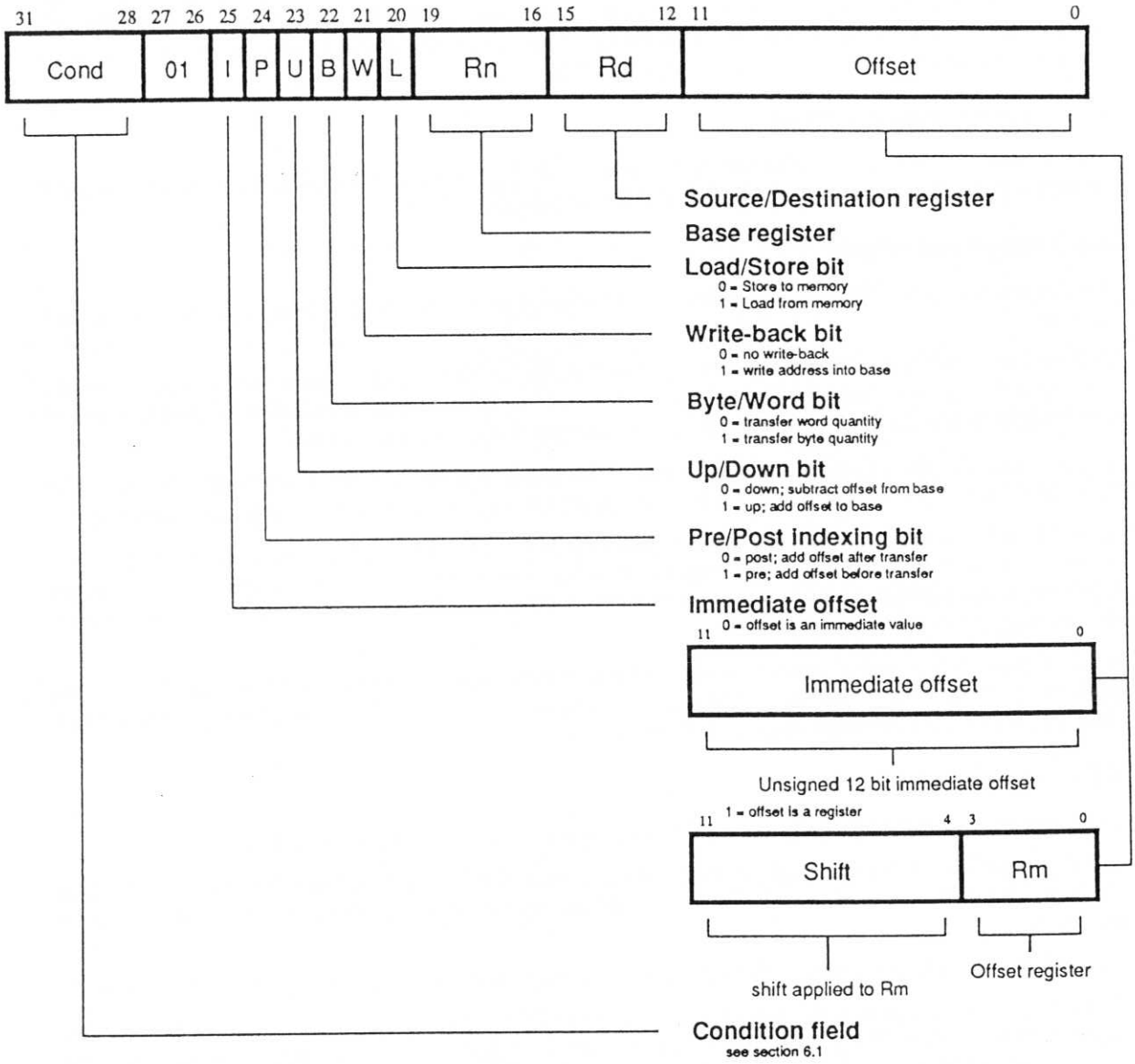
```
MUL R1,R2,R3          ; R1:=R2*R3

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

The multiply instruction may be used to synthesize higher precision multiplications, for instance to multiply two 32 bit integers and generate a 64 bit result:

```
mul64
  MOV   a1,A,LSR #16    ; a1:= top half of A
  MOV   D,B,LSR #16     ; D := top half of B
  BIC   A,A,a1,LSL #16  ; A := bottom half of A
  BIC   B,B,D,LSL #16   ; B := bottom half of B
  MUL   C,A,B           ; low section of result
  MUL   B,a1,B          ; ) middle sections
  MUL   A,D,A           ; )     of result
  MUL   D,a1,D          ; high section of result
  ADDS  A,B,A           ; add middle sections
                        ;  (couldn't use MLA as we need C correct)
  ADDCS D,D,#&10000     ; carry from above add
  ADDS  C,C,A,LSL #16   ; C is now bottom 32 bits of product
  ADC   D,D,A,LSR #16   ; D is top 32 bits
```

(A, B are registers containing the 32 bit integers; C, D are registers for the 64 bit result; a1 is a temporary register. A and B are overwritten during the multiply)

6.5 Single data transfer (LDR, STR)



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

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calulated by adding an offset to or subtracting an offset from a base register. The result of this calculation may be written back into the base register if 'auto-indexing' is required.

6.5.1 Offsets and auto-indexing

The offset from the base may be either a 12 bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to (U=1) or subtracted from (U=0) the base register Rn. The offset modification may be performed either before (pre-indexed, P=1) or after (post-indexed, P=0) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base (W=1), or the old base value may be kept (W=0). In the case of post-indexed

addressing, the write back bit is redundant, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in non-user mode code, where setting the W bit forces the $\overline{\text{TRANS}}$ pin to go LOW for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this pin.

6.5.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions (section 6.2.3), but the register specified shift amounts are not available in this instruction class.

6.5.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM register and memory.

A byte load (LDRB) expects the data on bits 0 to 7 if the supplied address is on a word boundary, on bits 8 to 15 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeroes.

A byte store (STRB) repeats the bottom 8 bits of the source register four times across the data bus. The external memory system should activate the appropriate byte subsystem to store the data (see chapter 7).

A word load (LDR) should generate a word aligned address. An address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. External hardware could perform a double access to memory to allow non-aligned word loads, but existing systems do not support this.

A word store (STR) should generate a word aligned address. The data presented to the data bus are not affected if the address is not word aligned, so if support were required for non-aligned stores external hardware would have to switch bytes around on the bus.

6.5.4 Use of R15

These instructions will never cause the PSR to be modified, even when Rd or Rn is R15.

If R15 is specified as the base register (Rn), the PC is used without the PSR flags. When using the PC as the base register one must remember that it contains an address 8 bytes on from the address of the current instruction.

If R15 is specified as the register offset (Rm), the value presented will be the PC together with the PSR.

When R15 is the source register (Rd) of a register store (STR) instruction, the value stored will be the PC together with the PSR. The stored value of the PC will be 12 bytes on from the address of the instruction. A load register (LDR) with R15 as Rd will change only the PC, and the PSR will be unchanged.

6.5.5 Address exceptions

If the address used for the transfer (ie the unmodified contents of the base register for post-indexed addressing, or the base modified by the offset for pre-indexed addressing) has a logic one in any of the bits 26 to 31, the transfer will not take place and the address exception trap will be taken.

Note that it is only the address actually used for the transfer which is checked. A base containing an address outside the legal range may be used in a pre-indexed transfer if the offset brings the address within the legal range, and likewise a base within the legal range may be modified by post-indexing to outside the legal range without causing an address exception.

6.5.6 Data Aborts

A transfer to or from a legal address may still cause problems for a memory management system. For instance, in a system which uses virtual memory the required data may be absent from main memory. The memory manager can signal a problem by taking the processor **ABORT** pin HIGH, whereupon the data transfer instruction will be prevented from changing the processor state and the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

6.5.7 Assembler syntax

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

LDR - load from memory into a register.

STR - store from a register into memory.

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

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

{T} - if T is present the W bit will be set in a post-indexed instruction, causing the $\overline{\text{TRANS}}$ pin to go LOW for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid 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

  [Rn,{+/-}Rm{,<shift>}]{!} offset of +/- contents of index register, shifted by <shift>.

* A post-indexed addressing specification:

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

  [Rn],{+/-}Rm{,<shift>} offset of +/- contents of index register, shifted as by <shift>.

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

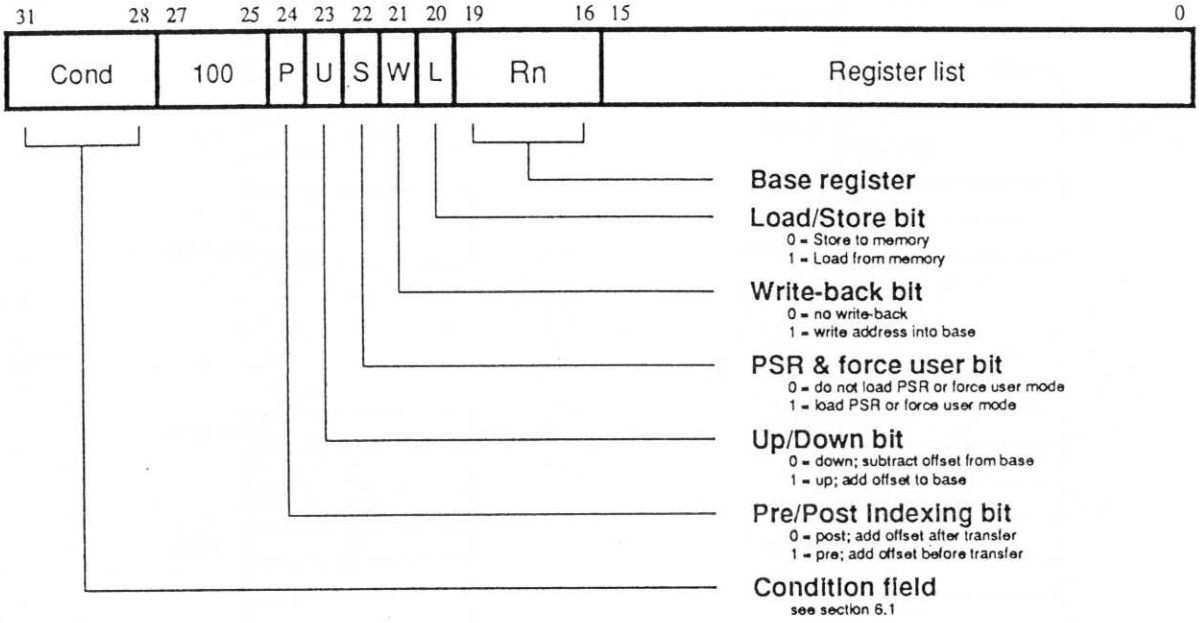
<shift> is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.

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

6.5.8 Examples

```
    STR R1,[BASE,INDEX]!        ; store R1 at BASE+INDEX (both of which are
                                ;  registers) and write back address to BASE

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

    LDR R1,[BASE,#16]           ; load R1 from contents of BASE+16.
                                ;  Don't write back

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

    LDREQB R1,[BASE,#5]         ; conditionally load byte at BASE+5 into
                                ;  R1 bits 0 to 7, filling bits 8 to 31 with zeroes

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

6.6 Block data transfer (LDM, STM)



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

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

6.6.1 The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

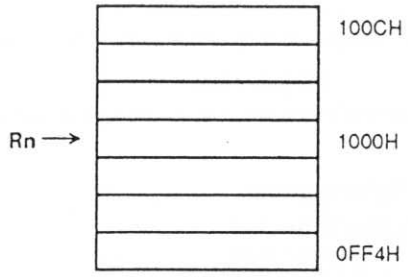
Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

6.6.2 Addressing modes

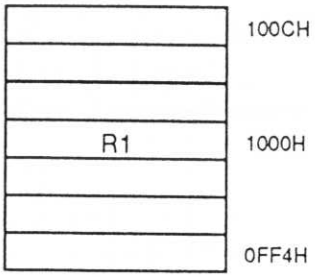
The transfer addresses are determined by the contents of the base register (Rn), the pre/post bit (P) and the up/down bit (U). The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address. By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=1000H and write back of the modified base is required (W=1). The following figures show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

(In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 1000H unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.)

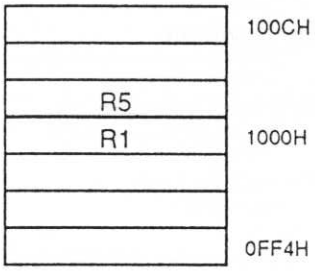
Post-increment addressing



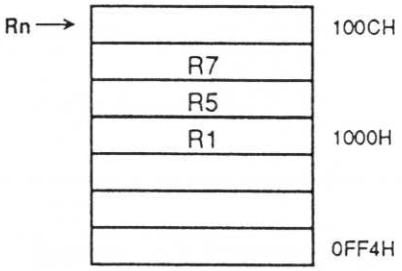
(1)



(2)

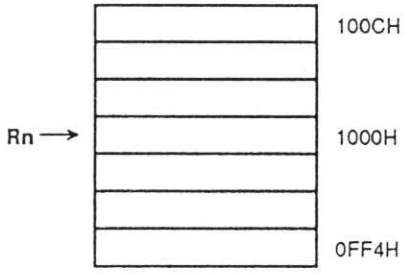


(3)

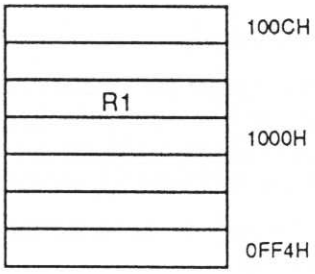


(4)

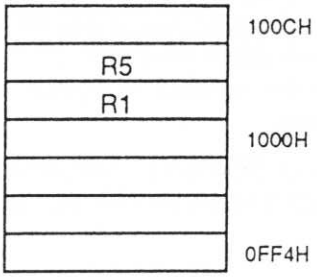
Pre-increment addressing



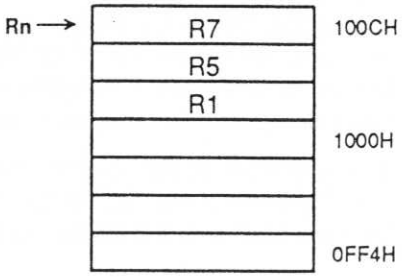
(1)



(2)

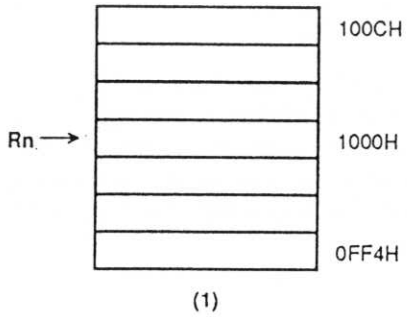


(3)

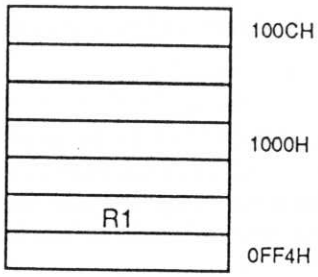


(4)

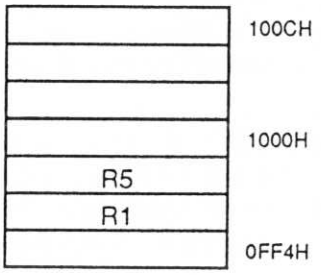
Post-decrement addressing



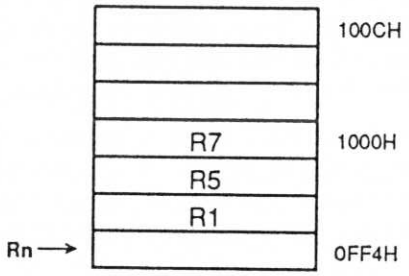
(1)



(2)

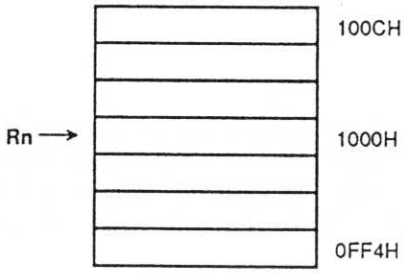


(3)

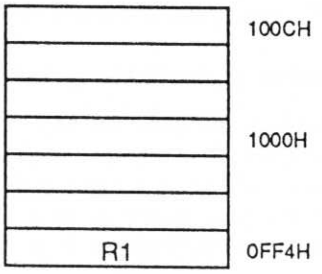


(4)

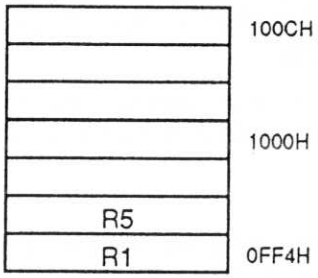
Pre-decrement addressing



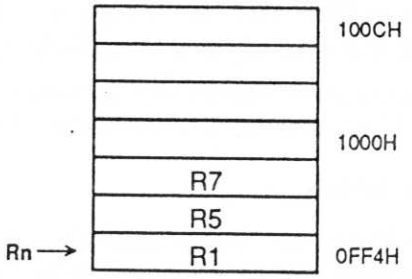
(1)



(2)



(3)



(4)

6.6.3 Transfer of R15

Whenever R15 is stored to memory, the value transferred is the PC together with the PSR flags. The stored value of the PC will be 12 bytes on from the address of the STM instruction.

If R15 is in the transfer list of a load multiple (LDM) instruction the PC is overwritten, and the effect on the PSR is controlled by the S bit. If the S bit is 0 the PSR is preserved unchanged, but if the S bit is 1 the PSR will be overwritten by the corresponding bits of the loaded value. In user mode, however, the I, F, M0 and M1 bits are protected from change whatever the value of the S bit. The mode at the start of the instruction determines whether these bits are protected, and the supervisor may return to the user program, reenabling interrupts and restoring user mode with one LDM instruction.

6.6.4 Forcing transfer of the user bank

For STM instructions the S bit is redundant as the PSR is always stored with the PC whenever R15 is in the transfer list. In user mode programs the S bit is ignored, but in other modes it has a second interpretation. S=1 used to force transfers to take values from the user register bank instead of from the current register bank. This is useful for saving the user state on process switches. Note that when it is so used, write back of the base 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.

In LDM instructions the S bit is redundant if R15 is not in the transfer list, and again in user mode programs it is ignored in this case. In non-user mode programs where R15 is not in the transfer list, S=1 is used to force loaded values to go to the user registers instead of the current register bank. When so used, care must be taken not to read from a banked register during the following cycle - if in doubt insert a no-op. Again don't use write back when forcing user bank transfer.

6.6.5 Use of R15 as the base

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 (setting the W bit will have no effect).

6.6.6 Inclusion of the base in the register list

When writeback is specified, the base is written back at the end of the second cycle of the instruction. During a STM, the first register is written out at the start of the second cycle. A STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. An LDM will always overwrite the updated base if the base is in the list.

6.6.7 Address exceptions

When the address of the first transfer falls outside the legal address space (ie has a 1 somewhere in bits 26 to 31), an address exception trap will be taken. The instruction will first complete in the usual number of cyles, though an STM will be prevented from writing to memory. The processor state will be the same as if a data abort had occurred on the first transfer cycle (see next section).

Only the address of the first transfer is checked in this way; if subsequent addresses over- or under-flow into illegal address space they will be truncated to 26 bits but will not cause an address exception trap.

6.6.8 Data Aborts

Some legal addresses may be unacceptable to a memory management system, and the memory manager can indicate a problem with an address by taking the **ABORT** pin HIGH. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, ARM takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When ARM detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

* Overwriting of registers stops when the abort happens. The aborting load will not take place, nor will the preceding one, but registers two or more positions ahead of the abort (if any) will be loaded. (This guarantees that the PC will be preserved, since it is always the last register to be overwritten.)
* The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

6.6.9 Assembler syntax

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

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

Rn is an expression evaluating to a valid register number.

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

{!} if present requests write-back (W=1), otherwise W=0.

{^} if present set S bit to load the PSR with the PC, or force transfer of user bank when in non-user mode.

Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalences between the names and the values of the bits in the instruction are:

name	stack	other	L bit	P bit	U bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required. The F and E refer to a "full" or "empty" stack, i.e. whether a pre-index has to be done (full) before storing to the stack. The A and D refer to whether the stack is ascending or descending. If ascending, a STM will

go up and LDM down, if descending, vice-versa.

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

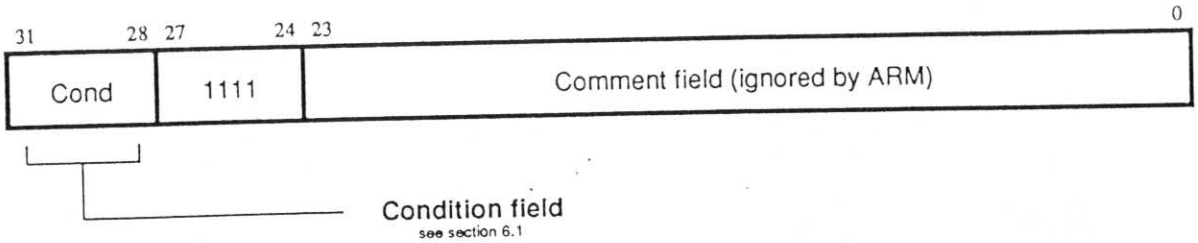
6.6.10 Examples

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

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

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

```
STMED SP!,{R0-R3,R14}  ; save R0 to R3 to use as workspace
                       ;  and R14 for returning

BL somewhere           ; this nested call will overwrite R14

LDMED SP!,{R0-R3,R15}^ ; restore workspace and return
                       ;  (also restoring PSR flags)
```

6.7 Software interrupt



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

The software interrupt instruction is used to enter supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change but forces the PC to a fixed value (08H). If this address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

6.7.1 Return from the supervisor

The PC and PSR are saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS R15,R14_svc will return to the user program, restore the user PSR and return the processor to user mode.

Note that the link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address.

6.7.2 Comment field

The bottom 24 bits of the instruction are ignored by ARM, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

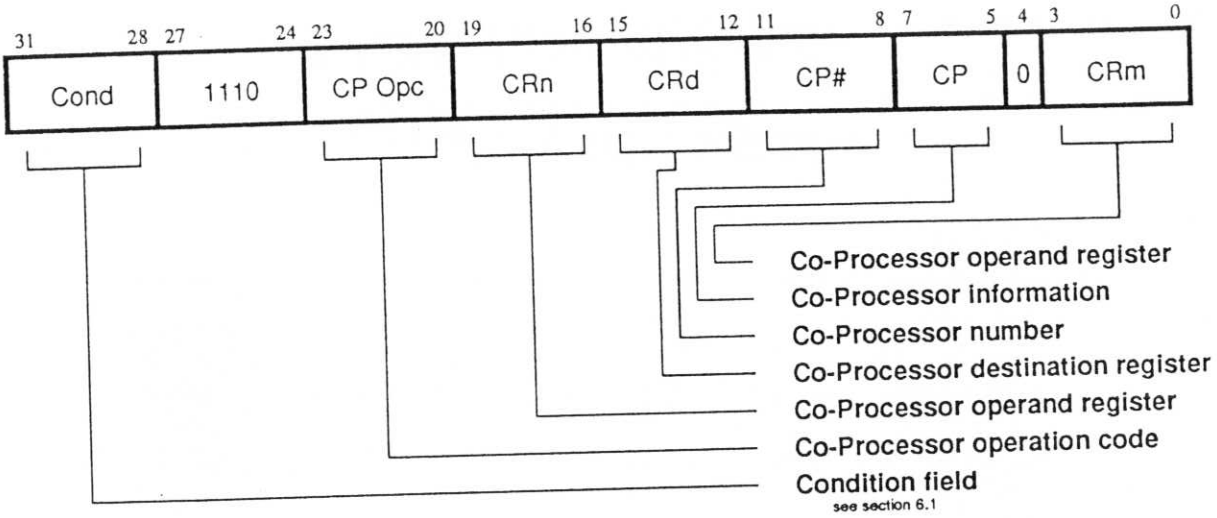
6.7.3 Assembler syntax

SWI{cond} <expression>

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

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

6.8 Co-Processor data operations



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 tell a Co-Processor to perform some internal operation. No result is communicated back to ARM, and ARM will not wait for the operation to complete. The Co-Processor could contain a queue of such instructions awaiting execution, and their execution can overlap other ARM activity allowing the Co-Processor and ARM to perform independent tasks in parallel.

6.8.1 The Co-Processor fields

Only bit 4 and bits 24 to 31 are significant to ARM; the remaining bits are used by Co-Processors. The above field names are used by convention, and particular Co-Processors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each Co-Processor, and a Co-Processor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the Co-Processor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

6.8.2 Assembler syntax

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

{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.

CRd, 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.

6.8.3 Examples

```
CDP 1,10,CR1,CR2,CR3     ; request Co-Proc 1 to do operation 10
                         ;  on CR2 and CR3, and put the result in CR1

CDPEQ 2,5,CR1,CR2,CR3,2 ; if Z flag is set request Co-Proc 2 to do
                         ;  operation 5 (type 2) on CR2 and CR3,
                         ;  and put the result in CR1
```