B1

APPENDIX

ARM and Thumb Assembler Instructions

Andrew Sloss,
ARM; *Dominic Symes,* ARM;
Chris Wright,
Ultimodule Inc.

This appendix lists the ARM and Thumb instructions available up to, and including, ARM architecture ARMv6, which was just released at the time of writing. We list the operations in alphabetical order for easy reference. Sections B1.5 and B1.4 give quick reference guides to the ARM and GNU assemblers *armasm* and *gas*.

We have designed this appendix for practical programming use, both for writing assembly code and for interpreting disassembly output. It is not intended as a definitive architectural ARM reference. In particular, we do not list the exhaustive details of each instruction bitmap encoding and behavior. For this level of detail, see the *ARM Architecture Reference Manual*, edited by David Seal, published by Addison Wesley. We do give a summary of ARM and Thumb instruction set encodings in Appendix B2.

B1.1 Using This Appendix

Each appendix entry begins by enumerating the available instructions formats for the given instruction class. For example, the first entry for the instruction class ADD reads

```
1. ADD<cond>{S} Rd, Rn, #<rotated_immed>     ARMv1
```

The fields <cond> and <rotated_immed> are two of a number of standard fields described in Section B1.2. Rd and Rn denote ARM registers. The instruction is only executed if the condition <cond> is passed. Each entry also describes the action of the instruction if it is executed.

The {S} denotes that you may apply an optional S suffix to the instruction. Finally, the right-hand column specifies that the instruction is available from the listed ARM architecture version onwards. Table B1.1 shows the entries possible for this column.

TABLE B1.1 Instruction types.

Type	Meaning
ARMvX	32-bit ARM instruction first appearing in ARM architecture version *X*
THUMBvX	16-bit Thumb instruction first appearing in Thumb architecture version *X*
MACRO	Assembler pseudoinstruction

Note that there is no direct correlation between the Thumb architecture number and the ARM architecture number. The THUMBv1 architecture is used in ARMv4T processors; the THUMBv2 architecture, in ARMv5T processors; and the THUMBv3 architecture, in ARMv6 processors.

Each instruction definition is followed by a notes section describing restrictions on the use of the instruction. When we make a statement such as " *Rd* must not be *pc*," we mean that the description of the function only applies when this condition holds. If you break the condition, then the instruction may be unpredictable or have predictable effects that we haven't had space to describe here. Well-written programs should not need to break these conditions.

B1.2 Syntax

We use the following syntax and abbreviations throughout this appendix.

Optional Expressions

- *{<expr>}* is an optional expression. For example, LDR{B} is shorthand for LDR or LDRB.
- *{<exp1>|<exp2>|...|<expN>}*, including at least one "|" divider, is a list of expressions. One of the listed expressions must appear. For example LDR{B|H} is shorthand for LDRB or LDRH. It does not include LDR. We would represent these three possibilities by LDR{|B|H}.

Register Names

- *Rd*, *Rn*, *Rm*, *Rs*, *RdHi*, *RdLo* represent ARM registers in the range *r0* to *r15*.
- *Ld*, *Ln*, *Lm*, *Ls* represent low-numbered ARM registers in the range *r0* to *r7*.
- *Hd*, *Hn*, *Hm*, *Hs* represent high-numbered ARM registers in the range *r8* to *r15*.
- *Cd*, *Cn*, *Cm* represent coprocessor registers in the range *c0* to *c15*.
- *sp*, *lr*, *pc* are names for *r13*, *r14*, *r15*, respectively.
- *Rn*[*a*] denotes bit *a* of register *Rn*. Therefore *Rn*[*a*] = (*Rn* » *a*) & 1.
- *Rn*[*a:b*] denotes the $a + 1 - b$ bit value stored in bits *a* to *b* of *Rn* inclusive.
- *RdHi*:*RdLo* represents the 64-bit value with high 32 *RDHi* bits and low 32 bits *RdLo*.

Values Stored as Immediates

- `<immedN>` is any unsigned *N*-bit immediate. For example, `<immed8>` represents any integer in the range 0 to 255. `<immed5>*4` represents any integer in the list 0, 4, 8, ..., 124.
- `<addressN>` is an address or label stored as a relative offset. The address must be in the range $pc - 2^N \leq address < pc + 2^N$. Here, *pc* is the address of the instruction plus eight for ARM state, or the address of the instruction plus four for Thumb state. The address must be four-byte aligned if the destination is an ARM instruction or two-byte aligned if the destination is a Thumb instruction.
- `<A-B>` represents any integer in the range *A* to *B* inclusive.
- `<rotated_immed>` is any 32-bit immediate that can be represented as an eight-bit unsigned value rotated right (or left) by an even number of bit positions. In other words, `<rotated_immed>` = `<immed8> ROR (2*<immed4>)`. For example `0xff`, `0x104`, `0xe0000005`, and `0x0bc00000` are possible values for <rotated_immed>. However, `0x101` and `0x102` are not. When you use a rotated immediate, `<shifter_C>` is set according to Table B1.3 (discussed in Section *Shift Operations*). A nonzero rotate may cause a change in the carry flag. For this reason, you can also specify the rotation explicitly, using the assembly syntax `<immed8>, 2*<immed4>`.

Condition Codes and Flags

- `<cond>` represents any of the standard ARM condition codes. Table B1.2 shows the possible values for `<cond>`.

TABLE B1.2 ARM condition mnemonics.

<cond>	Instruction is executed when	*cpsr* condition
{|AL}	ALways	TRUE
EQ	EQual (last result zero)	Z==1
NE	Not Equal (last result nonzero)	Z==0
{CS|HS}	Carry Set, unsigned Higher or Same (following a compare)	C==1
{CC|LO}	Carry Clear, unsigned LOwer (following a comparison)	C==0
MI	MInus (last result negative)	N==1
PL	PLus (last result greater than or equal to zero)	N==0
VS	V flag Set (signed overflow on last result)	V==1
VC	V flag Clear (no signed overflow on last result)	V==0
HI	unsigned HIgher (following a comparison)	C==1 && Z==0
LS	unsigned Lower or Same (following a comparison)	C==0 || Z==1
GE	signed Greater than or Equal	N==V
LT	signed Less Than	N!=V
GT	signed Greater Than	N==V && Z==0
LE	signed Less than or Equal	N!=V || Z==1
NV	NeVer—ARMv1 and ARMv2 only—*DO NOT USE*	FALSE

- `<SignedOverflow>` is a flag indicating that the result of an arithmetic operation suffered from a signed overflow. For example, `0x7fffffff` + 1 = `0x80000000` produces a signed overflow because the sum of two positive 32-bit signed integers is a negative 32-bit signed integer. The *V* flag in the *cpsr* typically records signed overflows.
- `<UnsignedOverflow>` is a flag indicating that the result of an arithmetic operation suffered from an unsigned overflow. For example, `0xffffffff` + 1 = 0 produces an overflow in unsigned 32-bit arithmetic. The *C* flag in the *cpsr* typically records unsigned overflows.
- `<NoUnsignedOverflow>` is the same as 1 – `<UnsignedOverflow>`.

- <Zero> is a flag indicating that the result of an arithmetic or logical operation is zero. The *Z* flag in the *cpsr* typically records the zero condition.
- <Negative> is a flag indicating that the result of an arithmetic or logical operation is negative. In other words, <Negative> is bit 31 of the result. The *N* flag in the *cpsr* typically records this condition.

Shift Operations

- <imm_shift> represents a shift by an immediate specified amount. The possible shifts are LSL #<0-31>, LSR #<1-32>, ASR #<1-32>, ROR #<1-31>, and RRX. See Table B1.3 for the actions of each shift.
- <reg_shift> represents a shift by a register-specified amount. The possible shifts are LSL Rs, LSR Rs, ASR Rs, and ROR Rs. *Rs* must not be *pc*. The bottom eight bits of *Rs* are used as the shift value *k* in Table B1.3. Bits *Rs*[31:8] are ignored.
- <shift> is shorthand for <imm_shift> or <reg_shift>.
- <shifted_Rm> is shorthand for the value of *Rm* after the specified shift has been applied. See Table B1.3.
- <shifter_C> is shorthand for the carry value output by the shifting circuit. See Table B1.3.

TABLE B1.3 Barrel shifter circuit outputs for different shift types.

Shift	*k* range	<shifted_Rm>	<shifter_C>
LSL k	k = 0	Rm	C (from *cpsr*)
LSL k	$1 \le k \le 31$	Rm « k	Rm[32-k]
LSL k	$k = 32$	0	Rm[0]
LSL k	$k \ge 33$	0	0
LSR k	$k = 0$	Rm	C
LSR k	$1 \le k \le 31$	(unsigned)Rm » k	Rm[k-1]
LSR k	$k = 32$	0	Rm[31]
LSR k	$k \ge 33$	0	0
ASR k	$k = 0$	Rm	C

Shift	*k* range	<shifted_Rm>	<shifter_C>
ASR k	$1 \le k \le 31$	(signed)Rm»k	Rm[k-1]
ASR k	$k \ge 32$	– Rm[31]	Rm[31]
ROR k	$k = 0$	Rm	C
ROR k	$1 \le k \le 31$	((unsigned)Rm » k)| (Rm » (32-k))	Rm[k-1]
ROR k	$k \ge 32$	Rm ROR (k & 31)	Rm[(k-1) & 31]
RRX		(C « 31) | ((unsigned)Rm » 1)	Rm[0]

B1.3 Alphabetical List of ARM and Thumb Instructions

Instructions are listed in alphabetical order. However, where signed and unsigned variants of the same operation exist, the main entry is under the signed variant.

`ADC` Add two 32-bit values and carry

```
1.  ADC<cond>{S}    Rd, Rn, #<rotated_immed>      ARMv1
2.  ADC<cond>{S}    Rd, Rn, Rm {, <shift>}        ARMv1
3.  ADC             Ld, Lm                        THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = Rn + <rotated_immed> + C	Updated if S suffix specified
2. Rd = Rn + <shifted_Rm> + C	Updated if S suffix specified
3. Ld = Ld + Lm + C	Updated (see Notes below)

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*UnsignedOverflow*>, *V* = <*SignedOverflow*>.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Examples

```
ADDS   r0, r0, r2    ; first half of a 64-bit add
ADC    r1, r1, r3    ; second half of a 64-bit add
ADCS   r0, r0, r0    ; shift r0 left, inserting carry (RLX)
```

`ADD` Add two 32-bit values

1.	`ADD<cond>S  Rd, Rn, #<rotated_immed>`	ARMv1
2.	`ADD<cond>S  Rd, Rn, Rm {, <shift>}`	ARMv1
3.	`ADD  Ld, Ln, #<immed3>`	THUMBv1
4.	`ADD  Ld, #<immed8>`	THUMBv1
5.	`ADD  Ld, Ln, Lm`	THUMBv1
6.	`ADD  Hd, Lm`	THUMBv1
7.	`ADD  Ld, Hm`	THUMBv1
8.	`ADD  Hd, Hm`	THUMBv1
9.	`ADD  Ld, pc, #<immed8>*4`	THUMBv1
10.	`ADD  Ld, sp, #<immed8>*4`	THUMBv1
11.	`ADD  sp, #<immed7>*4`	THUMBv1

Action	Effect on the *cpsr*
1. `Rd = Rn + <rotated_immed>`	Updated if S suffix specified
2. `Rd = Rn + <shifted_Rm>`	Updated if S suffix specified
3. `Ld = Ln + <immed3>`	Updated (see Notes below)
4. `Ld = Ld + <immed8>`	Updated (see Notes below)
5. `Ld = Ln + Lm`	Updated (see Notes below)
6. `Hd = Hd + Lm`	Preserved
7. `Ld = Ld + Hm`	Preserved
8. `Hd = Hd + Hm`	Preserved
9. `Ld = pc + 4*<immed8>`	Preserved
10. `Ld = sp + 4*<immed8>`	Preserved
11. `sp = sp + 4*<immed7>`	Preserved

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*UnsignedOverflow*>, *V* = <*SignedOverflow*>.
- If *Rd* or *Hd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.
- If *Hd* or *Hm* is *pc*, then the value used is the address of the instruction plus four bytes.

Examples

```
ADD  r0, r1, #4          ; r0 = r1 + 4
ADDS r0, r2, r2          ; r0 = r2 + r2 and flags updated
ADD  r0, r0, r0, LSL #1  ; r0 = 3*r0
ADD  pc, pc, r0, LSL #2  ; skip r0+1 instructions
ADD  r0, r1, r2, ROR r3  ; r0 = r1 + ((r2r»3)|(r2«(32-r3))
ADDS pc, lr, #4          ; jump to lr+4, restoring the cpsr
```

`ADR` Address relative

```
1. ADR{L}<cond> Rd, <address>     MACRO
```

This is not an ARM instruction, but an assembler macro that attempts to set *Rd* to the value `<address>` using a *pc*-relative calculation. The `ADR` instruction macro always uses a single ARM (or Thumb) instruction. The long-version `ADRL` always uses two ARM instructions and so can access a wider range of addresses. If the assembler cannot generate an instruction sequence reaching the address, then it will generate an error.

The following example shows how to call the function pointed to by *r9*. We use `ADR` to set *lr* to the return address; in this case, it will assemble to `ADD lr, pc, #4`. Recall that *pc* reads as the address of the current instruction plus eight in this case.

```
      ADR    lr, return_address    ; set return address
      MOV    r0, #0                ; set a function argument
      BX     r9                    ; call the function
return_address                     ; resume
```

`AND` Logical bitwise AND of two 32-bit values

```
1. AND<cond>{S}   Rd, Rn, #<rotated_immed>   ARMv1
2. AND<cond>{S}   Rd, Rn, Rm {, <shift>}     ARMv1
3. AND            Ld, Lm                     THUMBv1
```

Action	Effect on the *cpsr*
`1. Rd = Rn & <rotated_immed>`	`Updated if S suffix specified`
`2. Rd = Rn & <shifted_Rm>`	`Updated if S suffix specified`
`3. Ld = Ld & Lm`	`Updated (see Notes below)`

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3), *V* is preserved.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.

- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Example

```
AND     r0, r0, #0xFF      ; extract the lower 8 bits of a byte
ANDS    r0, r0, #1≪31      ; extract sign bit
```

ASR Arithmetic shift right for Thumb (see MOV for the ARM equivalent)

```
1. ASR Ld, Lm, #<immed5>     THUMBv1
2. ASR Ld, Ls                THUMBv1
```

Action	Effect on the cpsr
1. Ld = Lm ASR #<immed5>	Updated (see Notes below)
2. Ld = Ld ASR Ls[7:0]	Updated

Note

- The *cpsr* is updated: *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3).

B Branch relative

```
1. B<cond>              <address25>      ARMv1
2. B<cond>              <address8>       THUMBv1
3. B                    <address11>      THUMBv1
```

Branches to the given address or label. The address is stored as a relative offset.

Examples

```
B   label  ; branch unconditionally to a label
BGT loop   ; conditionally continue a loop
```

BIC Logical bit clear (AND NOT) of two 32-bit values

```
1. BIC<cond>{S} Rd, Rn, #<rotated_immed>        ARMv1
2. BIC<cond>{S} Rd, Rn, Rm {, <shift>}          ARMv1
3. BIC              Ld, Lm                      THUMBv1
```

Action	Effect on the cpsr
1. Rd = Rn & ~<rotated_immed>	Updated if S suffix specified
2. Rd = Rn & ~<shifted_Rm>	Updated if S suffix specified
3. Ld = Ld & ~Lm	Updated (see Notes below)

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3), *V* is preserved.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Examples

```
BIC          r0, r0, #1«22     ; clear bit 22 of r0
```

`BKPT` Breakpoint instruction

```
1. BKPT <immed16>                                   ARMv5
2. BKPT <immed8>                                    THUMBv2
```

The breakpoint instruction causes a prefetch data abort, unless overridden by debug hardware. The ARM ignores the immediate value. This immediate can be used to hold debug information such as the breakpoint number.

`BL` Relative branch with link (subroutine call)

```
1. BL<cond>   <address25>                           ARMv1
2. BL         <address22>                           THUMBv1
```

Action	Effect on the *cpsr*
1. lr = ret+0; pc = <address25>	None
2. lr = ret+1; pc = <address22>	None

Note

- These instructions set *lr* to the address of the following instruction `ret` plus the current *cpsr T*-bit setting. Therefore you can return from the subroutine using BX `lr` to resume execution address and ARM or Thumb state.

Examples

```
BL   subroutine ; call subroutine (return with MOV pc,lr)
BLVS overflow   ; call subroutine on an overflow
```

`BLX` Branch with link and exchange (subroutine call with possible state switch)

```
1. BLX          <address25>                         ARMv5
```

2. BLX<cond>	Rm	ARMv5
3. BLX	<address22>	THUMBv2
4. BLX	Rm	THUMBv2

Action	Effect on the *cpsr*
1. lr = ret+0; pc = <address25>	T=1 (switch to Thumb state)
2. lr = ret+0; pc = Rm & 0xfffffffe	T=Rm & 1
3. lr = ret+1; pc = <address22>	T=0 (switch to ARM state)
4. lr = ret+1; pc = Rm & 0xfffffffe	T=Rm & 1

Notes

- These instructions set *lr* to the address of the following instruction ret plus the current *cpsr T*-bit setting. Therefore you can return from the subroutine using `BX lr` to resume execution address and ARM or Thumb state.
- *Rm* must not be *pc*.
- *Rm* & 3 must not be 2. This would cause a branch to an unaligned ARM instruction.

Example

```
BLX  thumb_code  ; call a Thumb subroutine from ARM state
BLX  r0          ; call the subroutine pointed to by r0
                 ; ARM code if r0 even, Thumb if r0 odd
```

`BX` `BXJ` Branch with exchange (branch with possible state switch)

1. BX<cond>	Rm	ARMv4T
2. BX	Rm	THUMBv1
3. BXJ<cond>	Rm	ARMv5J

Action	Effect on the *cpsr*
1. pc = Rm & 0xfffffffe	T=Rm & 1
2. pc = Rm & 0xfffffffe	T=Rm & 1
3. Depends on JE configuration bit	J,T affected

Notes

- If *Rm* is *pc* and the instruction is word aligned, then *Rm* takes the value of the current instruction plus eight in ARM state or plus four in Thumb state.
- *Rm* & 3 must not be 2. This would cause a branch to an unaligned ARM instruction.

- If the JE (Java Enable) configuration bit is clear, then BXJ behaves as a BX. Otherwise, the behavior is defined by the architecture of the Java Extension hardware. Typically it sets *J = 1* in the *cpsr* and starts executing Java instructions from a general purpose register designated as the Java program counter *jpc*.

Examples

```
BX    lr    ; return from ARM or Thumb subroutine
BX    r0    ; branch to ARM or Thumb function pointer r0
```

`CDP` Coprocessor data processing operation

```
1. CDP<cond> <copro>, <op1>, Cd, Cn, Cm, <op2>          ARMv2
2. CDP2      <copro>, <op1>, Cd, Cn, Cm, <op2>          ARMv5
```

These instructions initiate a coprocessor-dependent operation. *<copro>* is the number of the coprocessor in the range *p0* to *p15*. The core takes an undefined instruction trap if the coprocessor is not present. The coprocessor operation specifiers *<op1>* and *<op2>*, and the coprocessor register numbers *Cd*, *Cn*, *Cm*, are interpreted by the coprocessor and ignored by the ARM. `CDP2` provides an additional set of coprocessor instructions.

`CLZ` Count leading zeros

```
1. CLZ<cond> Rd, Rm                                     ARMv5
```

Rn is set to the maximum left shift that can be applied to *Rm* without unsigned overflow. Equivalently, this is the number of zeros above the highest one in the binary representation of *Rm*. If *Rm* = 0, then *Rn* is set to 32. The following example normalizes the value in *r0* so that bit 31 is set

```
CLZ r1, r0            ; find normalization shift
MOV r0, r0, LSL r1    ; normalize so bit 31 is set (if r0!=0)
```

`CMN` Compare negative

```
1. CMN<cond>    Rn, #<rotated_immed>                    ARMv1
2. CMN<cond>    Rn, Rm {, <shift>}                      ARMv1
3. CMN          Ln, Lm                                  THUMBv1
```

Action

```
1. cpsr flags set on the result of (Rn + <rotated_immed>)
2. cpsr flags set on the result of (Rn + <shifted_Rm>)
3. cpsr flags set on the result of (Ln + Lm)
```

Notes

- In the *cpsr: N = <Negative>*, *Z = <Zero>*, *C = <Unsigned-Overflow>*, *V = <SignedOverflow>*. These are the same flags as generated by `CMP` with the second operand negated.

- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Example

```
CMN     r0, #3       ; compare r0 with -3
BLT     label        ; if (r0 <— 3) goto label
```

CMP Compare two 32-bit integers

```
1. CMP<cond>     Rn, #<rotated_immed>     ARMv1
2. CMP<cond>     Rn, Rm {, <shift>}       ARMv1
3. CMP           Ln, #<immed8>            THUMBv1
4. CMP           Rn, Rm                   THUMBv1
```

Action

```
1. cpsr    flags  set on the result of     (Rn - <rotated_immed>)
2. cpsr    flags  set on the result of     (Rn - <shifted_Rm>)
3. cpsr    flags  set on the result of     (Ln - <immed8>)
4. cpsr    flags  set on the result of     (Rn - Rm)
```

Notes

- In the *cpsr: N = <Negative>*, *Z = <Zero>*, *C = <NoUnsigned-Overflow>*, *V = <SignedOverflow>*. The carry flag is set this way because the subtract $x-y$ is implemented as the add $x + \sim y + 1$. The carry flag is one if $x + \sim y + 1$ overflows. This happens when $x \geq y$ (equivalently when $x - Ây$ doesn't overflow).
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes for ARM instructions, or plus four bytes for Thumb instructions.

Example

```
CMP    r0, r1, LSR#2  ; compare r0 with (r1/4)
BHS    label          ; if (r0 >= (r1/4)) goto label;
```

CPS Change processor state; modifies selected bits in the *cpsr*

```
1. CPS             #<mode>          ARMv6
2. CPSID <flags> {, #<mode>}        ARMv6
3. CPSIE <flags> {, #<mode>}        ARMv6
4. CPSID <flags>                    THUMBv3
5. CPSIE <flags>                    THUMBv3
```

Action

```
1. cpsr[4:0] = <mode>
2. cpsr = cpsr | mask; { cpsr[4:0]=<mode> }
```

```
3. cpsr = cpsr & ~mask; { cpsr[4:0]=<mode> }
4. cpsr = cpsr | mask
5. cpsr = cpsr & ~mask
```

Bits are set in `mask` according to letters in the `<flags>` value as in Table B1.4. The `ID` (interrupt disable) variants mask interrupts by setting `cpsr` bits. The `IE` (interrupt enable) variants unmask interrupts by clearing *cpsr* bits.

TABLE B1.4 CPS flags characters.

Character	*cpsr* bit affected	Bit set in mask
a	imprecise data Abort mask bit	0 × 100 = 1 << 8
i	IRQ mask bit	0 × 080 = 1 << 7
f	FIQ mask bit	0 × 040 = 1 << 6

`CPY` Copy one ARM register to another without affecting the *cpsr*.

```
1. CPY<cond>    Rd, Rm                  ARMv6
2. CPY          Rd, Rm                  THUMBv3
```

This assembles to MOV *<cond> Rd, Rm* except in the case of Thumb where *Rd* and *Rm* are low registers in the range *r0* to *r7*. Then it is a new operation that sets *Rd=Rm* without affecting the *cpsr*.

`EOR` Logical exclusive OR of two 32-bit values

```
1. EOR<cond>{S} Rd, Rn, #<rotated_immed>        ARMv1
2. EOR<cond>{S} Rd, Rn, Rm {, <shift>}          ARMv1
3. EOR          Ld, Lm                          THUMBv1
```

Action	Effect on the *cpsr*
`1. Rd = Rn ^<rotated_immed>`	`Updated if S suffix specified`
`2. Rd = Rn ^ <shifted_Rm>`	`Updated if S suffix specified`
`3. Ld = Ld ^ Lm`	`Updated (see Notes below)`

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = *<Negative>*, *Z* = *<Zero>*, *C* = *<shifter_C>* (see Table B1.3), *V* is preserved.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.

- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

  Example

```
EOR    r0, r0, #1≪16    ; toggle bit 16
```

LDC Load to coprocessor single or multiple 32-bit values

```
1. LDC<cond>{L}     <copro>, Cd, [Rn {, #{-}<immed8>*4}]{!}    ARMv2
2. LDC<cond>{L}     <copro>, Cd, [Rn], #{-}<immed8>*4          ARMv2
3. LDC<cond>{L}     <copro>, Cd, [Rn], <option>                ARMv2
4. LDC2{L}          <copro>, Cd, [Rn {, #{-}<immed8>*4}]{!}    ARMv5
5. LDC2{L}          <copro>, Cd, [Rn], #{-}<immed8>*4          ARMv5
6. LDC2{L}          <copro>, Cd, [Rn], <option>                ARMv5
```

These instructions initiate a memory read, transferring data to the given coprocessor. *<copro>* is the number of the coprocessor in the range *p0* to *p15*. The core takes an undefined instruction trap if the coprocessor is not present. The memory read consists of a sequence of words from sequentially increasing addresses. The initial address is specified by the addressing mode in Table B1.5. The coprocessor controls the number of words transferred, up to a maximum limit of 16 words. The fields *{L}* and *Cd* are interpreted by the coprocessor and ignored by the ARM. Typically *Cd* specifies the destination coprocessor register for the transfer. The `<option>` field is an eight-bit integer enclosed in { }. Its interpretation is coprocessor dependent.

TABLE B1.5 LDC addressing modes.

Addressing format	Address accessed	Value written back to *Rn*
[Rn {,#{-}<immed>}]	Rn + {{-}<immed>}	Rn preserved
[Rn {,#{-}<immed>}]!	Rn + {{-}<immed>}	Rn + {{-}<immed>}
[Rn], #{-}<immed>	Rn	Rn + {-}<immed>
[Rn], < option>	Rn	Rn preserved

If the address is not a multiple of four, then the access is *unaligned*. The restrictions on unaligned accesses are the same as for `LDM`.

LDM Load multiple 32-bit words from memory to ARM registers

```
1. LDM<cond><amode>  Rn{!}, <register_list>{^}    ARMv1
2. LDMIA             Rn!,   <register_list>       THUMBv1
```

These instructions load multiple words from sequential memory addresses. The `<register_list>` specifies a list of registers to load, enclosed in curly brackets

{}. Although the assembler allows you to specify the registers in the list in any order, the order is not stored in the instruction, so it is good practice to write the list in increasing order of register number because this is the usual order of the memory transfer.

The following pseudocode shows the normal action of LDM. We use <register_list>[i] to denote the register appearing at position *i* in the list, starting at 0 for the first register. This assumes that the list is in order of increasing register number.

```
N = the number of registers in <register_list>
start = the lowest address accessed given in Table B1.6
for (i=0; i<N; i++)
       <register_list>[i] = memory(start+i*4, 4);
if (! specified) then update Rn according to Table B1.6
```

Note that memory(a, 4) returns the four bytes at address a packed according to the current processor data endianness. If a is not a multiple of four, then the load is unaligned. Because the behavior of an unaligned load depends on the architecture revision, memory system, and system coprocessor (CP15) configuration, it's best to avoid unaligned loads if possible. Assuming that the external memory system does not abort unaligned loads, then the following rules usually apply:

- If the core has a system coprocessor and bit 1 (A-bit) or bit 22 (U-bit) of CP15:c1:c0:0 is set, then unaligned load multiples cause an alignment fault data abort exception.
- Otherwise the access ignores the bottom two address bits.

Table B1.6 lists the possible addressing modes specified by <amode>. If you specify the !, then the base address register is updated according to Table B1.6; otherwise it is preserved. Note that the lowest register number is always read from the lowest address.

The first half of the addressing mode mnemonics stands for Increment After, Increment Before, Decrement After, and Decrement Before, respectively. Increment modes load the registers sequentially forward, starting from address *Rn* (increment after) or *Rn* + 4 (increment before). Decrement modes have the same effect as if you loaded the register list backwards from sequentially

TABLE B1.6 LDM addressing modes.

Addressing mode	Lowest address accessed	Highest address accessed	Value written back to *Rn* if ! specified
{IA|FD}	Rn	Rn + N*4 - 4	Rn + N*4
{IB|ED}	Rn + 4	Rn + N*4	Rn + N*4
{DA|FA}	Rn − N*4 + 4	Rn	Rn − N*4
{DB|EA}	Rn − N*4	Rn - 4	Rn − N*4

descending memory addresses, starting from address *Rn* (decrement after) or *Rn* – 4 (decrement before).

The second half of the addressing mode mnemonics stands for the stack type you can implement with that address mode: Full Descending, Empty Descending, Full Ascending, and Empty Ascending, With a full stack, *Rn* points to the last stacked value; with an empty stack, *Rn* points to the first unused stack location. ARM stacks are usually full descending. You should use full descending or empty ascending stacks by preference, since `LDC` also supports these addressing modes.

Notes

- For Thumb (format 2), *Rn* and the register list registers must be in the range *r0* to *r7*.
- The number of registers *N* in the list must be nonzero.
- *Rn* must not be *pc*.
- *Rn* must not appear in the register list if ! (writeback) is specified.
- If *pc* appears in the register list, then on ARMv5 and above the processor performs a BX to the loaded address. For ARMv4 and below, the processor branches to the loaded address.
- If ^ is specified, then the operation is modified. The processor must not be in *user* or *system* mode. If *pc* is not in the register list, then the registers appearing in the register list refer to the *user* mode versions of the registers and writeback must not be specified. If *pc* is in the register list, then the *spsr* is copied to the *cpsr* in addition to the standard operation.
- The time order of the memory accesses may depend on the implementation. Be careful when using a load multiple to access I/O locations where the access order matters. If the order matters, then check that the memory locations are marked as I/O in the page tables, do not cross page boundaries, and do not use *pc* in the register list.

Examples

```
LDMIA   r4!, {r0, r1}  ; r0=*r4, r1=*(r4+4), r4+=8
LDMDB   r4!, {r0, r1}  ; r1=*(r4-4), r0=*(r4-8), r4-=8
LDMEQFD sp!, {r0, pc}  ; if (result zero) then unstack r0, pc
LDMFD   sp, {sp}^      ; load sp_usr from sp_current
LDMFD   sp!, {r0-pc}^  ; return from exception, restore cpsr
```

LDR Load a single value from a virtual address in memory

```
1. LDR<cond>{|B}       Rd, [Rn {, #{-}<immed12>}]{!}    ARMv1
2. LDR<cond>{|B}       Rd, [Rn, {-}Rm {,<imm_shift>}]{!} ARMv1
```

```
 3. LDR<cond>{|B}{T}     Rd, [Rn], #{-}<immed12>          ARMv1
 4. LDR<cond>{|B}{T}     Rd, [Rn], {-}Rm {,<imm_shift>}   ARMv1
 5. LDR<cond>{H|SB|SH}   Rd, [Rn, {, #{-}<immed8>}]{!}    ARMv4
 6. LDR<cond>{H|SB|SH}   Rd, [Rn, {-}Rm]{!}               ARMv4
 7. LDR<cond>{H|SB|SH}   Rd, [Rn], #{-}<immed8>           ARMv4
 8. LDR<cond>{H|SB|SH}   Rd, [Rn], {-}Rm                  ARMv4
 9. LDR<cond>D           Rd, [Rn, {, #{-}<immed8>}]{!}    ARMv5E
10. LDR<cond>D           Rd, [Rn, {-}Rm]{!}               ARMv5E
11. LDR<cond>D           Rd, [Rn], #{-}<immed8>           ARMv5E
12. LDR<cond>D           Rd, [Rn], {-}Rm                  ARMv5E
13. LDREX<cond>          Rd, [Rn]                         ARMv6
14. LDR{|B|H}            Ld, [Ln, #<immed5>*<size>]       THUMBv1
15. LDR{|B|H|SB|SH}      Ld, [Ln, Lm]                     THUMBv1
16. LDR                  Ld, [pc, #<immed8>*4]            THUMBv1
17. LDR                  Ld, [sp, #<immed8>*4]            THUMBv1
18. LDR<cond><type>      Rd, <label>                      MACRO
19. LDR<cond>            Rd, =<32-bit-value>              MACRO
```

Formats 1 to 17 load a single data item of the type specified by the opcode suffix, using a preindexed or postindexed addressing mode. Tables B1.7 and B1.8 show the different addressing modes and data types.

TABLES B1.7 LDR Addressing Modes.

Addressing format	Address a accessed	Value written back to *Rn*
[Rn {,#{-}<immed>}]	Rn + {{-}<immed>}	Rn preserved
[Rn {,#{-}<immed>}]!	Rn + {{-}<immed>}	Rn + {{-}<immed>}
[Rn, {-}Rm {,<shift>}]	Rn + {-}<shifted_Rm>	Rn preserved
[Rn, {-}Rm {,<shift>}]!	Rn + {-}<shifted_Rm>	Rn + {-}<shifted_Rm>
[Rn], #{-}<immed>	Rn	Rn + {-}<immed>
[Rn], {-}Rm {,<shift>}	Rn	Rn + {-}<shifted_Rm>

In Table B1.8 `memory(a, n)` reads `n` sequential bytes from address `a`. The bytes are packed according to the configured processor data endianness. The function `memoryT(a, n)` performs the same access but with *user* mode privileges, regardless of the current processor mode. The function `memoryEx(a, n)` used by `LDREX` performs the access and marks the access as exclusive. If address a has the *shared* TLB attribute, then this marks address a as exclusive to the current processor and clears any other exclusive addresses for this processor. Otherwise the processor remembers that there is an outstanding exclusive access. Exclusivity only affects the action of the STREX instruction.

TABLES B1.8 LDR datatypes.

Load	Datatype	<size> (bytes)	Action
LDR	word	4	Rd = memory(a, 4)
LDRB	unsigned Byte	1	Rd = (zero-extend)memory(a, 1)
LDRBT	Byte Translated	1	Rd = (zero-extend)memoryT(a, 1)
LDRD	Double word	8	Rd = memory(a, 4) R(d+1) = memory(a+4, 4)
LDREX	word EXclusive	4	Rd = memoryEx(a, 4)
LDRH	unsigned Halfword	2	Rd = (zero-extend)memory(a, 2)
LDRSB	Signed Byte	1	Rd = (sign-extend)memory(a, 1)
LDRSH	Signed Halfword	2	Rd = (sign-extend)memory(a, 2)
LDRT	word Translated	4	Rd = memoryT(a, 4)

If address a is not a multiple of `<size>`, then the load is *unaligned*. Because the behavior of an unaligned load depends on the architecture revision, memory system, and system coprocessor (CP15) configuration, it's best to avoid unaligned loads if possible. Assuming that the external memory system does not abort unaligned loads, then the following rules usually apply. In the rules, *A* is bit 1 of system coprocessor register CP15:c1:c0:0, and *U* is bit 22 of CP15:c1:c0:0, introduced in ARMv6. If there is no system coprocessor, then $A = U = 0$.

- If A = 1, then unaligned loads cause an alignment fault data abort exception except that word-aligned double-word loads are supported if $U = 1$.
- If $A = 0$ and $U = 1$, then unaligned loads are supported for `LDR{|T|H|SH}`. Word-aligned loads are supported for `LDRD`. A non-word-aligned `LDRD` generates an alignment fault data abort.
- If $A = 0$ and $U = 0$, then `LDR` and `LDRT` return the value `memory(a & ~ 3, 4) ROR ((a&3)*8)`. All other unaligned operations are unpredictable but do not generate an alignment fault.

Format 18 generates a *pc*-relative load accessing the address specified by <label>. In other words, it assembles to `LDR<cond><type> Rd, [pc, #<offset>]` whenever this instruction is supported and `<offset>=<label>-pc` is in range.

Format 19 generates an instruction to move the given 32-bit value to the register *Rd*. Usually the instruction is `LDR<cond> Rd, [pc, #<offset>]`, where the 32-bit value is stored in a literal pool at address `pc+<offset>`.

Notes

- For double-word loads (formats 9 to 12), *Rd* must be even and in the range *r0* to *r12*.
- If the addressing mode updates *Rn*, then *Rd* and *Rn* must be distinct.

- If *Rd* is *pc*, then <*size*> must be 4. Up to ARMv4, the core branches to the loaded address. For ARMv5 and above, the core performs a BX to the loaded address.
- If *Rn* is *pc*, then the addressing mode must not update *Rn* . The value used for *Rn* is the address of the instruction plus eight bytes for ARM or four bytes for Thumb.
- *Rm* must not be *pc*.
- For ARMv6 use LDREX and STREX to implement semaphores rather than SWP.

Examples

```
LDR     r0, [r0]              ; r0 = *(int*)r0;
LDRSH   r0, [r1], #4          ; r0 = *(short*)r1; r1 += 4;
LDRB    r0, [r1, #-8]!        ; r1 -= 8; r0 = *(char*)r1;
LDRD    r2, [r1]              ; r2 =* (int*)r1;
                                r3 =* (int*)(r1+4);
LDRSB   r0, [r2, #55]         ; r0 = *(signed char*)
                                (r2+55);
LDRCC   pc, [pc, r0, LSL #2]  ; if (C==0) goto *(pc+4*r0);
LDRB    r0, [r1], -r2, LSL #8 ; r0 = *(char*)r1;
                                r1 -= 256*r2;
LDR     r0, =0x12345678       ; r0 = 0x12345678;
```

LSL Logical shift left for Thumb (see MOV for the ARM equivalent)

```
1. LSL Ld, Lm, #<immed5>                          THUMBv1
2. LSL Ld, Ls                                     THUMBv1
```

Action	Effect on the *cpsr*
1. Ld = Lm LSL #<immed5>	Updated (see Note below)
2. Ld = Ld LSL Ls[7:0]	Updated

Note

- The *cpsr* is updated: *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3).

LSR Logical shift right for Thumb (see MOV for the ARM equivalent)

```
1. LSR Ld, Lm, #<immed5>                          THUMBv1
2. LSR Ld, Ls                                     THUMBv1
```

Action	Effect on the *cpsr*
1. Ld = Lm LSR #<immed5>	Updated (see Note below)
2. Ld = Ld LSR Ls[7:0]	Updated

Note

- The *cpsr* is updated: *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3).

`MCR` Move to coprocessor from an ARM register
`MCRR`

```
1. MCR<cond>   <copro>, <op1>, Rd, Cn, Cm {, <op2>}  ARMv2
2. MCR2        <copro>, <op1>, Rd, Cn, Cm {, <op2>}  ARMv5
3. MCRR<cond>  <copro>, <op1>, Rd, Rn, Cm            ARMv5E
4. MCRR2       <copro>, <op1>, Rd, Rn, Cm            ARMv6
```

These instructions transfer the value of ARM register *Rd* to the indicated coprocessor. Formats 3 and 4 also transfer a second register *Rn*. `<copro>` is the number of the coprocessor in the range *p0* to *p15*. The core takes an undefined instruction trap if the coprocessor is not present. The coprocessor operation specifiers `<op1>` and `<op2>`, and the coprocessor register numbers *Cn*, *Cm*, are interpreted by the coprocessor, and ignored by the ARM. *Rd* and *Rn* must not be *pc*. Coprocessor *p15* controls memory management options. For example, the following code sequence enables alignment fault checking:

```
MRC    p15, 0, r0, c1, c0, 0    ; read the MMU register, c1
ORR    r0, r0, #2               ; set the A bit
MCR    p15, 0, r0, c1, c0, 0    ; write the MMU register, c1
```

`MLA` Multiply with accumulate

```
1. MLA<cond>{S} Rd, Rm, Rs, Rn          ARMv2
```

Action	Effect on the *cpsr*
1. Rd = Rn + Rm*Rs	Updated if S suffix supplied

Notes

- *Rd* is set to the lower 32 bits of the result.
- *Rd*, *Rm*, *Rs*, *Rn* must not be *pc*.
- *Rd* and *Rm* must be different registers.
- Implementations may terminate early on the value of the *Rs* operand. For this reason use small or constant values for *Rs* where possible. See Appendix B3.
- If the *cpsr* is updated, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* is unpredictable, and *V* is preserved. Avoid using the instruction `MLAS` because implementations often

impose penalty cycles for this operation. Instead use MLA followed by a compare, and schedule the compare to avoid multiply result use interlocks.

MOV Move a 32-bit value into a register

```
1. MOV<cond>{S}  Rd, #<rotated_immed>   ARMv1
2. MOV<cond>{S}  Rd, Rm {, <shift>}     ARMv1
3. MOV           Ld, #<immed8>          THUMBv1
4. MOV           Ld, Ln                 THUMBv1
5. MOV           Hd, Lm                 THUMBv1
6. MOV           Ld, Hm                 THUMBv1
7. MOV           Hd, Hm                 THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = <rotated_immed>	Updated if S suffix specified
2. Rd = <shifted_Rm>	Updated if S suffix specified
3. Ld = <immed8>	Updated (see Notes below)
4. Ld = Ln	Updated (see Notes below)
5. Hd = Lm	Preserved
6. Ld = Hm	Preserved
7. Hd = Hm	Preserved

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3), and *V* is preserved.
- If *Rd* or *Hd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.
- If *Hm* is *pc*, then the value used is the address of the instruction plus four bytes.

Examples

```
MOV   r0, #0x00ff0000 ; r0 = 0x00ff0000
MOV   r0, r1, LSL#2   ; r0 = 4*r1
MOV   pc, lr      ; return from subroutine (pc=lr)
MOVS  pc, lr      ; return from exception (pc=lr, cpsr=spsr)
```

MRC
MRRC Move to ARM register from a coprocessor

```
1. MRC<cond>        <copro>, <op1>, Rd, Cn, Cm , <op2>     ARMv2
2. MRC2             <copro>, <op1>, Rd, Cn, Cm , <op2>     ARMv5
3. MRRC<cond>       <copro>, <op1>, Rd, Rn, Cm             ARMv5E
4. MRRC2            <copro>, <op1>, Rd, Rn, Cm             ARMv6
```

These instructions transfer a 32-bit value from the indicated coprocessor to the ARM register *Rd*. Formats 3 and 4 also transfer a second 32-bit value to *Rn*. <copro> is the number of the coprocessor in the range *p0* to *p15*. The core takes an undefined instruction trap if the coprocessor is not present. The coprocessor operation specifiers <op1> and <op2>, and the coprocessor register numbers *Cn*, *Cm*, are interpreted by the coprocessor and ignored by the ARM. For formats 1 and 2, if *Rd* is *pc*, then the top four bits of the *cpsr* (the *NZCV* condition code flags) are set from the top four bits of the 32-bit value transferred; *pc* is not affected. For other formats, *Rd* and *Rn* must be distinct and not *pc*.

Coprocessor *p15* controls memory management options. For example, the following instruction reads the main ID register from *p15*:

```
MRC   p15, 0, r0, c0, c0   ; read the MMU ID register, c0
```

MRS Move to ARM register from status register (*cpsr* or *spsr*)

```
1. MRS<cond> Rd, cpsr      ARMv3
2. MRS<cond> Rd, spsr      ARMv3
```

These instructions set *Rd* = *cpsr* and *Rd* = *spsr*, respectively. *Rd* must not be *pc*.

MSR Move to status register (*cpsr* or *spsr*) from an ARM register

```
1. MSR<cond> cpsr_<fields>, #<rotated_immed>  ARMv3
2. MSR<cond> cpsr_<fields>, Rm                ARMv3
3. MSR<cond> spsr_<fields>, #<rotated_immed>  ARMv3
4. MSR<cond> spsr_<fields>, Rm                ARMv3
```

Action

```
1. cpsr = (cpsr & ~<mask>) | (<rotated_immed> & <mask>)
2. cpsr = (cpsr & ~<mask>) | (Rm & <mask>)
3. spsr = (spsr & ~<mask>) | (<rotated_immed> & <mask>)
4. spsr = (spsr & ~<mask>) | (Rm & <mask>)
```

These instructions alter selected bytes of the *cpsr* or *spsr* according to the value of <mask>. The <fields> specifier is a sequence of one or more letters, determining which bytes of <mask> are set. See Table B1.9.

TABLE B1.9 Format of the <fields> specifier.

<fields> letter	Meaning	Bits set in <mask>
c	Control byte	0x000000ff
x	eXtension byte	0x0000ff00
s	Status byte	0x00ff0000
f	Flags byte	0xff000000

Some old ARM toolkits allowed *cpsr* or *cpsr_all* in place of *cpsr_fsxc*. They also used *cpsr_flg* and *cpsr_ctl* in place of *cpsr_f* and *cpsr_c*, respectively. These formats, and the *spsr* equivalents, are obsolete, so you should not use them. The following example changes to *system* mode and enables IRQ, which is useful in a reentrant interrupt handler:

```
MRS     r0, cpsr        ; read cpsr state
BIC     r0, r0, #0x9f   ; clear IRQ disable and mode bits
ORR     r0, r0, #0x1f   ; set system mode
MSR     cpsr_c, r0      ; update control byte of the cpsr
```

`MUL` Multiply

```
1. MUL<cond>{S}    Rd, Rm, Rs       ARMv2
2. MUL             Ld, Lm           THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = Rm*Rs	Updated if S suffix supplied
2. Ld = Lm*Ld	Updated

Notes

- *Rd* or *Ld* is set to the lower 32 bits of the result.
- *Rd*, *Rm*, *Rs* must not be *pc*.
- *Rd* and *Rm* must be different registers. Similarly *Ld* and *Lm* must be different.
- Implementations may terminate early on the value of the *Rs* or *Ld* operand. For this reason use small or constant values for *Rs* or *Ld* where possible.
- If the *cpsr* is updated, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* is unpredictable, and *V* is preserved. Avoid using the instruction `MULS` because implementations often impose penalty cycles for this operation. Instead use `MUL` followed by a compare, and schedule the compare, to avoid multiply result use interlocks.

`MVN` Move the logical not of a 32-bit value into a register

```
1. MVN<cond>{S}   Rd, #<rotated_immed>      ARMv1
2. MVN<cond>{S}   Rd, Rm {, <shift>}        ARMv1
3. MVN            Ld, Lm                    THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = ~<rotated_immed>	Updated if S suffix specified
2. Rd = ~<shifted_Rm>	Updated if S suffix specified
3. Ld = ~ Lm	Updated (see Notes below)

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3), and *V* is preserved.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Examples

```
MVN  r0, #0xff      ; r0 = 0xffffff00
MVN  r0, #0         ; r0 = -1
```

NEG Negate value in Thumb (use `RSB` to negate in ARM state)

```
1. NEG Ld, Lm       THUMBv1
```

Action	Effect on the *cpsr*
`1. Ld = -Lm`	`Updated (see Notes below)`

Notes

- The *cpsr* is updated: *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*NoUnsignedOverflow*>, *V* = <*SignedOverflow*>. Note that *Z* = *C* and *V* = (*Ld*== 0x80000000).
- This is the same as the operation `RSBS Ld, Lm, #0` in ARM state.

`NOP` No operation

```
1. NOP      MACRO
```

This is not an ARM instruction. It is an assembly macro that produces an instruction having no effect other than advancing the *pc* as normal. In ARM state it assembles to `MOV r0, r0`. In Thumb state it assembles to `MOV r8, r8`. The operation is not guaranteed to take one processor cycle. In particular, if you use `NOP` after a load of *r0*, then the operation may cause pipeline interlocks.

`ORR` Logical bitwise OR of two 32-bit values

```
1. ORR<cond>{S}      Rd, Rn, #<rotated_immed>   ARMv1
2. ORR<cond>{S}      Rd, Rn, Rm {, <shift>}     ARMv1
3. ORR               Ld, Lm                     THUMBv1
```

Action	Effect on the *cpsr*
`1. Rd = Rn | <rotated_immed>`	`Updated if S suffix specified`
`2. Rd = Rn | <shifted_Rm>`	`Updated if S suffix specified`
`3. Ld = Ld | Lm`	`Updated (see Notes below)`

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3), and *V* is preserved.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*, in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Example

```
ORR     r0, r0,#1 <<1 ; set bit 13 of r0
```

PKH Pack 16-bit halfwords into a 32-bit word

```
1. PKHBT<cond> Rd, Rn, Rm {, LSL #<0-31>} ARMv6
2. PKHTB<cond> Rd, Rn, Rm {, ASR #<1-32>} ARMv6
```

Action

```
1. Rd[15:00] = Rn[15:00]; Rd[31:16]=<shifted_Rm>[31:16]
2. Rd[31:16] = Rn[31:16]; Rd[15:00]=<shifted_Rm>[15:00]
```

Note

- *Rd*, *Rn*, *Rm* must not be *pc*. *cpsr* is not affected.

Examples

```
PKHBT  r0, r1, r2, LSL#16  ; r0 = (r2[15:00]<<16)|r1[15:00]
PKHTB  r0, r2, r1, ASR#16  ; r0 = (r2[31:15]<<16)|r1[31:15]
```

PLD Preload hint instruction

```
1. PLD [Rn {, #{-}<immed12>}]              ARMv5E
2. PLD [Rn, {-}Rm {,<imm_shift>}]          ARMv5E
```

Action

```
1. Preloads from address (Rn + {{-}<immed12>})
2. Preloads from address (Rn + {-}<shifted_Rm>)
```

This instruction does not affect the processor registers (other than advancing *pc*). It merely hints that the programmer is likely to read from the given address in future. A cached processor may take this as a hint to load the cache line containing the address into the cache. The instruction should not generate a data abort or any other memory

system error. If *Rn* is *pc*, then the value used for *Rn* is the address of the instruction plus eight. *Rm* must not be *pc*.

Examples

```
PLD     [r0, #7]          ; Preload from r0+7
PLD     [r0, r1, LSL#2]   ; Preload from r0+4*r1
```

`POP` Pops multiple registers from the stack in Thumb state (for ARM state use `LDM`)

```
1. POP <regster_list>                          THUMBv1
```

Action

```
1. equivalent to the ARM instruction LDMFD sp!, <register_list>
```

The `<register_list>` can contain registers in the range *r0* to *r7* and *pc*. The following example restores the low-numbered ARM registers and returns from a subroutine:

```
POP {r0-r7,pc}
```

`PUSH` Pushes multiple registers to the stack in Thumb state (for ARM state use `STM`)

```
1. PUSH <regster_list>                         THUMBv1
```

Action

```
1. equivalent to the ARM instruction STMFD sp!, <register_list>
```

The `<register_list>` can contain registers in the range *r0* to *r7* and *lr*. The following example saves the low-numbered ARM registers and link register.

```
PUSH {r0-r7,lr}
```

`QADD` `QDADD` `QDSUB` `QSUB` Saturated signed and unsigned arithmetic

```
1. QADD<cond>           Rd, Rm, Rn        ARMv5E
2. QDADD<cond>          Rd, Rm, Rn        ARMv5E
3. QSUB<cond>           Rd, Rm, Rn        ARMv5E
4. QDSUB<cond>          Rd, Rm, Rn        ARMv5E
5. {U}QADD16<cond>      Rd, Rn, Rm        ARMv6
6. {U}QADDSUBX<cond>    Rd, Rn, Rm        ARMv6
7. {U}QSUBADDX<cond>    Rd, Rn, Rm        ARMv6
8. {U}QSUB16<cond>      Rd, Rn, Rm        ARMv6
9. {U}QADD8<cond>       Rd, Rn, Rm        ARMv6
10. {U}QSUB8<cond>      Rd, Rn, Rm        ARMv6
```

Action

```
 1. Rd = sat32(Rm+Rn)
 2. Rd = sat32(Rm+sat32(2*Rn))
 3. Rd = sat32(Rm-Rn)
 4. Rd = sat32(Rm-sat32(2*Rn))
 5. Rd[31:16] = sat16(Rn[31:16] + Rm[31:16]);
    Rd[15:00] = sat16(Rn[15:00] + Rm[15:00])
 6. Rd[31:16] = sat16(Rn[31:16] + Rm[15:00]);
    Rd[15:00] = sat16(Rn[15:00] - Rm[31:16])
 7. Rd[31:16] = sat16(Rn[31:16] - Rm[15:00]);
    Rd[15:00] = sat16(Rn[15:00] + Rm[31:16])
 8. Rd[31:16] = sat16(Rn[31:16] - Rm[31:16]);
    Rd[15:00] = sat16(Rn[15:00] - Rm[15:00])
 9. Rd[31:24] = sat8(Rn[31:24]  + Rm[31:24]);
    Rd[23:16] = sat8(Rn[23:16]  + Rm[23:16]);
    Rd[15:08] = sat8(Rn[15:08]  + Rm[15:08]);
    Rd[07:00] = sat8(Rn[07:00]  + Rm[07:00])
10. Rd[31:24] = sat8(Rn[31:24]  - Rm[31:24]);
    Rd[23:16] = sat8(Rn[23:16]  - Rm[23:16]);
    Rd[15:08] = sat8(Rn[15:08]  - Rm[15:08]);
    Rd[07:00] = sat8(Rn[07:00]  - Rm[07:00])
```

Notes

- The operations are signed unless the `U` prefix is present. For signed operations, `satN(x)` saturates x to the range $-2^{N-1} \le x < 2^{N-1}$. For unsigned operations, `satN(x)` saturates x to the range $0 \le x < 2^{N}$.
- The *cpsr Q*-flag is set if saturation occurred; otherwise it is preserved.
- *Rd*, *Rn*, *Rm* must not be *pc*.
- The `X` operations are useful for packed complex numbers. The following examples assume bits [15:00] hold the real part and [31:16] the imaginary part.

Examples

```
QDADD    r0, r0, r2   ; add Q30 value r2 to Q31 accumulator r0
QADD16   r0, r1, r2   ; SIMD saturating add
QADDSUBX r0, r1, r2   ; r0=r1+i*r2 in packed complex arithmetic
QSUBADDX r0, r1, r2   ; r0=r1-i*r2 in packed complex arithmetic
```

`REV` Reverse bytes within a word or halfword.

```
1. REV<cond>      Rd, Rm            ARMv6/THUMBv3
2. REV16<cond>    Rd, Rm            ARMv6/THUMBv3
3. REVSH<cond>    Rd, Rm            ARMv6/THUMBv3
```

Action

```
1. Rd[31:24] = Rm[07:00]; Rd[23:16] = Rm[15:08];
   Rd[15:08] = Rm[23:16]; Rd[07:00] = Rm[31:24]
2. Rd[31:24] = Rm[23:16]; Rd[23:16] = Rm[31:24];
   Rd[15:08] = Rm[07:00]; Rd[07:00] = Rm[15:08]
3. Rd[31:08] = sign-extend(Rm[07:00]); Rd[07:00] = Rm[15:08]
```

Notes

- *Rd* and *Rm* must not be *pc*.
- For Thumb, *Rd*, *Rm* must be in the range *r0* to *r7* and <cond> cannot be specified.
- These instructions are useful to convert big-endian data to little-endian and vice versa.

Examples

```
REV     r0, r0 ; switch endianness of a word
REV16   r0, r0 ; switch endianness of two packed halfwords
REVSH   r0, r0 ; switch endianness of a signed halfword
```

`RFE` Return from exception

```
1. RFE<amode> Rn!                                   ARMv6
```

This performs the operation that `LDM<amode> Rn{!}, {pc, cpsr}` would perform if `LDM` allowed a register list of `{pc, cpsr}`. See the entry for LDM.

`ROR` Rotate right for Thumb (see `MOV` for the ARM equivalent)

```
1. ROR Ld, Ls                                       THUMBv1
```

Action	Effect on the *cpsr*
`1. Ld = Ld ROR Ls[7:0]`	`Updated`

Notes

- The *cpsr* is updated: *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*shifter_C*> (see Table B1.3).

`RSB` Reverse subtract of two 32-bit integers

```
1. RSB<cond>{S} Rd, Rn, #<rotated_immed>            ARMv1
2. RSB<cond>{S} Rd, Rn, Rm {, <shift>}              ARMv1
```

Action	Effect on the *cpsr*
`1. Rd = <rotwated_immed> - Rn`	`Updated if S suffix present`
`2. Rd = <shifted_Rm> - Rn`	`Updated if S suffix present`

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*NoUnsignedOverflow*>, and *V* = <*SignedOverflow*>. The carry flag is set this way because the subtract $x - y$ is implemented as the add $x + \sim y + 1$. The carry flag is one if $x + \sim y + 1$ overflows. This happens when $x \geq y$, when $x - y$ doesn't overflow.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr* in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Examples

```
RSB     r0, r0, #0          ; r0 = -r0
RSB     r0, r1, r1, LSL#3   ; r0 = 7*r1
```

RSC Reverse subtract with carry of two 32-bit integers

```
1. RSC<cond>{S} Rd, Rn, #<rotated_immed>  ARMv1
2. RSC<cond>{S} Rd, Rn, Rm {, <shift>}    ARMv1
```

Action	Effect on the *cpsr*
1. Rd = <rotated_immed> - Rn - (~C)	Updated if S suffix present
2. Rd = <shifted_Rm> - Rn - (~C)	Updated if S suffix present

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*NoUnsignedOverflow*>, *V* = <*SignedOverflow*>. The carry flag is set this way because the subtract $x - y - \sim C$ is implemented as the add $x + \sim y + C$. The carry flag is one if $x + \sim y + C$ overflows. This happens when $x - y - \sim C$ doesn't overflow.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

The following example negates a 64-bit integer where *r0* is the low 32 bits and *r1* the high 32 bits.

```
RSBS    r0, r0, #0    ; r0 = -r0    C=NOT(borrow)
RSC     r1, r1, #0    ; r1 = -r1-borrow
```

SADD Parallel modulo add and subtract operations

```
1. {S|U}ADD16<cond>      Rd, Rn, Rm      ARMv6
2. {S|U}ADDSUBX<cond>    Rd, Rn, Rm      ARMv6
```

```
 3. {S|U}SUBADDX<cond>    Rd, Rn, Rm        ARMv6
 4. {S|U}SUB16<cond>      Rd, Rn, Rm        ARMv6
 5. {S|U}ADD8<cond>       Rd, Rn, Rm        ARMv6
 6. {S|U}SUB8<cond>       Rd, Rn, Rm        ARMv6
```

Action	Effect on the *cpsr*
1. Rd[31:16]=Rn[31:16]+Rm[31:16];	GE3=GE2=cmn(Rn[31:16],Rm[31:16])
Rd[15:00]=Rn[15:00]+Rm[15:00]	GE1=GE0=cmn(Rn[15:00],Rm[15:00])
2. Rd[31:16]=Rn[31:16]+Rm[15:00];	GE3=GE2=cmn(Rn[31:16],Rm[15:00])
Rd[15:00]=Rn[15:00]-Rm[31:16]	GE1=GE0=(Rn[15:00] >= Rm[31:16])
3. Rd[31:16]=Rn[31:16]-Rm[15:00];	GE3=GE2=(Rn[31:16] >= Rm[15:00])
Rd[15:00]=Rn[15:00]+Rm[31:16]	GE1=GE0=cmn(Rn[15:00],Rm[31:16])
4. Rd[31:16]=Rn[31:16]-Rm[31:16];	GE3=GE2=(Rn[31:16] >= Rm[31:16])
Rd[15:00]=Rn[15:00]-Rm[15:00]	GE1=GE0=(Rn[15:00] >= Rm[15:00])
5. Rd[31:24]=Rn[31:24]+Rm[31:24];	GE3 = cmn(Rn[31:24],Rm[31:24])
Rd[23:16]=Rn[23:16]+Rm[23:16];	GE2 = cmn(Rn[23:16],Rm[23:16])
Rd[15:08]=Rn[15:08]+Rm[15:08];	GE1 = cmn(Rn[15:08],Rm[15:08])
Rd[07:00]=Rn[07:00]+Rm[07:00]	GE0 = cmn(Rn[07:00],Rm[07:00])
6. Rd[31:24]=Rn[31:24]-Rm[31:24];	GE3 = (Rn[31:24] >= Rm[31:24])
Rd[23:16]=Rn[23:16]-Rm[23:16];	GE2 = (Rn[23:16] >= Rm[23:16])
Rd[15:08]=Rn[15:08]-Rm[15:08];	GE1 = (Rn[15:08] >= Rm[15:08])
Rd[07:00]=Rn[07:00]-Rm[07:00]	GE0 = (Rn[07:00] >= Rm[07:00])

Notes

- If you specify the `S` prefix, then all comparisons are signed. The *cmn(x, y)* function returns $x \geq -y$ or equivalently $x + y \geq 0$.
- If you specify the `U` prefix, then all comparisons are unsigned. The *cmn(x, y)* function returns $x \geq$ (*unasigned*) $(-y)$ or equivalently if the $x + y$ operation produces a carry.
- *Rd*, *Rn*, and *Rm* must not be *pc*.
- The `X` operations are useful for packed complex numbers. The following examples assume bits [15:00] hold the real part and [31:16] the imaginary part.

Examples

```
SADD16      r0, r1, r2   ; Signed 16-bit SIMD add
SADDSUBX    r0, r1, r2   ; r0=r1+i*r2 in packed complex arithmetic
 SSUBADDX   r0, r1, r2   ; r0=r1-i*r2 in packed complex arithmetic
```

SBC Subtract with carry

```
 1. SBC<cond>{S} Rd, Rn, #<rotated_immed>       ARMv1
 2. SBC<cond>{S} Rd, Rn, Rm {, <shift>}         ARMv1
 3. SBC          Ld, Lm                         THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = Rn - <rotated_immed> - (~C)	Updated if S suffix specified
2. Rd = Rn - <shifted_Rm> - (~C)	Updated if S suffix specified
3. Ld = Ld - Lm - (~C)	Updated (see Notes below)

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then $N = <Negative>$, $Z = <Zero>$, $C = <NoUnsignedOverflow>$, $V = <SignedOverflow>$. The carry flag is set this way because the subtract $x - y - \sim C$ is implemented as the add $x + \sim y + C$. The carry flag is one if $x + \sim y + C$ overflows. This happens when $x - y - \sim C$ doesn't overflow.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*. In this case the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

The following example implements a 64-bit subtract:

```
SUBS    r0, r0, r2    ; subtract low words, C=NOT(borrow)
SBC     r1, r1, r3    ; subtract high words and borrow
```

`SEL` Select between two source operands based on the *GE* flags

```
1. SEL<cond> Rd, Rn, Rm                ARMv6
```

Action

```
1. Rd[31:24] = GE3 ? Rn[31:24] : Rm[31:24];
   Rd[23:16] = GE2 ? Rn[23:16] : Rm[23:16];
   Rd[15:08] = GE1 ? Rn[15:08] : Rm[15:08];
   Rd[07:00] = GE0 ? Rn[07:00] : Rm[07:00]
```

Notes

- *Rd*, *Rn*, *Rm* must not be *pc*.
- See `SADD` for instructions that set the *GE* flags in the *cpsr*.

`SETEND` Set the endianness for data accesses

```
1. SETEND BE                           ARMv6/THUMBv3
2. SETEND LE                           ARMv6/THUMBv3
```

Action

```
1. In the cpsr E=1 so data accesses will be big-endian
2. In the cpsr E=0 so data accesses will be little-endian
```

Note

- ARMv6 uses a byte-invariant endianness model. This means that byte loads and stores are not affected by the configured endianess. For little-endian data access the byte at the lowest address appears in the least significant byte of the loaded word. For big-endian data accesses the byte at the lowest address appears in the most significant byte of the loaded word.

SHADD Parallel halving add and subtract operations

```
1. {S|U}HADD16<cond>      Rd, Rn, Rm          ARMv6
2. {S|U}HADDSUBX<cond>    Rd, Rn, Rm          ARMv6
3. {S|U}HSUBADDX<cond>    Rd, Rn, Rm          ARMv6
4. {S|U}HSUB16<cond>      Rd, Rn, Rm          ARMv6
5. {S|U}HADD8<cond>       Rd, Rn, Rm          ARMv6
6. {S|U}HSUB8<cond>       Rd, Rn, Rm          ARMv6
```

Action

```
1. Rd[31:16] = (Rn[31:16] + Rm[31:16])>>1;
   Rd[15:00] = (Rn[15:00] + Rm[15:00])>>1
2. Rd[31:16] = (Rn[31:16] + Rm[15:00])>>1;
   Rd[15:00] = (Rn[15:00] - Rm[31:16])>>1
3. Rd[31:16] = (Rn[31:16] - Rm[15:00])>>1;
   Rd[15:00] = (Rn[15:00] + Rm[31:16])>>1
4. Rd[31:16] = (Rn[31:16] - Rm[31:16])>>1;
   Rd[15:00] = (Rn[15:00] - Rm[15:00])>>1
5. Rd[31:24] = (Rn[31:24] + Rm[31:24])>>1;
   Rd[23:16] = (Rn[23:16] + Rm[23:16])>>1;
   Rd[15:08] = (Rn[15:08] + Rm[15:08])>>1;
   Rd[07:00] = (Rn[07:00] + Rm[07:00])>>1
6. Rd[31:24] = (Rn[31:24] - Rm[31:24])>>1;
   Rd[23:16] = (Rn[23:16] - Rm[23:16])>>1;
   Rd[15:08] = (Rn[15:08] - Rm[15:08])>>1;
   Rd[07:00] = (Rn[07:00] - Rm[07:00])>>1
```

Notes

- If you use the `S` prefix, then all operations are signed and values are sign-extended before the addition.
- If you use the `U` prefix, then all operations are unsigned and values are zero-extended before the addition.
- *Rd*, *Rn*, and *Rm* must not be *pc*.

- These operations provide parallel arithmetic that cannot overflow, which is useful for DSP processing of normalized signals.

SMLS
SMLA Signed multiply accumulate instructions

```
1. SMLA<x><y><cond>    Rd,   Rm,    Rs,    Rn     ARMv5E
2. SMLAW<y><cond>      Rd,   Rm,    Rs,    Rn     ARMv5E
3. SMLAD{X}<cond>      Rd,   Rm,    Rs,    Rn     ARMv6
4. SMLSD{X}<cond>      Rd,   Rm,    Rs,    Rn     ARMv6
5. {U|S}MLAL<cond>{S}  RdLo, RdHi,  Rm,    Rs     ARMv3M
6. SMLAL<x><y><cond>   RdLo, RdHi,  Rm,    Rs     ARMv5E
7. SMLALD{X}<cond>     RdLo, RdHi,  Rm,    Rs     ARMv6
8. SMLSLD{X}<cond>     RdLo, RdHi,  Rm,    Rs     ARMv6
```

Action

```
1. Rd = Rn + (Rm.<x> * Rs.<y>)
2. Rd = Rn + (((signed)Rm * Rs.<y>)>>16)
3. Rd = Rn + Rm.B*<rotated_Rs>.B + Rm.T*<rotated_Rs>.T
4. Rd = Rn + Rm.B*<rotated_Rs>.B - Rm.T*<rotated_Rs>.T
5. RdHi:RdLo = RdHi:RdLo + (Rm * Rs)
6. RdHi:RdLo = RdHi:RdLo + (Rm.<x> * Rm.<y>)
7. RdHi:RdLo = RdHi:RdLo + Rm.B*<rotated_Rs>.B + Rm.T*<rotated_Rs>.T
8. RdHi:RdLo = RdHi:RdLo + Rm.B*<rotated_Rs>.B - Rm.T*<rotated_Rs>.T
```

Notes

- <x> and <y> can be B or T.
- *Rm.B* is shorthand for *(sign-extend)Rm*[15:00], the bottom 16 bits of *Rm*.
- *Rm.T* is shorthand for *(sign-extend)Rm*[31:16], the top 16 bits of *Rm*.
- *<rotated_Rs>* is *Rs* if you do not specify the X suffix or Rs ROR 16 if you do specify the X suffix.
- *RdHi* and *RdLo* must be different registers. For format 5, *Rm* must be a different register from *RdHi* and *RdLo*.
- Formats 1 to 4 update the *cpsr* Q-flag: *Q* = Q < *SignedOverflow*>.
- Format 5 implements an unsigned multiply with the U prefix or a signed multiply with the S prefix.
- Format 5 updates the *cpsr* if the S suffix is present: *N* = *RdHi*[31], *Z* = (*RdHi*==0 && *RdLo*==0); the *C* and *V* flags are unpredictable. Avoid using {U|S}MLALS because implementations often impose penalty cycles for this operation.
- Implementations may terminate early on the value of *Rs*. For this reason use small or constant values for *Rs* where possible.

- The X suffix and multiply subtract versions are useful for packed complex numbers. The following examples assume bits [15:00] hold the real part and [31:16] the imaginary part.

  Examples

```
SMLABB  r0, r1, r2, r0 ;   r0  += (short)r1 * (short)r2
SMLABT  r0, r1, r2, r0  ;  r0  += (short)r1 * ((signed)r>>216)
SMLAWB  r0, r1, r2, r0  ;  r0  += (r1*(short)r2)>>16
SMLAL   r0, r1, r2, r3  ;  acc += r2*r3, acc is 64 bits [r1:r0]
SMLALTB r0, r1, r2, r3  ;  acc += ((signed)r2>>16)*((short)r3)
SMLSD   r0, r1, r2, r0  ;  r0  += real(r1*r2) in complex maths
SMLADX  r0, r1, r2, r0  ;  r0  += imag(r1*r2) in complex maths
```

SMMUL Signed most significant word multiply instructions
SMMLA
SMMLS

```
1. SMMUL{R}<cond> Rd, Rm, Rs                    ARMv6
2. SMMLA{R}<cond> Rd, Rm, Rs, Rn                ARMv6
3. SMMLS{R}<cond> Rd, Rm, Rs, Rn                ARMv6
```

Action

```
1. Rd = ((signed)Rm*(signed)Rs + round)>>32
2. Rd = ((Rn<<32) + (signed)Rm*(signed)Rs + round)>>32
3. Rd = ((Rn<<32) - (signed)Rm*(signed)Rs + round)>>32
```

Notes

- If you specify the R suffix then *round* = 2^{31}; otherwise, *round* = 0.
- *Rd*, *Rm*, *Rs*, and *Rn* must not be *pc*.
- Implementations may terminate early on the value of *Rs*.
- For 32-bit DSP algorithms these operations have several advantages over using the high result register from SMLAL: They often take fewer cycles than SMLAL. They also implement rounding, multiply subtract, and don't require a temporary scratch register for the low 32 bits of result.

  Example

```
SMMULR  r0, r1, r2    ; r0=r1*r2/2 using Q31 arithmetic
```

SMUL Signed multiply instructions
SMUA
SMUS

```
1. SMUL<x><y><cond>   Rd,   Rm,   Rs            ARMv5E
2. SMULW<y><cond>     Rd,   Rm,   Rs            ARMv5E
3. SMUAD{X}<cond>     Rd,   Rm,   Rs            ARMv6
```

```
4. SMUSD{X}<cond>     Rd,    Rm,    Rs            ARMv6
5. {U|S}MULL<cond>{S} RdLo,  RdHi,  Rm,  Rs       ARMv3M
```

Action

```
1. Rd     = Rm.<x> * Rs.<y>
2. Rd     = (Rm * Rs.<y>)>>16
3. Rd     = Rm.B*<rotated_Rs>.B + Rm.T*<rotated_Rs>.T
4. Rd     = Rm.B*<rotated_Rs>.B - Rm.T*<rotated_Rs>.T
5. RdHi:RdLo = Rm*Rs
```

Notes

- <x> and <y> can be B or T.
- *Rm.B* is shorthand for *(sign-extend)Rm*[15:00], the bottom 16 bits of *Rm*.
- *Rm.T* is shorthand for *(sign-extend)Rm*[31:16], the top 16 bits of *Rm*.
- *<rotated_Rs>* is *Rs* if you do not specify the X suffix or Rs ROR 16 if you do specify the X suffix.
- *RdHi* and *RdLo* must be different registers. For format 5, *Rm* must be a different register from *RdHi* and *RdLo*.
- Format 4 updates the *cpsr* *Q*-flag: *Q* = *Q* | *<SignedOverflow>*.
- Format 5 implements an unsigned multiply with the U prefix or a signed multiply with the S prefix.
- Format 5 updates the *cpsr* if the S suffix is present: *N* = *RdHi*[31], *Z* = (*RdHi*==0 && *RdLo*==0); the *C* and *V* flags are unpredictable. Avoid using {SIU}MULLS because implementations often impose penalty cycles for this operation.
- Implementations may terminate early on the value of *Rs*. For this reason use small or constant values for *Rs* where possible.
- The X suffix and multiply subtract versions are useful for packed complex numbers. The following examples assume bits [15:00] hold the real part and [31:16] the imaginary part.

Examples

```
SMULBB r0, r1, r2     ; r0  = (short)r1 * (short)r2
SMULBT r0, r1, r2     ; r0  = (short)r1 * ((signed)r2>>16)
SMULWB r0, r1, r2     ; r0  = (r1*(short)r2)>>16
SMULL  r0, r1, r2, r3 ; acc =  r2*r3, acc is 64 bits [r1:r0]
SMUADX r0, r1, r2     ; r0  = imag(r1*r2) in complex maths
```

SRS Save return state

```
1. SRS<amode> #<mode>{!}                  ARMv6
```

This performs the operation that *STM<amode> sp_<mode>{!}* , *{lr, spsr}* would perform if STM allowed a register list of *{lr, spsr}* and allowed you to reference the stack pointer of a different mode. See the entry for STM.

SSAT Saturate to *n* bits

```
1. {S|U}SAT<cond>   Rd, #<n>, Rm {, LSL#<0-31>}
2. {S|U}SAT<cond>   Rd, #<n>, Rm {, ASR#<1-32>}
3. {S|U}SAT16<cond> Rd, #<n>, Rm
```

Action	Effect on the *cpsr*
1. Rd = sat(<shifted_Rm>, n);	Q=Q | 1 if saturation occurred
2. Rd = sat(<shifted_Rm>, n);	Q=Q | 1 if saturation occurred
3. Rd[31:16] = sat(Rm[31:16], n); Rd[15:00] = sat(Rm[15:00], n)	Q=Q | 1 if saturation occurred

Notes

- If you specify the S prefix, then *sat* (*x, n*) saturates the signed value *x* to a signed *n*-bit value in the range $-2^{n-1} \leq x < 2^{n-1}$. *n* is encoded as 1 + <*immed5*> for SAT and 1 + <*immed4*> for SAT16.
- If you specify the U prefix, then *sat* (*x, n*) saturates the signed value *x* to an unsigned *n*-bit value in the range $0 \leq x \leq 2^n$. *n* is encoded as <*immed5*> for SAT and <*immed4*> for SAT16.
- *Rd* and *Rm* must not be *pc*.

SSUB Signed parallel subtract (see SADD)

STC Store to coprocessor single or multiple 32-bit values

```
1. STC<cond>{L}  <copro>, Cd,  [Rn {, #{-}<immed8>*4}]{!} ARMv2
2. STC<cond>{L}  <copro>, Cd,  [Rn], #{-}<immed8>*4       ARMv2
3. STC<cond>{L}  <copro>, Cd,  [Rn], <option>             ARMv2
4. STC2{L}       <copro>, Cd,  [Rn {, #{-}<immed8>*4}]{!} ARMv5
5. STC2{L}       <copro>, Cd,  [Rn], #{-}<immed8>*4       ARMv5
6. STC2{L}       <copro>, Cd,  [Rn], <option>             ARMv5
```

These instructions initiate a memory write, transferring data to memory from the given coprocessor. <copro> is the number of the coprocessor in the range *p*0 to *p*15. The core takes an undefined instruction trap if the coprocessor is not present. The memory write consists of a sequence of words to sequentially increasing addresses. The initial address is specified by the addressing mode in Table B1.10. The coprocessor controls the number of words transferred, up to a maximum

limit of 16 words. The fields *{L}* and *Cd* are interpreted by the coprocessor and ignored by the ARM. Typically *Cd* specifies the source coprocessor register for the transfer. The `<option>` field is an eight-bit integer enclosed in *{}*. Its interpretation is coprocessor dependent.

If the address is not a multiple of four, then the access is unaligned. The restrictions on an unaligned access are the same as for `STM`.

TABLE B1.10 STC addressing modes.

Addressing format	Address accessed	Value written back to *Rn*
[Rn {, #{-}<immed>}]	Rn + {{-}<immed>}	Rn preserved
[Rn {, #{-}<immed>}]!	Rn + {{-}<immed>}	Rn + {{-}<immed>}
[Rn], #{-}<immed>	Rn	Rn + {-}<immed>
[Rn], <option>	Rn	Rn preserved

`STM` Store multiple 32-bit registers to memory

```
1. STM<cond><a mode> Rn{!}, <register_list>{^}    ARMv1
2. STMIA             Rn!,   <register_list>       THUMBv1
```

These instructions store multiple words to sequential memory addresses. The *<register_list>* specifies a list of registers to store, enclosed in curly brackets *{}*. Although the assembler allows you to specify the registers in the list in any order, the order is not stored in the instruction, so it is good practice to write the list in increasing order of register number since this is the usual order of the memory transfer.

The following pseudocode shows the normal action of `STM`. We use `<register_list>[i]` to denote the register appearing at position `i` in the list starting at `0` for the first register. This assumes that the list is in order of increasing register number.

```
N = the number of registers in <register_list>
start = the lowest address accessed given in Table B1.11
 for (i=0; i<N; i++)
 memory(start+i*4, 4) = <register_list>[i];
 if (! specified) then update Rn according to Table B1.11
```

Note that `memory(a, 4)` refers to the four bytes at address a packed according to the current processor data endianness. If a is not a multiple of four, then the store is unaligned. Because the behavior of an unaligned store depends on the architecture revision, memory system, and system coprocessor (CP15) configuration, it is best to avoid unaligned stores if possible. Assuming that the external memory system does not abort unaligned stores, then the following rules usually apply:

- If the core has a system coprocessor and bit 1 (*A*-bit) or bit 22 (*U*-bit) of CP15: c1:c0:0 is set, then unaligned store-multiples cause an alignment fault data abort exception.
- Otherwise, the access ignores the bottom two address bits.

Table B1.11 lists the possible addressing modes specified by `<amode>`. If you specify the !, then the base address register is updated according to Table B1.11; otherwise, it is preserved. Note that the lowest register number is always written to the lowest address.

TABLE B1.11 STM addressing modes.

Addressing mode	Lowest address accessed	Highest address accessed	Value written back to *Rn* if ! specified
{IA|EA}	Rn	Rn + N*4 - 4	Rn + N*4
{IB|FA}	Rn + 4	Rn + N*4	Rn + N*4
{DA|ED}	Rn - N*4 + 4	Rn	Rn - N*4
{DB|FD}	Rn - N*4	Rn - 4	Rn - N*4

The first half of the addressing mode mnemonics stands for Increment After, Increment Before, Decrement After, and Decrement Before, respectively. Increment modes store the registers sequentially forward starting from address *Rn* (increment after) or *Rn* + 4 (increment before). Decrement modes have the same effect as if you stored the register list backwards to sequentially descending memory addresses starting from address *Rn* (decrement after) or *Rn* − 4 (decrement before).

The second half of the addressing mode mnemonics stands for the stack type you can implement with that address mode: Full Descending, Empty Descending, Full Ascending, and Empty Ascending. With a full stack, *Rn* points to the last stacked value. With an empty stack, *Rn* points to the first unused stack location. ARM stacks are usually full descending. You should use full descending or empty ascending stacks by preference, since `STC` also supports these addressing modes.

Notes

- For Thumb (format 2), *Rn* and the register list registers must be in the range *r0* to *r7*.
- The number of registers *N* in the list must be nonzero.
- *Rn* must not be *pc*.
- If *Rn* appears in the register list and ! (writeback) is specified, the behavior is as follows: If *Rn* is the lowest register number in the list, then the original value is stored; otherwise, the stored value is unpredictable.

- If *pc* appears in the register list, then the value stored is implementation defined.
- If ^ is specified, then the operation is modified. The processor must not be in *user* or *system* mode. The registers appearing in the register list refer to the *user* mode versions of the registers and writeback must not be specified.
- The time order of the memory accesses may depend on the implementation. Be careful when using a store multiple to access I/O locations where the access order matters. If the order matters, then check that the memory locations are marked as I/O in the page tables. Do not cross page boundaries, and do not use *pc* in the register list.

Examples

```
STMIA   r4!, {r0, r1}   ; *r4=r0, *(r4+4)=r1, r4+=8
STMDB   r4!, {r0, r1}   ; *(r4-4)=r1, *(r4-8)=r0, r4-=8
STMEQFD sp!, {r0, lr}   ; if (result zero) then stack r0, lr
STMFD   sp,  {sp}^      ; store sp_usr on stack sp_current
```

STR Store a single value to a virtual address in memory

```
 1. STR<cond>{|B}    Rd,  [Rn {, #{-}<immed12>}]{!}       ARMv1
 2. STR<cond>{|B}    Rd,  [Rn, {-}Rm {,<imm_shift>}]{!}   ARMv1
 3. STR<cond>{|B}{T} Rd,  [Rn], #{-}<immed12>             ARMv1
 4. STR<cond>{|B}{T} Rd,  [Rn], {-}Rm {,<imm_shift>}      ARMv1
 5. STR<cond>{H}     Rd,  [Rn, {, #{-}<immed8>}]{!}       ARMv4
 6. STR<cond>{H}     Rd,  [Rn, {-}Rm]{!}                  ARMv4
 7. STR<cond>{H}     Rd,  [Rn], #{-}<immed8>              ARMv4
 8. STR<cond>{H}     Rd,  [Rn], {-}Rm                     ARMv4
 9. STR<cond>D       Rd,  [Rn, {, #{-}<immed8>}]{!}       ARMv5E
10. STR<cond>D       Rd,  [Rn, {-}Rm]{!}                  ARMv5E
11. STR<cond>D       Rd,  [Rn], #{-}<immed8>              ARMv5E
12. STR<cond>D       Rd,  [Rn], {-}Rm                     ARMv5E
13. STREX<cond>      Rd,  Rm, [Rn]                        ARMv6
14. STR{|B|H}        Ld,  [Ln, #<immed5>*<size>]          THUMBv1
15. STR{|B|H}        Ld,  [Ln, Lm]                        THUMBv1
16. STR              Ld,  [sp, #<immed8>*4]               THUMBv1
17. STR<cond><type>  Rd,  <label>                         MACRO
```

Formats 1 to 16 store a single data item of the type specified by the opcode suffix, using a preindexed or postindexed addressing mode. Tables B1.12 and B1.13 show the different addressing modes and data types.

In Table B1.13, `memory` (a, n) refers to n sequential bytes at address `a`. The bytes are packed according to the configured processor data endianness. `memoryT(a,n)` performs the access with *user* mode privileges, regardless of the current processor mode. The act of function `IsExclusive`(a) used by `STREX` depends on address `a`. If a has the shared TLB attribute, then `IsExclusive(a)` is true if address `a` is marked as exclusive for this processor. It then clears any exclusive accesses on this processor and any exclusive

accesses to address a on other processors in the system. If a does not have the shared TLB attribute, then `IsExclusive(a)` is true if there is an outstanding exclusive access on this processor. It then clears any such outstanding access.

TABLE B1.12 STR addressing modes.

Addressing format	Address *a* accessed	Value written back to *Rn*
[Rn {,#{-}<immed>}]	Rn + {{-}<immed>}	Rn preserved
[Rn {,#{-}<immed>}]!	Rn + {{-}<immed>}	Rn + {{-}<immed>}
[Rn, {-}Rm {,<shift>}]	Rn + {-}<shifted_Rm>	Rn preserved
[Rn, {-}Rm {,<shift>}]!	Rn + {-}<shifted_Rm>	Rn + {-}<shifted_Rm>
[Rn], #{-}<immed>	Rn	Rn + {-}<immed>
[Rn], {-}Rm {,<shift>}	Rn	Rn + {-}<shifted_Rm>

TABLE B1.13 STR data types.

Store	Datatype	<size> (bytes)	Action
STR	word	4	memory(a, 4) = Rd
STRB	unsigned Byte	1	memory(a, 1) = (char)Rd
STRBT	Byte Translated	1	memoryT(a, 1) = (char)Rd
STRD	Double word	8	memory(a, 4) = Rd
			memory(a+4, 4) = R(d+1)
STREX	word EXclusive	4	if (IsExclsuive(a)) {
			memory (a, 4) = Rm;
			Rd = 0;
			} else {
			Rd = 1;
			}
STRH	unsigned Halfword	2	memory(a, 2) = (short) Rd
STRT	word Translated	4	memoryT(a, 4) = Rd

If the address *a* is not a multiple of <*size*>, then the store is unaligned. Because the behavior of an unaligned store depends on the architecture revision, memory system, and system coprocessor (CP15) configuration, it is best to avoid unaligned stores if possible. Assuming that the external memory system does not abort unaligned stores, then the following rules usually apply. In the rules, *A* is bit 1 of system coprocessor register CP15:c1:c0:0, and *U* is bit 22 of CP15:c1:c0:0, introduced in ARMv6. If there is no system coprocessor, then $A = U = 0$.

- If *A* = 1, then unaligned stores cause an alignment fault data abort exception except that word-aligned double-word stores are supported if *U* = 1.
- If *A* = 0 and *U* = 1, then unaligned stores are supported for `STR{ITIHISH}`. Word-aligned stores are supported for `STRD`. A non-word-aligned `STRD` generates an alignment fault data abort.
- If *A* = 0 and *U* = 0, then `STR` and `STRT` write to *memory*(*a*&~ 3, 4). All other unaligned operations are unpredictable but do not cause an alignment fault

Format 17 generates a *pc* -relative store accessing the address specified by <*label*> . In other words it assembles to *STR*<*cond*><*type*> *Rd, [pc, #*<*offset*>*]* whenever this instruction is supported and <*offset*>=<*label*>-*pc* is in range.

Notes

- For double-word stores (formats 9 to 12), *Rd* must be even and in the range *r0* to *r12*.
- If the addressing mode updates *Rn*, then *Rd* and *Rn* must be distinct.
- If *Rd* is *pc*, then <*size*> must be 4. The value stored is implementation defined.
- If *Rn* is *pc*, then the addressing mode must not update *Rn* . The value used for *Rn* is the address of the instruction plus eight bytes.
- *Rm* must not be *pc*.

Examples

```
STR   r0,  [r0]       ; *(int*)r0 = r0;
STRH  r0,  [r1], #4   ; *(short*)r1 = r0; r1+=4;
STRD  r2,  [r1, #-8]! ; r1-=8; *(int*)r1=r2; *(int*)(r1+4)=r3
STRB  r0,  [r2, #55]        ; *(char*)(r2+55) = r0;
STRB  r0,  [r1], -r2, LSL #8 ; *(char*)r1 = r0; r1-=256*r2;
```

`SUB` Subtract two 32-bit values

```
1. SUB<cond>{S}  Rd, Rn, #<rotated_immed>          ARMv1
2. SUB<cond>{S}  Rd, Rn, Rm {, <shift>}            ARMv1
3. SUB           Ld, Ln, #<immed3>                 THUMBv1
4. SUB           Ld, #<immed8>                     THUMBv1
5. SUB           Ld, Ln, Lm                        THUMBv1
6. SUB           sp, #<immed7>*4                   THUMBv1
```

Action	Effect on the *cpsr*
1. Rd = Rn - <rotated_immed>	Updated if S suffix specified
2. Rd = Rn - <shifted_Rm>	Updated if S suffix specified
3. Ld = Ln - <immed3>	Updated (see Notes below)
4. Ld = Ld - <immed8>	Updated (see Notes below)

```
5. Ld = Ln - Lm                 Updated (see Notes below)
6. sp = sp - <immed7>*4         Preserved
```

Notes

- If the operation updates the *cpsr* and *Rd* is not *pc*, then *N* = <*Negative*>, *Z* = <*Zero*>, *C* = <*NoUnsignedOverflow*>, and *V* = <*SignedOverflow*>. The carry flag is set this way because the subtract $x - y$ is implemented as the add $x + \sim y + 1$. The carry flag is one if $x + \sim y + 1$ overflows. This happens when $x \geq y$, when $x - y$ doesn't overflow.
- If *Rd* is *pc*, then the instruction effects a jump to the calculated address. If the operation updates the *cpsr*, then the processor mode must have an *spsr*; in this case, the *cpsr* is set to the value of the *spsr*.
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

Examples

```
SUBS  r0, r0, #1            ; r0-=1, setting flags
SUB   r0, r1, r1, LSL #2    ; r0 = -3*r1
SUBS  pc, lr, #4            ; jump to lr-4, set cpsr=spsr
```

SWI Software interrupt

```
1. SWI<cond> <immed24>                          ARMv1
2. SWI <immed8>                                 THUMBv1
```

The SWI instruction causes the ARM to enter *supervisor* mode and start executing from the SWI vector. The return address and *cpsr* are saved in *lr_svc* and *spsr_svc*, respectively. The processor switches to ARM state and IRQ interrupts are disabled. The SWI vector is at address 0x00000008, unless high vectors are configured; then it is at address 0xFFFF0008.

The immediate operand is ignored by the ARM. It is normally used by the SWI exception handler as an argument determining which function to perform.

Example

```
SWI 0x123456 ; Used by the ARM tools to implement Semi-Hosting
```

SWP Swap a word in memory with a register, without interruption

```
1. SWP<cond>  Rd, Rm, [Rn]                      ARMv2a
2. SWP<cond>B Rd, Rm, [Rn]                      ARMv2a
```

Action

```
1. temp=memory(Rn,4); memory(Rn,4)=Rm; Rd=temp;
2. temp=(zero extend)memory(Rn,1); memory(Rn,1)=(char)Rm; Rd=temp;
```

Notes

- The operations are atomic. They cannot be interrupted partway through.
- *Rd*, *Rm*, *Rn* must not be *pc*.
- *Rn* and *Rm* must be different registers. *Rn* and *Rd* must be different registers.
- *Rn* should be aligned to the size of the memory transfer.
- If a data abort occurs on the load, then the store does not occur. If a data abort occurs on the store, then *Rd* is not written.

You can use the SWP instruction to implement 8-bit or 32-bit semaphores on ARMv5 and below. For ARMv6 use LDREX and STREX in preference. As an example, suppose a byte semaphore register pointed to by *r1* can have the value 0xFF (claimed) or 0x00 (free). The following example claims the lock. If the lock is already claimed, then the code loops, waiting for an interrupt or task switch that will free the lock.

```
      MOV   r0, #0xFF       ; value to claim the lock
loops WPB   r0, r0, [r1]    ; try and claim the lock
      CMP   r0, #0xFF       ; check to see if it was already claimed
      BEQ   loop            ; if so wait for it to become free
```

SXT Byte or halfword extract or extract with accumulate
SXTA

```
1. {S|U}XTB16<cond>    Rd, Rm   {,  ROR#8*<rot> }       ARMv6
2. {S|U}XTB<cond>      Rd, Rm   {,  ROR#8*<rot> }       ARMv6
3. {S|U}XTH<cond>      Rd, Rm   {,  ROR#8*<rot> }       ARMv6
4. {S|U}XTAB16<cond>   Rd, Rn, Rm  {, ROR#8*<rot> }     ARMv6
5. {S|U}XTAB<cond>     Rd, Rn, Rm  {, ROR#8*<rot> }     ARMv6
6. {S|U}XTAH<cond>     Rd, Rn, Rm  {, ROR#8*<rot> }     ARMv6
7. {S|U}XTB            Ld, Lm   THUMBv3
8. {S|U}XTH            Ld, Lm   THUMBv3
```

Action

```
1. Rd[31:16] = extend(<shifted_Rm>[23:16]);
   Rd[15:00] = extend(<shifted_Rm>[07:00])
2. Rd = extend(<shifted_Rm>[07:00])
3. Rd = extend(<shifted_Rm>[15:00])
4. Rd[31:16] = Rn[31:16] + extend(<shifted_Rm>[23:16]);
   Rd[15:00] = Rn[15:00] + extend(<shifted_Rm>[07:00])
5. Rd = Rn + extend(<shifted_Rm>[07:00])
6. Rd = Rn + extend(<shifted_Rm>[15:00])
7. Ld = extend(Lm[07:00])
8. Ld = extend(Lm[15:00])
```

Notes

- If you specify the `S` prefix, then extend(*x*) sign extends *x*.
- If you specify the `U` prefix, then extend(*x*) zero extends *x*.
- *Rd* and *Rm* must not be *pc*.
- *<rot>* is an immediate in the range 0 to 3.

`TEQ` Test for equality of two 32-bit values

```
1. TEQ<cond> Rn, #<rotated_immed>                    ARMv1
2. TEQ<cond> Rn, Rm {, <shift>}                      ARMv1
```

Action

```
1. Set the cpsr on the result of (Rn ^ <rotated_immed>)
2. Set the cpsr on the result of (Rn ^ <shifted_Rm>)
```

Notes

- The *cpsr* is updated: *N* = *<Negative>*, *Z* = *<Zero>*, *C* = *<shifter_C>* (see Table B1.3).
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.
- Use this instruction instead of `CMP` when you want to check for equality and preserve the carry flag.

Example

```
  TEQ     r0, #1        ; test to see if r0==1
```

`TST` Test bits of a 32-bit value

```
1. TST<cond> Rn, #<rotated_immed>                    ARMv1
2. TST<cond> Rn, Rm {, <shift>}                      ARMv1
3. TST       Ln, Lm                                  THUMBv1
```

Action

```
1. Set the cpsr on the result of (Rn & <rotated_immed>)
2. Set the cpsr on the result of (Rn & <shifted_Rm>)
3. Set the cpsr on the result of (Ln & Lm)
```

Notes

- The *cpsr* is updated: *N* = *<Negative>*, *Z* = *<Zero>*, *C* = *<shifter_C>* (see Table B1.3).
- If *Rn* or *Rm* is *pc*, then the value used is the address of the instruction plus eight bytes.

- Use this instruction to test whether a selected set of bits are all zero.

  Example

```
TST r0, #0xFF   ; test if the bottom 8 bits of r0 are 0
```

`UADD` Unsigned parallel modulo add (see the entry for `SADD`)

`UHADD` Unsigned halving add and subtract (see the entry for `SHADD`)
`UHSUB`

`UMAAL` Unsigned multiply accumulate accumulate long

```
1. UMAAL<cond> RdLo, RdHi, Rm, Rs   ARMv6
```

Action

```
1. RdHi:RdLo = (unsigned)Rm*Rs + (unsigned)RdLo + (unsigned)RdHi
```

Notes

- *RdHi* and *RdLo* must be different registers.
- *RdHi*, *RdLo*, *Rm*, *Rs* must not be *pc*.
- This operation cannot overflow because $(2^{32}-1)\ (2^{32}-1)+(2^{32}-1)+(2^{32}-1)=(2^{64}-1)$. You can use it to synthesize the multiword multiplications used by public key cryptosystems.

`UMLAL` Unsigned long multiply and multiply accumulate (see the `SMLAL` and
`UMULL` `SMULL` entries)

`UQADD` Unsigned saturated add and subtract (see the `QADD` entry)
`UQSUB`

`USAD` Unsigned sum of absolute differences

```
1. USAD8<cond>      Rd, Rm, Rs                  ARMv6
2. USADA8<cond>     Rd, Rm, Rs, Rn              ARMv6
```

Action

```
1. Rd =   abs(Rm[31:24]-Rs[31:24]) + abs(Rm[23:16]-Rs[23:16])
        + abs(Rm[15:08]-Rs[15:08]) + abs(Rm[07:00]-Rs[07:00])
2. Rd = Rn + abs(Rm[31:24]-Rs[31:24]) + abs(Rm[23:16]-Rs[23:16])
           + abs(Rm[15:08]-Rs[15:08]) + abs(Rm[07:00]-Rs[07:00])
```

Note

- *abs*(*x*) returns the absolute value of *x*. *Rm* and *Rs* are treated as unsigned.
- *Rd*, *Rm*, and *Rs* must not be *pc*.
- The sum of absolute differences operation is common in video codecs where it provides a metric to measure how similar two images are.

`USAT`	Unsigned saturation instruction (see the `SSAT` entry)
`USUB`	Unsigned parallel modulo subtracts (see the `SADD` entry)
`UXT` `UXTA`	Unsigned extract, extract with accumulate (see the entry for `SXT`)

B1.4 ARM Assembler Quick Reference

This section summarizes the more useful commands and expressions available with the ARM assembler, *armasm*. Each assembly line has one of the following formats:

```
{<label>}  {<instruction>}                              ; comment
{<symbol>} <directive>                                  ; comment
{<arg_0>}  <macro> {<arg_1>} {,<arg_2>} .. {,<arg_n>} ; comment
```

where

- <*instruction*> is any ARM or Thumb instruction supported by the processor you are assembling for. See Section B1.3.
- <*label*> is the name of a symbol to store the address of the instruction.
- <*directive*> is an ARM assembler directive. See Section *ARM Assembler Directives*.
- <*symbol*> is the name of a symbol used by the <*directive*>.
- <*macro*> is the name of a new directive defined using the `MACRO` directive.
- <*arg_k*> is the *k*th macro argument.

You must use an `AREA` directive to define an area before any ARM or Thumb instructions appear. All assembly files must finish with the END directive. The following example shows a simple assembly file defining a function add that returns the sum of the two input arguments:

```
      AREA    maths_routines, CODE, READONLY
      EXPORT  add          ; give the symbol add external linkage

add   ADD     r0, r0, r1    ; add input arguments
      MOV     pc, lr        ; return from sub-routine

      END
```

ARM Assembler Variables

The ARM assembler supports three types of assemble time variables (see Table B1.14). Variable names are case sensitive and must be declared before use with the directives `GBLx` or `LCLx`.

TABLE B1.14 ARM assembler variable types.

Variable type	Declare globally	Declare locally to a macro	Set value	Example values
Unsigned 32-bit integer	GBLA	LCLA	SETA	15, 0xab
ASCII string	GBLS	LCLS	SETS	“”, “ADD”
Logical	GBLL	LCLL	SETL	{TRUE}, {FALSE}

You can use variables in expressions (see Section *ARM Assembler Labels*), or substitute their value at assembly time using the $ operator. Specifically, `$name`. expands to the value of the variable `name` before the line is assembled. You can omit the final period if `name` is not followed by an alphanumeric or underscore. Use $$ to produce a single $. Arithmetic variables expand to an eight-digit hexadecimal string on substitution. Logical variables expand to *T* or *F*.

The following example code shows how to declare and substitute variables of each type:

```
      ; arithmetic variables
        GBLA  count  ; declare an integer variable count
count   SETA  1      ; set count = 1
        WHILE   count<15
          BL    test$count    ; call test00000001, test00000002 ...
count   SETA  count+1       ;  .... test00000000E
       WEND

      ; string variables
      GBLS   cc              ; declare a string variable called cc
cc    SETS   “NE”            ; set cc=”NE”
      ADD$cc r0, r0, r0      ; assembles as ADDNE  r0,r0,r0
      STR$cc.B r0, [r1]      ; assembles as STRNEB r0,[r1]
```

```
      ; logical variable
      GBLL    debug           ; declare a logical variable called debug
debug SETL    {TRUE}          ; set debug={TRUE}
      IF debug                ; if debug is TRUE then
        BL     print_debug    ; print out some debug information
      ENDIF
```

ARM Assembler Labels

A label definition must begin on the first character of a line. The assembler treats indented text as an instruction, directive, or macro. It treats labels of the form `<N><name>` as a local label, where `<N>` is an integer in the range 0 to 99 and `<name>` is an optional textual name. Local labels are limited in scope by the `ROUT` directive. To reference a local label, you refer to it as `%{|F|B}{|A|T}<N>{<name>}`. The extra prefix letters tell the assembler how to search for the label:

- If you specify `F`, the assembler searches forward; if `B`, then the assembler searches backwards. Otherwise the assembler searches backwards and then forwards.
- If you specify `T`, the assembler searches the current macro only; if `A`, then the assembler searches all macro levels. Otherwise the assembler searches the current and higher macro nesting levels.

ARM Assembler Expressions

The ARM assembler can evaluate a number of numeric, string, and logical expressions at assembly time. Table B1.15 shows some of the unary and binary operators you can use within expressions. Brackets can be used to change the order of evaluation in the usual way.

TABLE B1.15: ARM assembler unary and binary operators.

Expression	Result	Example
`A+B, A-B`	A plus or minus B	1-2 = 0xffffffff
`A*B, A/B`	A multiplied by or divided by B	2*3 = 6, 7/3 = 2
`A:MOD:B`	A modulo B	7:`MOD`:3 = 1
`:CHR:A`	string with `ASCII` code A	:`CHR`:32 = " "
`'X'`	the ASCII value of X	'a' = 0x61
`:STR:A, :STR:L`	A or L converted to a string	:`STR`:32 = "00000020" : `STR`:{TRUE} = "T"
`A<<B, A:SHL:B`	A shifted left by B bits	1 << 3 = 8
`A>>B, A:SHR:B`	A shifted right by B bits (logical shift)	0x80000000 >> 4 = 0x08000000
`A:ROR:B, A:ROL:B`	A rotated right/left by B bits	1:ROR:1 = 0x80000000 0x80000000:ROL:1 = 1

A=B, A>B, A>=B, A<B, A<=B, A/=B, A<>B	comparison of arithmetic or string variables (/= and <> both mean not equal)	(1=2) = {FALSE}, (1<2) = {TRUE}, ("a"="c") = {FALSE}, ("a"<"c") = {TRUE}
A: AND: B, A: OR: B, A: EOR: B, :NOT:A	Bitwise AND, OR, exclusive OR of A and B; bitwise NOT of A.	1:AND:3 = 1 1:OR:3 = 3:NOT:0 = 0xFFFFFFFF
:LEN:S	length of the string S	:LEN:"ABC" = 3
S:LEFT:B, S:RIGHT:B	leftmost or rightmost B characters of S	"ABC":LEFT:2 = "AB", "ABC":RIGHT:2 = "BC"
S:CC:T	the concatenation of S, T	"AB":CC:"C" = "ABC"
L:LAND:M, L:LOR:M, L:LEOR:M	logical AND, OR, exclusive OR of L and M	{TRUE}:LAND:{FALSE} = {FALSE}
:DEF:X	returns TRUE if a variable called X is defined	
:BASE:A :INDEX:A	see the MAP directive	

TABLE B1.16 Predefined expressions.

Variable	Value
{ARCHITECURE}	The ARM architecture of the CPU ("4T" for ARMv4T)
{ARMASM_VERSION}	The assembler version number
{CONFIG} or {CODESIZE}	The bit width of the instructions being assembled (32 for ARM state, 16 for Thumb state)
{CPU}	The name of the CPU being assembled for
{ENDIAN}	The configured endianness, "big'' or "little"
{INTER}	{TRUE} if ARM/Thumb interworking is on
{PC} (alias .)	The address of the current instruction being assembled
{ROPI}, {RWPI}	{TRUE} if read-only/read-write position independent
{VAR} (alias @)	The MAP counter (see the MAP directive)

In Table B1.15, A and B represent arbitrary integers; S and T, strings; and L and M, logical values. You can use labels and other symbols in place of integers in many expressions.

Predefined Variables

Table B1.16 shows a number of special variables that can appear in expressions. These are predefined by the assembler, and you cannot override them.

ARM Assembler Directives

Here is an alphabetical list of the more common *armasm* directives.

ALIGN

```
ALIGN   {<expression>, {<offset>}}
```

Aligns the address of the next instruction to the form q*<expression>+<offset>. The alignment is relative to the start of the ELF section so this must be aligned appropriately (see the AREA directive). <expression> must be a power of two; the default is 4. <offset> is zero if not specified.

AREA

```
AREA    <section> {,<attr_1>} {,<attr_2>} ... {,<attr_k>}
```

Starts a new code or data section of name <section>. Table B1.17 lists the possible attributes.

TABLE B1.17 AREA attributes.

Attribute	Meaning
ALIGN=<expression>	Align the ELF section to a $2^{expression}$ byte boundary.
ASSOC=<sectionname>	If this section is linked, also link <sectionname>.
CODE	The section contains instructions and is read only.
DATA	The section contains data and is read write.
NOINIT	The data section does not require initialization.
READONLY	The section is read only.
READWRITE	The section is read write.

ASSERT

```
ASSERT  <logical-expression>
```

Assemble time assert. If the logical expression is false, then assembly terminates with an error.

CN

```
<name>  CN      <numeric-expression>

Set <name> to be an alias for coprocessor register <numeric-expression>.
```

CODE16, CODE32

CODE16 tells the assembler to assemble the following instructions as 16-bit Thumb instructions. CODE32 indicates 32-bit ARM instructions (the default for *armasm*).

CP

```
<name>  CP      <numeric-expression>
```

Set <name> to be an alias for coprocessor number <numeric-expression>.

DATA

```
<label> DATA
```

The DATA directive indicates that the label points to data rather than code. In Thumb mode this prevents the linker from setting the bottom bit of the label. Bit 0 of a function pointer or code label is 0 for ARM code and 1 for Thumb code (see the BX instruction).

DCB, DCD{U}, DCI, DCQ{U}, DCW{U}

These directives allocate one or more bytes of initialized memory according to Table B1.18. Follow each directive with a comma-separated list of initialization values. If you specify the optional U suffix, then the assembler does not insert any alignment padding.

Examples

```
hello   DCB "hello", 0
powers  DCD 1, 2, 4, 8, 10, 0x20, 0x40, 0x80
        DCI 0xEA000000
```

TABLE B1.18 Memory initialization directives.

Directive	Alias	Data size (bytes)	Initialization value
DCB	=	1	byte or string
DCW		2	16-bit integer (aligned to 2 bytes)
DCD	&	4	32-bit integer (aligned to 4 bytes)
DCQ		8	64-bit integer (aligned to 4 bytes)
DCI		2 or 4	integer defining an ARM or Thumb instruction

ELSE (alias |)

See IF.

END

This directive must appear at the end of a source file. Assembler source after an END directive is ignored.

ENDFUNC (alias ENDP), ENDIF (alias])

See FUNCTION and IF, respectively.

ENTRY

This directive specifies the program entry point for the linker. The entry point is usually contained in the ARM C library.

EQU (alias *)

```
<name>  EQU <numeric-expression>
```

This directive is similar to #define in C. It defines a symbol <name> with value defined by the expression. This value cannot be redefined. See Section *ARM Assembler Variables* for the use of redefinable variables.

EXPORT (alias GLOBAL)

```
EXPORT  <symbol>{[WEAK]}
```

Assembler symbols are local to the object file unless exported using this command. You can link exported symbols with other object and library files. The optional [WEAK] suffix indicates that the linker should try and resolve references with other instances of this symbol before using this instance.

EXTERN, IMPORT

```
EXTERN  <symbol>{[WEAK]}
IMPORT  <symbol>{[WEAK]}
```

Both of these directives declare the name of an external symbol, defined in another object file or library. If you use this symbol, then the linker will resolve it at link time. For IMPORT, the symbol will be resolved even if you don't use it. For EXTERN, only used symbols are resolved. If you declare the symbol as [WEAK], then no error is generated if the linker cannot resolve the symbol; instead the symbol takes the value 0.

FIELD (alias #)

See MAP.

FUNCTION (alias PROC) and ENDFUNC (alias ENDP)

The FUNCTION and ENDFUNC directives mark the start and end of an ATPCS-compliant function. Their main use is to improve the debug view and allow backtracking of function calls during debugging. They also allow the profiler to

more accurately profile assembly functions. You must precede the function directive with the ATPCS function name. For example:

```
sub  FUNCTION
     SUB     r0, r0, r1
     MOV     pc, lr
     ENDFUNC
```

GBLA, GBLL, GBLS

Directives defining global arithmetic, logic, and string variables, respectively. See Section *ARM Assembler Variables*.

GET

See INCLUDE.

GLOBAL

See EXPORT.

IF (alias [), ELSE (alias |), ENDIF (alias])

These directives provide for conditional assembly. They are similar to #if, #else, #endif, available in C. The IF directive is followed by a logical expression. The ELSE directive may be omitted. For example:

```
IF ARCHITECTURE="5TE"
  SMULBB r0, r1, r1
ELSE
  MUL    r0, r1, r1
ENDIF
```

IMPORT

See EXTERN.

INCBIN

```
INCBIN <filename>
```

This directive includes the raw data contained in the binary file <filename> at the current point in the assembly. For example, INCBIN table.dat.

INCLUDE (alias GET)

```
INCLUDE <filename>
```

Use this directive to include another assembly file. It is similar to the *#include* command in C. For example, INCLUDE header.h.

INFO (alias !)

```
INFO    <numeric_expression>, <string_expression>
```

If *<numeric_expresssion>* is nonzero, then assembly terminates with error *<string_expresssion>*. Otherwise the assembler prints *<string_expression>* as an information message.

KEEP

```
KEEP    {<symbol>}
```

By default the assembler does not include local symbols in the object file, only exported symbols (see EXPORT). Use KEEP to include all local symbols or a specified local symbol. This aids the debug view.

LCLA, LCLL, LCLS

These directives declare macro-local arithmetic, logical, and string variables, respectively. See Section *ARM Assembler Variables*.

LTORG

Use LTORG to insert a literal pool. The assembler uses literal pools to store the constants appearing in the *LDR Rd,=<value>* instruction. See LDR format 19. Usually the assembler inserts literal pools automatically, at the end of each area. However, if an area is too large, then the LDR instruction cannot reach this literal pool using *pc*-relative addressing. Then you need to insert a literal pool manually, near the LDR instruction.

MACRO, MEXIT, MEND

Use these directives to declare a new assembler macro or pseudoinstruction. The syntax is

```
            MACRO
{$<arg_0>}  <macro_name> {$<arg_1>} {,$<arg_2>} ... {,$<arg_k>}
            <macro_code>
            MEND
```

The macro parameters are stored in the dummy variables $<arg_i>. This argument is set to the empty string if you don't supply a parameter when calling the macro. The MEXIT directive terminates the macro early and is usually used inside IF statements. For example, the following macro defines a new pseudoinstruction SMUL, which evaluates to a SMULBB on an ARMv5TE processor, and an MUL otherwise.

```
        MACRO
$label  SMUL    $a, $b, $c
        IF {ARCHITECTURE}="5TE"
$label  SMULBB $a, $b, $c
        MEXIT
        ENDIF
$label  MUL     $a, $b, $c
        MEND
```

MAP (alias ^), FIELD (alias #)

These directives define objects similar to C structures. `MAP` sets the base address or offset of a structure, and `FIELD` defines structure elements. The syntax is

```
        MAP      <base> {, <base_register>}
<name>  FIELD   <field_size_in_bytes>
```

The `MAP` directive sets the value of the special assembler variable {`VAR`} to the base address of the structure. This is either the value `<base>` or the register relative value `<base_register>+<base>`. Each `FIELD` directive sets `<name>` to the value `VAR` and increments `VAR` by the specified number of bytes. For register relative values, the expressions :`INDEX`:`<name>` and :`BASE`:`<name>` return the element offset from base register, and base register number, respectively.

In practice the base register form is not that useful. Instead you can use the plain form and mention the base register explicitly in the instruction. This allows you to point to a structure of the same type with different base registers. The following example sets up a structure on the stack of two `int` variables:

```
        MAP     0               ; structure elements offset from 0
count   FIELD   4               ; define an int called count
type    FIELD   4               ; define an int called type
size    FIELD   0               ; record the struct size

        SUB     sp, sp, #size   ; make room on the stack
        MOV     r0, #0
        STR     r0, [sp, #count] ; clear the count element
        STR     r0, [sp, #type]  ; clear the type element
```

NOFP

This directive bans the use of floating-point instructions in the assembly file. We don't cover floating-point instructions and directives in this appendix.

OPT

The OPT directive controls the formatting of the *armasm -list* option. This is seldom used now that source-level debugging is available. See the *armasm* documentation.

PROC

See FUNCTION.

RLIST, RN

```
<name>    RN <numeric expression>
<name>    RLIST <list of ARM register enclosed in {}>
```

These directives name a list of ARM registers or a single ARM register. For example, the following code names *r0* as arg and the ATPCS preserved registers as saved.

```
arg       RN 0
saved     RLIST {r4-r11}
```

ROUT

The ROUT directive defines a new local label area. See Section *ARM Assembler Labels*.

SETA, SETL, SETS

These directives set the values of arithmetic, logical, and string variables, respectively. See Section *ARM Assembler Variables*.

SPACE (alias %)

```
{<label>}  SPACE      <numeric_expression>
```

This directive reserves <numeric_expression> bytes of space. The bytes are zero initialized.

WHILE, WEND

These directives supply an assemble-time looping structure. WHILE is followed by a logical expression. While this expression is true, the assembler repeats the code between WHILE and WEND. The following example shows how to create an array of powers of two from 1 to 65,536.

```
        GBLA    count
count   SETA    1
        WHILE   count<=65536
        DCD     count
count   SETA    2*count
        WEND
```

B1.5 GNU Assembler Quick Reference

This section summarizes the more useful commands and expressions available with the GNU assembler, *gas*, when you target this assembler for ARM. Each assembly line has the format

```
{<label>:} {<instruction or directive>} @ comment
```

Unlike the ARM assembler, you needn't indent instructions and directives. Labels are recognized by the following colon rather than their position at the start of the line. The following example shows a simple assembly file defining a function *add* that returns the sum of the two input arguments:

```
.section    .text, "x"

.global     add                 @ give the symbol add external linkage

add:
            ADD     r0, r0, r1  @ add input arguments
            MOV     pc, lr      @ return from subroutine
```

GNU Assembler Directives

Here is an alphabetical list of the more common *gas* directives.

```
.ascii "<string>"
```

Inserts the string as data into the assembly, as for `DCB` in *armasm*.

```
.asciz "<string>"
```

As for `.ascii` but follows the string with a zero byte.

```
.balign <power_of_2> {,<fill_value> {,<max_padding>} }
```

Aligns the address to <*power_of_2*> bytes. The assembler aligns by adding bytes of value <*fill_value*> or a suitable default. The alignment will not occur if more than <*max_padding*> fill bytes are required. Similar to `ALIGN` in *armasm*.

```
.byte <byte1> {,<byte2>} ...
```

Inserts a list of byte values as data into the assembly, as for `DCB` in *armasm*.

```
.code <number_of_bits>
```

Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly. Similar to `CODE16` and `CODE32` in *armasm*.

```
.else
```

Use with .if and .endif. Similar to `ELSE` in *armasm*.

```
.end
```

Marks the end of the assembly file. This is usually omitted.

```
.endif
```

Ends a conditional compilation code block. See `.if, .ifdef, .ifndef`. Similar to `ENDIF` in *armasm*.

```
.endm
```

Ends a macro definition. See `.macro`. Similar to `MEND` in *armasm*.

```
.endr
```

Ends a repeat loop. See `.rept` and `.irp`. Similar to `WEND` in *armasm*.

```
.equ <symbol name>, <value>
```

This directive sets the value of a symbol. It is similar to `EQU` in *armasm*.

```
.err
```

Causes assembly to halt with an error.

```
.exitm
```

Exit a macro partway through. See `.macro`. Similar to `MEXIT` in *armasm*.

```
.global <symbol>
```

This directive gives the symbol external linkage. It is similar to `EXPORT` in *armasm*.

```
.hword <short1> {,<short2>} ...
```

Inserts a list of 16-bit values as data into the assembly, as for `DCW` in *armasm*.

```
.if <logical_expression>
```

Makes a block of code conditional. `End` the block using .endif. Similar to `IF` in *armasm*. See also `.else`.

```
.ifdef <symbol>
```

Include a block of code if `<symbol>` is defined. End the block with `.endif`.

```
.ifndef <symbol>
```

Include a block of code if `<symbol>` is not defined. End the block with `.endif`.

```
.include "<filename>"
```

Includes the indicated source file. Similar to `INCLUDE` in *armasm* or `# include` in C.

```
.irp <param> {,<val_1>} {,<val_2>} ...
```

Repeats a block of code, once for each value in the value list. Mark the end of the block using a `.endr` directive. In the repeated code block, use `\<param>` to substitute the associated value in the value list.

```
.macro <name> {<arg_1>} {,<arg_1>} ... {,<arg_k>}
```

Defines an assembler macro called <*name*> with *k* parameters. The macro definition must end with `.endm`. To escape from the macro at an earlier point, use `.exitm`. These directives are similar to `MACRO`, `MEND`, and `MEXIT` in *armasm*. You must precede the dummy macro parameters by `\`. For example:

```
.macro SHIFTLEFT a, b
        .if \b < 0
          MOV \a, \a, ASR #-\b
          .exitm
         .endif
         MOV \a, \a, LSL #\b
.endm
```

```
.rept <number_of_times>
```

Repeats a block of code the given number of times. End the block with `.endr`.

```
<register_name> .req <register_name>
```

This directive names a register. It is similar to the RN directive in *armasm* except that you must supply a name rather than a number on the right. For example, *acc .req r0.*

```
.section <section_name> {,"<flags>"}
```

Starts a new code or data section. Usually you should call a code section .text, an initialized data section .data, and an uninitialized data section .bss . These have default flags, and the linker understands these default names. The directive is similar to the *armasm* directive AREA. Table B1.19 lists possible characters to appear in the <flags> string for ELF format files.

```
.set <variable_name>, <variable_value>
```

TABLE B1.19 .section flags for ELF format files.

Flag	Meaning
a	allocatable section
w	writable section
x	executable section

This directive sets the value of a variable. It is similar to SETA in *armasm.*

```
.space <number_of_bytes> {,<fill_byte>}
```

Reserves the given number of bytes. The bytes are filled with zero or <fill_byte> if specified. It is similar to SPACE in *armasm.*

```
.word <word1> {,<word2>} ...
```

Inserts a list of 32-bit word values as data into the assembly, as for DCD in *armasm.*