Appendix A
Instruction Summary

A summary of the instructions available in ARM/Thumb assembly language is given in this Appendix.

For most instructions, additional explanation can be found in Chapter 5. The optional condition code field (denoted by `cond`) is described in *Conditional execution* on page 5-3. The format of the flexible `operand2` used by data processing operations is described in *Operand 2 and the barrel shifter* on page 5-7, while the various addressing mode options for loads and stores is given in *Addressing modes* on page 5-13.

This Appendix is intended for quick reference. If more detail about the precise operation of an instruction is required, refer to the *ARM Architecture Reference Manual*, or to the official ARM documentation (for example the *ARM Compiler Toolchain Assembler Reference*) that can be found at `http://infocenter.arm.com/help/index.jsp`.

A.1 Instruction Summary

Instructions are listed in alphabetic order, with a description of the syntax, operands and behavior of the instruction. Not all usage restrictions are documented here, nor do we show the associated binary encoding or any detail of changes associated with older architecture versions.

A.1.1 ADC

`ADC` (Add with Carry) adds together the values in `Rn` and `Operand2`, with the carry flag.

Syntax

```
ADC{S}{cond} {Rd}, Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.2 ADD

`ADD` adds together the values in `Rn` and `Operand2` (or `Rn` and `imm12`).

Syntax

```
ADD{S}{cond} {Rd,} Rn, <Operand2>
```

```
ADD{cond} {Rd,} Rn, #imm12
```
(Only available in Thumb)

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

`imm12` is in the range 0-4095.

A.1.3 ADR

`ADR` (Address) is an instruction that loads a program or register-relative address (short range). It generates an instruction that adds or subtracts a value to the PC (in the PC-relative case). Alternatively, it can be some other register for a label defined as an offset from a base register defined with the `MAP` directive (see the ARM tools documentation for more detail).

Syntax

```
ADR{cond} Rd, label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`label` is a PC or register-relative expression.

A.1.4 ADRL

`ADRL` (Address) is a pseudo-instruction that loads a program or register-relative address (long range). It always generates two instructions.

Syntax

```
ADRL{cond} Rd, label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`label` is a PC-relative expression. The offset between label and the current location has some restrictions.

The `ADRL` pseudo-instruction can generate a wider range of addresses than `ADR`.

A.1.5 AND

`AND` does a bitwise AND on the values in `Rn` and `Operand2`.

Syntax

```
AND{S}{cond} {Rd,} Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter*.

A.1.6 ASR

`ASR` (Arithmetic Shift Right) shifts the value in `Rm` right, by the number of bit positions specified and copies the sign bit into vacated bit positions on the left. Permitted shift values are in the range 1-32. It can be considered as giving the signed value of a register divided by a power of two.

Syntax

```
ASR{S}{cond} {Rd,} Rm, Rs
ASR{S}{cond} {Rd,} Rm, imm
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand to be shifted.

`Rs` is the register that holds a shift value to apply to the value in `Rm`. Only the least significant byte of the register is used.

`imm` is a shift amount, in the range 1-32.

A.1.7 B

`B` (Branch) transfers program execution to the address specified by label.

Syntax

```
B{cond}{.W} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`label` is a PC-relative expression.

`.W` is an optional instruction width specifier to force the use of a 32-bit instruction in Thumb.

A.1.8 BFC

`BFC` (Bit Field Clear) clears bits in a register. A number of bits specified by width are cleared in `Rd`, starting at `lsb`. Other bits in `Rd` are unchanged.

Syntax

```
BFC{cond} Rd, #lsb, #width
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3

`Rd` is the destination register.

`lsb` specifies the least significant bit to be cleared.

`width` is the number of bits to be cleared.

A.1.9 BFI

`BFI` (Bit Field Insert) copies bits into a register. A number of bits in `Rd` specified by width, starting at `lsb`, are replaced by bits from `Rn`, starting at bit[0]. Other bits in `Rd` are unchanged.

Syntax

```
BFI{cond} Rd, Rn, #lsb, #width
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register that contains the bits to be copied.

`lsb` specifies the least significant bit in `Rd` to be written to.

`width` is the number of bits to be copied.

A.1.10 BIC

`BIC` (bit clear) does an AND operation on the bits in `Rn`, with the complements of the corresponding bits in the value of `Operand2`.

Syntax

```
BIC{S}{cond} {Rd}, Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.11 BKPT

`BKPT` (Breakpoint) causes the core to enter Debug state.

Syntax

```
BKPT #imm
```

where:

`imm` is an integer in the range 0 – 65535 (ARM) or 0 – 255 (Thumb). This integer is not used by the core itself, but can be used by Debug tools.

A.1.12 BL

`BL` (Branch with Link) transfers program execution to the address specified by label and stores the address of the next instruction in the LR (R14) register.

Syntax

```
BL{cond} label
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

label is a PC-relative expression.

A.1.13 BLX

BLX (Branch with Link and eXchange) transfers program execution to the address specified by label and stores the address of the next instruction in the LR (R14) register. BLX can change the core state from ARM to Thumb, or from Thumb to ARM. BLX label always changes the core state from Thumb to ARM, or ARM to Thumb. BLX Rm will set the state based on bit[0] of Rm:

- Rm bit[0]=0 ARM state.
- Rm bit[0]=1 Thumb state.

Syntax

```
BLX{cond} label
BLX{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

label is a PC-relative expression.

Rm is a register that holds the address to branch to.

A.1.14 BX

BX (Branch and eXchange) transfers program execution to the address specified in a register. BX can change the core state from ARM to Thumb, or from Thumb to ARM. BX Rm will set the state based on bit[0] of Rm:

- Rm bit[0] = 0 ARM state.
- Rm bit[0] = 1 Thumb state.

Syntax

```
BX{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rm is a register that holds the address to branch to.

A.1.15 BXJ

BXJ (Branch and eXchange Jazelle) enter Jazelle State or perform a BX branch and exchange to the address contained in Rm.

Syntax

```
BXJ{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rm is a register that holds the address to branch to if entry to Jazelle fails.

A.1.16 CBNZ

CBNZ (Compare and Branch if Nonzero) causes a branch if the value in Rn is not zero. It does not change the PSR flags. There is no ARM or 32-bit Thumb versions of this instruction.

Syntax

```
CBNZ Rn, label
```

where:

label is a pc-relative expression.

Rn is a register that holds the operand.

A.1.17 CBZ

CBZ (Compare and Branch if Zero) causes a branch if the value in Rn is zero. It does not change the PSR flags. There is no ARM or 32-bit Thumb versions of this instruction.

Syntax

```
CBZ Rn, label
```

where:

label is a PC-relative expression.

Rn is a register that holds the operand.

A.1.18 CDP

CDP (Coprocessor Data Processing operation) performs a coprocessor operation. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
CDP{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRd, CRn, CRm are coprocessor registers.

A.1.19 CDP2

CDP2 (Coprocessor Data Processing operation) performs a coprocessor operation. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
CDP2{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}
```

where:

`cond` is an optional condition code See *Conditional execution* on page 5-3.

`coproc` is the name of the coprocessor the instruction is for. This is usually of the form `pn`, where `n` is an integer in the range 0 to 15.

`opcode1` is a 4-bit coprocessor-specific opcode.

`opcode2` is an optional 3-bit coprocessor-specific opcode.

`CRd`, `CRn`, `CRm` are coprocessor registers.

A.1.20 CHKA

`CHKA` (Check array) is a ThumbEE instruction. If the value in the first register is less than or equal to, the second, the IndexCheck handler is called. This instruction is only available in 16-bit ThumbEE and only when Thumb-2EE support is present.

Syntax

```
CHKA Rn, Rm
```

where:

`Rn` holds the size of the array.

`Rm` contains the array index.

A.1.21 CLREX

`CLREX` (Clear Exclusive) moves a local exclusive access monitor to its open-access state.

Syntax

```
CLREX{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.22 CLZ

`CLZ` (Count Leading Zeros) counts the number of leading zeros in the value in `Rm` and returns the result in `Rd`. The result returned is 32 if no bits are set in `Rm`, or 0 if bit [31] is set.

Syntax

```
CLZ{cond} Rd, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand.

A.1.23 CMN

CMN (Compare Negative) performs a comparison by adding the value of Operand2 to the value in Rn. The condition flags are changed, based on the result, but the result itself is discarded.

Syntax

```
CMN{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.24 CMP

CMP (Compare) performs a comparison by subtracting the value of Operand2 from the value in Rn. The condition flags are changed, based on the result, but the result itself is discarded.

Syntax

```
CMP{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.25 CPS

CPS (Change Processor State) can be used to change the processor mode or to enable or disable individual exception types.

Syntax

```
CPS #mode
CPSIE iflags{, #mode}
CPSID iflags{, #mode}
```

where:

mode is the number of a mode for the processor to enter.

IE Interrupt or Abort Enable.

ID Interrupt or Abort Disable.

iflags specifies one or more of:

- a = asynchronous abort.
- i = IRQ.
- f = FIQ.

A.1.26 DBG

DBG (Debug) is a hint operation, treated as a NOP by the processor, but can provide information to debug systems.

Syntax

```
DBG{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is in the range 0-15.

A.1.27 DMB

DMB (Data Memory Barrier) requires that all explicit memory accesses in program order before the DMB instruction are observed before any explicit memory accesses in program order after the DMB instruction. See Chapter 10 for a detailed description.

Syntax

```
DMB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is covered in depth in Chapter 10.

A.1.28 DSB

DSB (Data Synchronization Barrier) requires that no further instruction executes until all explicit memory accesses, cache maintenance operations, branch prediction and TLB maintenance operations before this instruction complete. See Chapter 10 for a detailed description.

Syntax

```
DSB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is covered in depth in Chapter 10.

A.1.29 ENTERX

ENTERX causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state. It is not available in the ARM instruction set.

Syntax

```
ENTERX
```

A.1.30 EOR

EOR performs an Exclusive OR operation on the values in Rn and Operand2.

Syntax

```
EOR{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.31 ERET

ERET (Exception Return) loads the PC from the ELR-hyp and loads the CPSR from SPSR-hyp when executed in Hyp mode.

When executed in a Secure or Non-Secure PLI mode, ERET behaves as:

- MOVS PC, LR in the ARM instruction set.
- SUBS PC, LR, #0 in the Thumb instruction set.

Syntax

```
ERET{cond} {q}
```

where:

cond is the optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

A.1.32 HB

HB (Handler Branch) branches to a specified handler (available in ThumbEE only).

Syntax

```
HB{L} #HandlerID
HB{L}P #imm, #HandlerID
```

where:

L indicates that the instruction saves a return address in the LR.

P means that the instruction passes the value of imm to the handler in R8.

imm is an immediate value in the range 0-31 (if L is present), otherwise in the range 0-7.

HandlerID is the index number of the handler to be called, in the range 0-31 (if P is specified), otherwise in the range 0-255.

A.1.33 ISB

ISB (Instruction Synchronization Barrier) flushes the processor pipeline and ensures that context altering operations (such as ASID or other CP15 changes, branch prediction or TLB maintenance activity) *before* the ISB, are visible to the instructions fetched *after* the ISB.

See Chapter 10 for a detailed description of barriers.

Syntax

```
ISB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option can be SY (full system), which is the default and so can be omitted.

A.1.34 IT

IT (If-then) makes up to four following instructions conditional (known as the IT block). The conditions can all be the same, or some can be the logical inverse of others. IT is a pseudo-instruction in ARM state.

Syntax

```
IT{x{y{z}}} {cond}
```

where:

cond is a condition code. See *Conditional execution* on page 5-3 that specifies the condition for the first instruction in the IT block.

x, y and z specify the condition switch for the second, third and fourth instructions in the IT block, for example, ITTET.

The condition switch can be either:

- T (Then) applies the condition cond to the instruction.
- E (Else) applies the inverse condition of cond to the instruction.

A.1.35 LDA

LDA (Load-Acquire Word) loads a word from memory and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDA imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDA{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rt` is the destination register.

`Rn` is the base register.

A.1.36 LDAB

`LDAB` (Load-Acquire Byte) loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`LDAB` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAB{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register.

`Rm` is the base register.

A.1.37 LDAEX

`LDAEX` (Load-Acquire Exclusive Word) loads a word from memory and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEX imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEX{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the destination register.

Rn is the base register.

A.1.38 LDAEXB

LDAEXB (Load-Acquire Exclusive Byte) loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXB{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the destination register.

Rm is the base register.

A.1.39 LDAEXD

LDAEXD (Load-Acquire Exclusive Double) loads a doubleword from memory and writes it to two registers. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXD imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXD{cond}{q} <Rt>, <Rt2>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the first destination register. It must be an even numbered register and not R14..

Rt2 is the second destination register. Rt2 must be R(t + 1).

Rn is the base register.

A.1.40 LDAEXH

LDAEXH (Load-Acquire Exclusive Halfword) loads a halfword from memory, zero-extends it to form a 32-bit word and writes ito a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXH imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXH{cond} <Rt,> [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.

- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rt` is the destination register.

`Rn` is the base register.

A.1.41 LDAH

`LDAH` (Load-Acquire Halfword) loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`LDAH` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAH{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register.

`Rm` is the base register.

A.1.42 LDC

`LDC` (Load Coprocessor Registers) reads a coprocessor register from memory (or multiple registers, if `L` is specified).

Syntax

```
LDC{L}{cond} coproc, CRd, [Rn]
LDC{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
LDC{L}{cond} coproc, CRd, [Rn], #{-}offset
LDC{L}{cond} coproc, CRd, label
```

where:

`L` specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

Offset is a multiple of 4, in the range 0-1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.43 LDC2

LDC2 (Load Coprocessor Registers) reads a coprocessor register from memory (or multiple registers, if L is specified).

Syntax

```
LDC2{L}{cond} coproc, CRd, [Rn]
LDC2{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
LDC2{L}{cond} coproc, CRd, [Rn], #{-}offset
LDC2{L}{cond} coproc, CRd, label
```

where:

L specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

Offset is a multiple of 4, in the range 0 – 1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.44 LDM

LDM (Load Multiple registers) loads one or more registers from consecutive addresses in memory at an address specified in a base register.

Syntax

```
LDM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).

- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA). For example LDMFD is a synonym of LDMDB.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the base register, giving the initial address for the transfer.

! if present, specifies that the final address is written back into Rn.

Reglist is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

^ if specified (in a mode other than User or System) means one of two possible special actions will be taken:

- Data is transferred into the User mode registers instead of the current mode registers (in the case where Reglist does not contain the PC).
- If Reglist does contain the PC, the normal multiple register transfer happens and the SPSR is copied into the CPSR. This is used for returning from exception handlers.

A.1.45 LDR

LDR (Load Register) loads a value from memory to an ARM register, optionally updating the register used to give the address.

A variety of addressing options are provided. For full details of the available addressing modes, see *Addressing modes* on page 5-13.

Syntax

```
LDR{type}{T}{cond} Rt, [Rn {, #offset}]
LDR{type}{cond} Rt, [Rn, #offset]!
LDR{type}{T}{cond} Rt, [Rn], #offset
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]!
LDR{type}{T}{cond} Rt, [Rn], +/-Rm {, shift}
```

where:

type can be any one of:

- B – unsigned Byte. (Zero extend to 32 bits on loads.)
- SB – signed Byte. (Sign extend to 32 bits.)
- H – unsigned Halfword. (Zero extend to 32 bits on loads.)
- SH – signed Halfword. (Sign extend to 32 bits.)

or omitted, for a Word load.

T specifies that memory is accessed as if the processor was in User mode (not available in all addressing modes).

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the base address for the memory operation.

! if present, specifies that the final address is written back into Rn.

offset is a numeric value.

Rm is a register holding an offset value to be applied.

shift is either a register or immediate based shift to apply to the offset value.

A.1.46 LDR (pseudo-instruction)

LDR (Load Register) pseudo-instruction loads a register with a 32-bit immediate value or an address. It generates either a MOV or MVN instruction (if possible), or a PC-relative LDR instruction that reads the constant from the literal pool.

Syntax

```
LDR{cond}{.W} Rt, =expr
LDR{cond}{.W} Rt, label_expr
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

.W specifies that a 32-bit Thumb instruction must be used.

Rt is the register to load.

expr is a numeric value.

label_expr is a label, optionally plus or minus a numeric value.

A.1.47 LDRD

LDRD (Load Register Dual) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers.

Syntax

```
LDRD{cond} Rt, Rt2, [{Rn},+/-{Rm}]{!}
LDRD{cond} Rt, Rt2, [{Rn}],+/-{Rm}
```

where:

cond is an optional condition code. See See *Conditional execution* on page 5-3.

Rt is the first destination register. This register must be even-numbered and not R14.

Rt is the second destination register. This register must be <R(t+1)>.

Rn is the base register. The SP or the PC can be used.

+/- is + or omitted if the value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (add == FALSE).

Rm contains the offset that is applied to the value of <Rn> to form the address.

A.1.48 LDREX

LDREX (Load register exclusive). Performs a load from a location and marks it for exclusive access. Byte, halfword, word and doubleword variants are provided.

Syntax

```
LDREX{cond} Rt, [Rn {, #offset}]
LDREXB{cond} Rt, [Rn]
```

```
LDREXH{cond} Rt, [Rn]
LDREXD{cond} Rt, Rt2, [Rn]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rt` is the register to load.

`Rt2` is the second register for doubleword loads.

`Rn` is the register holding the address.

`offset` is an optional value, permitted in Thumb only.

A.1.49 LEAVEX

`LEAVEX` causes a change from ThumbEE state to Thumb state, or has no effect in Thumb state. It is not available in the ARM instruction set.

Syntax

```
LEAVEX
```

A.1.50 LSL

`LSL` (Logical Shift Left) shifts the value in `Rm` left by the specified number of bits, inserting zeros into the vacated bit positions.

Syntax

```
LSL{S}{cond} Rd, Rm, Rs
LSL{S}{cond} Rd, Rm, imm
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand to be shifted.

`Rs` is the register that holds a shift value to apply to the value in `Rm`. Only the least significant byte of the register is used.

imm is a shift amount, in the range 0-31.

A.1.51 LSR

`LSR` (Logical Shift Right) shifts the value in `Rm` right by the specified number of bits, inserting zeros into the vacated bit positions.

Syntax

```
LSR{S}{cond} Rd, Rm, Rs
LSR{S}{cond} Rd, Rm, imm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register holding the operand to be shifted.

Rs is the register that holds a shift value to apply to the value in Rm. Only the least significant byte of the register is used.

imm is a shift amount, in the range 1-32.

A.1.52 MCR

MCR (Move to Coprocessor from Register) writes a coprocessor register, from an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCR{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.53 MCR2

MCR2 (Move to Coprocessor from Register) writes a coprocessor register, from an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCR2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.54 MCRR

MCRR (Move to Coprocessor from Registers) transfers a pair of ARM register to a coprocessor. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCRR{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.55 MCRR2

MCRR2 (Move to Coprocessor from Registers) transfers a pair of ARM register to a coprocessor. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCRR2{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.56 MLA

MLA (Multiply Accumulate) multiplies Rn and Rm, adds the value from Ra, and stores the least significant 32 bits of the result in Rd.

Syntax

```
MLA{S}{cond} Rd, Rn, Rm, Ra
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.57 MLS

MLS (Multiply and Subtract) multiplies Rn and Rm, subtracts the result from Ra, and stores the least significant 32 bits of the final result in Rd.

Syntax

```
MLS{S}{cond} Rd, Rn, Rm, Ra
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.58 MOV

MOV (Move) copies the value of Operand2 into Rd.

Syntax

```
MOV{S}{cond} Rn, <Operand2>
MOV{cond} Rd, #imm16
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See Section 6.2.1.

imm16 is an immediate value in the range 0-65535.

A.1.59 MOVT

MOVT (Move Top) writes imm16 to Rd[31:16]. It does not affect Rd[15:0].

Syntax

```
MOVT{cond} Rd, #imm16
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See Section 6.2.1.

imm16 is an immediate value in the range 0-65535.

A.1.60 MOV32

MOV32 is a pseudo-instruction that loads a register with a 32-bit immediate value or address. It generates two instructions, a MOV, MOVT pair.

Syntax

```
MOV32 Rd, expr
```

where:

Rd is the destination register.

expr is a 32-bit constant, or address label.

A.1.61 MRC

MRC (Move to Register from Coprocessor) reads a coprocessor register to an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MRC{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.62 MRC2

MRC2 (Move to Register from Coprocessor) reads a coprocessor register to an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MRC2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.63 MRRC

MRRC (Move to Registers from Coprocessor) transfers a value from a Coprocessor to a pair of ARM registers. The purpose of this instruction is defined by the Coprocessor implementer.

Syntax

```
MRRC{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3. MRRC instructions might not specify a condition code in ARM state.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.64 MRRC2

MRRC2 (Move to Registers from Coprocessor) transfers a value from a Coprocessor to a pair of ARM registers. The purpose of this instruction is defined by the Coprocessor implementer.

Syntax

```
MRRC2{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3. MRRC2 instructions might not specify a condition code in ARM state.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.65 MRS

MRS (Move Status register or Coprocessor Register to General purpose register) can be used to read the CPSR/APSR, CP14 or CP15 Coprocessor registers.

Syntax

```
MRS{cond} Rd, psr
MRS{cond} Rn, coproc_register
MRS{cond} APSR_nzcv, DBGDSCRint
MRS{cond} APSR_nzcv, FPSCR
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`psr` is one of: APSR, CPSR or SPSR.

`coproc_register` is the name of a CP14 or CP15 readable register.

`DBGDSCRint` is the name of a CP14 register that can be copied to the APSR.

A.1.66 MSR

`MSR` (Move Status Register or Coprocessor Register from General Purpose Register) can be used to write all or part of the CPSR/APSR or CP14 or CP15 registers.

Use of the `MSR` instruction to set the endianness bit in the CPSR in User Mode is deprecated. ARM strongly recommends that software executing in User Mode uses the `SETEND` instruction.

Syntax

```
MSR{cond} APSR_flags, Rm
MSR{cond} coproc_register
MSR{cond} APSR_flags, #constant
MSR{cond} psr_fields, #constant
MSR{cond} psr_fields, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rm and Rn are the source registers.

`flags` can be one or more of nzcvq (ALU flags) or g (SIMD flags).

`coproc_register` is the name of a CP14 or CP15 readable register.

`constant` is an 8-bit pattern rotated by an even number of bits within a 32-bit word. (Not available in Thumb.)

`psr` is one of: APSR, CPSR or SPSR.

`fields` is one or more of:

- `c` control field mask byte, PSR[7:0]
- `x` extension field mask byte, PSR[15:8]
- `s` status field mask byte, PSR[23:16]
- `f` flags field mask byte, PSR[31:24].

A.1.67 MUL

`MUL` (Multiply) Multiplies `Rn` and `Rm`, and stores the least significant 32 bits of the result in `Rd`.

Syntax

```
MUL{S}{cond} {Rd,} Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.68 MVN

MVN (Move Not) performs a bitwise NOT operation on the operand2 value, and places the result into Rd.

Syntax

```
MVN{S}{cond} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.69 NOP

NOP (No Operation) does nothing.

Syntax

```
NOP{cond}
```

where:

NOP does not have to consume clock cycles. It can be removed by the processor pipeline. It is used for padding, to ensure following instructions align to a boundary.

A.1.70 ORN

ORN (OR NOT) performs an OR operation on the bits in Rn with the complement of the corresponding bits in the value of Operand2.

Syntax

```
ORN{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.71 ORR

Performs an OR operation on the bits in Rn with the corresponding bits in the value of Operand2.

Syntax

```
ORR{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.72 PKHBT

PKHBT (Pack Halfword Bottom Top) combines bits[15:0] of Rn with bits[31:16] of the shifted value from Rm.

Syntax

```
PKHBT{cond} {Rd,} Rn, Rm{, LSL #leftshift}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

leftshift is a number in the range 0-31.

A.1.73 PKHTB

PKHTB (Pack Halfword Top Bottom) combines bits[31:16] of Rn with bits[15:0] of the shifted value from Rm.

Syntax

```
PKHTB{cond} {Rd,} Rn, Rm {, ASR #rightshift}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Rm` is the register holding the second operand.

`rightshift` is a number in the range 1-32.

A.1.74 PLD

`PLD` (Preload Data) is a hint instruction that can cause data to be preloaded into the cache.

Syntax

```
PLD{cond} [Rn {, #offset}]
PLD{cond} [Rn, +/-Rm {, shift}]
PLD{cond} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

`label` is a PC-relative expression.

A.1.75 PLDW

`PLDW` (Preload data with intent to write) is a hint instruction that can cause data to be preloaded into the cache. It is available only in processors that implement multi-processing extensions.

Syntax

```
PLDW{cond} [Rn {, #offset}]
PLDW{cond} [Rn, +/-Rm {, shift}]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

A.1.76 PLI

`PLI` (Preload instructions) is a hint instruction that can cause instructions to be preloaded into the cache.

Syntax

```
PLI{cond} [Rn {, #offset}]
PLI{cond} [Rn, +/-Rm {, shift}]
PLI{cond} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

`label` is a PC-relative expression.

A.1.77 POP

`POP` is used to pop registers off a full descending stack. `POP` is a synonym for `LDMIA sp!, reglist`.

Syntax

```
POP{cond} reglist
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`reglist` is a list of one or more registers, enclosed in braces.

A.1.78 PUSH

`PUSH` is used to push registers on to a full descending stack. `PUSH` is a synonym for `STMDB sp!, reglist`.

Syntax

```
PUSH{cond} reglist
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`reglist` is a list of one or more registers, enclosed in braces.

A.1.79 QADD

`QADD` (Saturating signed Add) does a signed addition and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QADD{cond} {Rd,} Rm, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.80 QADD8

QADD8 (Saturating signed bytewise Add) does a signed bytewise addition (4 adds) and saturates the results to the signed range $-2^7 \leq x \leq 2^7-1$. The Q flag is not affected by this instruction.

Syntax

```
QADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.81 QADD16

QADD16 (Saturating signed bytewise Add) does a signed halfword-wise addition (2 adds) and saturates the results to the signed range $-2^7 \leq x \leq 2^7-1$. The Q flag is not affected by this instruction.

Syntax

```
QADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.82 QASX

QASX (Saturating signed Add Subtract eXchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.83 QDADD

QADD (Saturating signed Add) does a signed doubling addition and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDADD{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is multiplied by 2, saturated and then added to the value in Rm. A second saturate operation is then performed.

A.1.84 QDSUB

QDSUB (Saturating signed Doubling Subtraction) does a signed doubling subtraction and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDSUB{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is multiplied by 2, saturated and then subtracted from the value in Rm. A second saturate operation is then performed.

A.1.85 QSAX

QSAX (Saturating signed Subtract Add Exchange) exchanges the halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.86 QSUB

QSUB (Saturating signed Subtraction) does a signed subtraction and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDSUB{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is subtracted from the value in Rm. A saturate operation is then performed.

A.1.87 QSUB8

QSUB8 (Saturating signed bytewise Subtract) does bytewise subtraction (4 subtracts), with saturation of the results to the signed range $-2^{7} \leq x \leq 2^{7}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.88 QSUB16

QSUB16 (Saturating signed halfword Subtract) does halfword-wise subtraction (2 subtracts), with saturation of the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.89 RBIT

RBIT (Reverse bits) reverses the bit order in a 32-bit word.

Syntax

```
RBIT{cond} Rd, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.90 REV

`REV` (Reverse) converts 32-bit big-endian data into little-endian data, or 32-bit little-endian data into big-endian data.

Syntax

```
REV{cond} {Rd}, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.91 REV16

`REV16` (Reverse byte order halfwords) converts 16-bit big-endian data into little-endian data, or 16-bit little-endian data into big-endian data.

Syntax

```
REV16{cond} {Rd}, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.92 REVSH

`REVSH` (Reverse byte order halfword, with sign extension) does a reverse byte order of the bottom halfword, and sign extends the result to 32 bits.

Syntax

```
REVSH{cond} Rd, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

Rn is the register holding the operand.

A.1.93 RFE

RFE (Return from Exception) is used to return from an exception where the return state was saved with SRS. If ! is specified, the final address is written back into Rn.

Syntax

```
RFE{addr_mode}{cond} Rn{!}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).
- DB – Decrement address Before each transfer.

cond is an optional condition codes. See *Conditional execution* on page 5-3, and is permitted only in Thumb, using a preceding IT instruction.

Rn specifies the base register.

A.1.94 ROR

ROR (Rotate right Register) rotates a value in a register by a specified number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

Syntax

```
ROR{S}{cond} {Rd,} Rm, Rs
ROR{S}{cond} {Rd,} Rm, imm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the operand.

Rm is the register holding the operand to be shifted.

Rs is the register that holds a shift value to apply to the value in Rm. Only the least significant byte of the register is used.

imm is a shift amount, in the range 1 – 31.

A.1.95 RRX

RRX (Rotate Right with extend) performs a shift right one bit on a register value. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.

Syntax

```
RRX{S}{cond} {Rd,} Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register holding the operand to be shifted.

A.1.96 RSB

RSB (Reverse Subtract) subtracts the value in Rn from the value of Operand2. This is useful because Operand2 has more options than Operand1 (which is always a register).

Syntax

```
RSB{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.97 RSC

RSC (Reverse Subtract with Carry) subtracts Rn from Operand2. If the carry flag is clear, the result is reduced by one.

Syntax

```
RSC{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.98 SADD8

SADD8 (Signed bytewise Add) does a signed bytewise addition (4 adds).

Syntax

```
SADD8{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.99 SADD16

`SADD16` (Signed bytewise Add) does a signed halfword-wise addition (2 adds).

Syntax

```
SADD16{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.100 SASX

`SASX` (Signed Add Subtract Exchange) exchanges halfwords of `Rm`, then adds the top halfwords and subtracts the bottom halfwords.

Syntax

```
SASX{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.101 SBC

`SBC` (Subtract with Carry) subtracts the value of `Operand2` from the value in `Rn`. If the carry flag is clear, the result is reduced by one.

Syntax

```
SBC{S}{cond} {Rd,} Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.102 SBFX

SBFX (Signed Bit Field Extract) writes adjacent bits from one register into the least significant bits of a second register and sign extends to 32 bits.

Syntax

```
SBFX{cond} Rd, Rn, #lsb, #width
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register that contains the bits to be extracted.

lsb specifies the least significant bit of the bitfield.

width is the width of the bitfield.

A.1.103 SDIV

SDIV (Signed Divide). divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. This instruction is not present in all variants of the ARMv7-A architecture.

Syntax

```
SDIV{cond}{q} {Rd,} Rn, Rm
```

where:

cond is the optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N (narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W (wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding.

Rd the destination register.

Rn is the register that contains the dividend.

Rm is the register that contains the divisor.

A.1.104 SEL

SEL (Select) selects bytes from Rn or Rm, depending on the APSR GE flags.

If GE[0] is set, Rd[7:0] comes from Rn[7:0], else from Rm[7:0].

If GE[1] is set, Rd[15:8] comes from Rn[15:8], else from Rm[15:8].

If GE[2] is set, Rd[23:16] comes from Rn[23:16], else from Rm[23:16].

If GE[3] is set, Rd[31:24] comes from Rn[31:24], else from Rm[31:24].

Syntax

```
SEL{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register that contains the bits to be extracted.

Rm is the register holding the second operand.

A.1.105 SETEND

SETEND (Set endianness) selects little-endian or big-endian memory access. See *Endianness* on page 14-2 for more details.

Syntax

```
SETEND LE
SETEND BE
```

A.1.106 SEV

SEV (Send Event) causes an event to be signaled to all cores in an MPCore. See *Assembly language power instructions* on page 20-8 for more detail.

Syntax

```
SEV{cond}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

A.1.107 SHADD8

SHADD8 (Signed halving bytewise Add) does a signed bytewise addition (4 adds) and halves the results.

Syntax

```
SHADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.108 SHADD16

SHADD16 (Signed halving bytewise Add) does a signed halfword-wise addition (2 adds) and halves the results.

Syntax

```
SHADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.109 SHASX

SHASX (Signed Halving Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and halves the results.

Syntax

```
SHASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.110 SHSAX

SHSAX (Signed Halving Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and halves the results.

Syntax

```
SHSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.111 SHSUB8

SHSUB8 (Signed halving bytewise subtraction) does a signed bytewise subtraction (4 subtracts) and halves the results.

Syntax

```
SHSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands

A.1.112 SHSUB16

SHSUB16 (Signed Halving halfword-wise Subtract) does a signed halfword-wise subtraction (2 subtracts) and halves the result.

Syntax

```
SHSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.113 SMC

SMC (Secure Monitor Call) is used by the ARM Security Extensions. This instruction was formerly called SMI. See Chapter 21 *Security* for more details.

Syntax

```
SMC{cond} #imm4
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

imm4 is an immediate value in the range 0-15, which is ignored by the processor, but can be used by the SMC exception handler.

A.1.114 SMLAxy

The SMLAxy (Signed Multiply Accumulate; 32 <= 32 + 16 × 16) instruction multiplies the 16-bit signed integers from the selected halves of Rn and Rm, adds the 32-bit result to the value from Ra, and writes the result in Rd.

Syntax

```
SMLA<x><y>{cond} Rd, Rn, Rm, Ra
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.115 SMLAD

SMLAD (Dual Signed Multiply Accumulate; 32 <= 32 + 16 × 16 + 16 × 16) multiplies the bottom halfword of Rn with the bottom halfword of Rm, and the top halfword of Rn with the top halfword of Rm. It then adds both products to the value in Ra and writes the sum to Rd.

Syntax

```
SMLAD{X}{cond} Rd, Rn, Rm, Ra
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.116 SMLAL

SMLAL (Signed Multiply Accumulate 64 <= 64 + 32 × 32) multiplies Rn and Rm (treated as signed integers) and adds the 64-bit result to the 64-bit signed integer contained in RdHi and RdLo.

Syntax

```
SMLAL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.117 SMLALxy

SMLALxy (Signed Multiply Accumulate; 64 <= 64 + 16 × 16) multiplies the signed integer from the selected half of Rm by the signed integer from the selected half of Rn, and adds the 32-bit result to the 64-bit value in RdHi and RdLo.

Syntax

```
SMLAL<x><y>{cond} RdLo, RdHi, Rn, Rm
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.118 SMLALD

SMLALD (Dual Signed Multiply Accumulate Long; 64 <= 64 + 16 × 16 + 16 × 16) multiplies the bottom halfword of Rn with the bottom halfword of Rm, and the top halfword of Rn with the top halfword of Rm and adds both products to the value in RdLo, RdHi and stores the result in RdLo and RdHi.

Syntax

```
SMLALD{X}{cond} RdLo, RdHi Rn, Rm
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.119 SMLAWy

SMLAW (Signed Multiply with Accumulate Wide; 32 <= 32 × 16 + 32) multiplies the signed integer from the selected half of Rm by the signed integer from Rn, adds the 32-bit result to the 32-bit value in Ra, and writes the result in Rd.

Syntax

```
SMLAW<y>{cond} Rd, Rn, Rm, Ra
```

where:

<y> can be either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.120 SMLSLD

SMLSLD (Dual Signed Multiply Subtract Accumulate Long; 64 <= 64 + 16 × 16 - 16 × 16) multiplies Rn[15:0] with Rm[15:0] and Rn[31:16] with Rm[31:16]. It then subtracts the second product from the first, adds the difference to the value in RdLo, RdHi, and writes the result to RdLo, RdHi.

Syntax

```
SMLSLD{X}{cond} RdLo, RdHi Rn, Rm
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers and hold the value to be accumulated.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.121 SMMLA

SMMLA (Signed top word Multiply with Accumulate; 32 <= top word (32 × 32 + 32)) multiplies Rn and Rm, adds Ra to the most significant 32 bits of the product, and writes the result in Rd.

Syntax

```
SMMLA{R}{cond} Rd, Rn, Rm, Ra
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.122 SMMLS

SMMLS (Signed top word Multiply with Subtract; 32 <= top word (32 × 32 - 32)) multiplies Rn and Rm, subtracts the product from the value in Ra shifted left by 32 bits, and stores the most significant 32 bits of the result in Rd.

Syntax

```
SMMLS{R}{cond} Rd, Rn, Rm, Ra
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.123 SMMUL

SMMUL (Signed top word Multiply; 32 <= top word (32 × 32)) multiplies Rn and Rm, and writes the most significant 32 bits of the 64-bit result to Rd.

Syntax

```
SMMUL{R}{cond} Rd, Rn, Rm
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.124 SMUAD

SMUAD (Dual Signed Multiply and Add products) multiplies Rn [15:0] with Rm [15:0] and Rn [31:16] with Rm [31:16]. It then adds the products and stores the sum to Rd.

Syntax

```
SMUAD{X}{cond} Rd, Rn, Rm
```

where:

X, if present means that the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.125 SMUSD

SMUSD (Dual Signed Multiply and Subtract products) multiplies Rn [15:0] with Rm [15:0] and Rn [31:16] with Rm [31:16]. It then subtracts the products and stores the sum to Rd.

Syntax

```
SMUSD{X}{cond} Rd, Rn, Rm
```

where:

X, if present means that the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.126 SMULxy

The SMULxy (Signed Multiply (32 <= 16 × 16) instruction multiplies the 16-bit signed integers from the selected halves of Rn and Rm, and places the 32-bit result in Rd.

Syntax

```
SMUL<x><y>{cond} {Rd}, Rn, Rm
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.127 SMULL

The SMULL (signed multiply long; 64 <= 32 × 32) instruction multiplies Rn and Rm (treated as containing as two's complement signed integers) and places the least significant 32 bits of the result in RdLo, and the most significant 32 bits of the result in RdHi.

Syntax

```
SMULL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.128 SMULWy

SMULWy (Signed Multiply Wide; 32 <= 32 × 16) multiplies the signed integer from the chosen half of Rm with the signed integer from Rn, and places the upper 32-bits of the 48-bit result in Rd.

Syntax

```
SMULW<y>{cond} {Rd}, Rn, Rm
```

where:

<y> is either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.129 SRS

SRS (Store Return State) stores the LR and the SPSR of the current mode, at the address contained in the SP of the mode specified by modenum. The optional ! means that the SP value is updated. This is compatible with the normal use of the STM instruction for stack accesses.

Syntax

```
SRS{addr_mode}{cond} sp{!}, #modenum
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).
- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA).

cond is an optional condition code. See *Conditional execution* on page 5-3.

modenum gives the number of the mode whose SP is used.

A.1.130 SSAT

SSAT (Signed Saturate) performs a shift and saturates the result to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1}-1$. If saturation occurs, the Q flag is set.

Syntax

```
SSAT{cond} Rd, #sat, Rm{, shift}
```

where:

<y> is either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 1 to 32.

Rm is the register holding the second multiplicand.

shift is optional shift amount and can be either ASR #n where n is in the range (1 – 32 ARM state, 1 – 31 Thumb state) or LSL #n where n is in the range (0-31).

A.1.131 SSAT16

SSAT16 (Signed Saturate, parallel halfwords) saturates each signed halfword to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} -1$. If saturation occurs, the Q flag is set.

Syntax

```
SSAT16{cond} Rd, #sat, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 1 to 32.

Rn is the register holding the operand.

A.1.132 SSAX

SSAX (Signed Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords.

Syntax

```
SSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.133 SSUB8

SSUB8 (Signed halving bytewise Subtraction) does a signed bytewise subtraction (4 subtracts).

Syntax

```
SSUB8{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination registers.

`Rm` and `Rn` are the register holding the operands.

A.1.134 SSUB16

`SSUB16` (Signed halfword-wise Subtract) does a signed halfword-wise subtraction (2 subtracts).

Syntax

```
SSUB16{cond} {Rd}, Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination registers.

`Rm` and `Rn` are the register holding the operands.

A.1.135 STC

`STC` (Store Coprocessor Registers) writes a coprocessor register to memory (or multiple registers, if `L` is specified).

Syntax

```
STC{L}{cond} coproc, CRd, [Rn]
STC{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
STC{L}{cond} coproc, CRd, [Rn], #{-}offset
STC{L}{cond} coproc, CRd, label
```

where:

`L` specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`coproc` is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

`CRd` is the coprocessor register to be stored.

`Rn` is the register holding the base address for the memory operation.

`offset` is a multiple of four, in the range 0-1020, to be added or subtracted from `Rn`. If `!` is present, the address including the offset is written back into `Rn`.

`label` is a word-aligned PC-relative address label.

A.1.136 STC2

STC2 (Store Coprocessor registers) writes a coprocessor register to memory (or multiple registers, if L is specified).

Syntax

```
STC2{L}{cond} coproc, CRd, [Rn]
STC2{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
STC2{L}{cond} coproc, CRd, [Rn], #{-}offset
STC2{L}{cond} coproc, CRd, label
```

where:

L specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

offset is a multiple of four, in the range 0 – 1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.137 STL

STL (Store-Release Word) stores a word from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STL imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STL{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the source register.

Rm is the base register.

A.1.138 STLB

STLB (Store-Release Byte) stores a byte from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLB{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the source register.

Rn is the base register.

A.1.139 STLEX

STLEX (Store-Release Exclusive Word) stores a word from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEX imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEX{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rn is the base register.

A.1.140 STLEXB

STLEXB (Store-Release Exclusive Byte) stores a byte from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEXB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXB{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rn is the base register.

A.1.141 STLEXD

STLEXD (Store-Release Exclusive Double) stores a doubleword from two registers to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEXD imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXD{cond}{q} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

`Rt` is the first source register. `Rt` must be an even numbered register and not R14.

`Rt2` is the second destination register. `Rt2` must be `R(t + 1)`.

`Rn` is the base register.

A.1.142 STLEXH

`STLEXH` (Store-Release Exclusive) stores a halfword from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`STLEXH` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXH{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rm is the base register.

A.1.143 STLH

STLH (Store-Release Halfword) stores a word from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLH imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLH{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the source register.

Rn is the base register.

A.1.144 STM

STM (Store Multiple registers) writes one or more registers to consecutive addresses in memory to an address specified in a base register.

Syntax

```
STM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).

- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA). For example STMFD is a synonym of STMDB.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the base register, giving the initial address for the transfer.

! if present, specifies that the final address is written back into Rn.

^ if specified (in ARM state and a mode other than User or System) means that data is transferred into or out of the User mode registers instead of the current mode registers.

reglist is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

A.1.145 STR

STR (Store Register) stores a value to memory from an ARM register, optionally updating the register used to give the address.

A variety of addressing options are provided. For full details of the available addressing modes, see *Addressing modes* on page 5-13.

Syntax

```
STR{type}{T}{cond} Rt, [Rn {, #offset}]
STR{type}{cond} Rt, [Rn, #offset]!
STR{type}{T}{cond} Rt, [Rn], #offset
STR{type}{cond} Rt, [Rn, +/-Rm {, shift}]
STR{type}{cond} Rt, [Rn, +/-Rm {, shift}]!
STR{type}{T}{cond} Rt, [Rn], +/-Rm {, shift}
```

where:

type can be any one of:

- B – unsigned Byte (Zero extend to 32 bits on loads.)
- H – unsigned Halfword (Zero extend to 32 bits on loads.)

or omitted, for a Word load.

T specifies that memory is accessed as if the processor was in User mode (not available in all addressing modes).

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the base address for the memory operation.

! if present, specifies that the final address is written back into Rn.

offset is a numeric value.

Rm is a register holding an offset value to be applied.

shift is either a register or immediate based shift to apply to the offset value.

A.1.146 STRD

STRD (Store Register Dual) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing.

Syntax

```
STRD{cond} Rt, Rt2, [Rn {,#+/-<imm>}]
STRD{cond} Rt, Rt2, [<Rn>, #+/-<imm>
STRD{cond} Rt, Rt2, [<Rn>, #+/-<imm>]!
STRD{cond} Rt, Rt2, [{Rn},+/-{Rm}]{!}
STRD{cond} Rt, Rt2, [{Rn}],+/-{Rm}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the first source register. For an ARM instruction Rt must be even-numbered and not R14.

Rt is the second source register. For an ARM instruction Rt2 must be <R(t+1)>.

Rn is the base register. The SP can be used. In the ARM instruction set for offset addressing only, the PC can be used. However, use of the PC is deprecated.

+/- is + or omitted if the value of <Rm> is to be added to the base register value (add = TRUE), or – if it is to be subtracted (add = FALSE). #0 and #-0 generate different instructions.

imm is the immediate offset used to form the address. imm can be omitted, meaning an offset of 0.

Rm contains the offset that is applied to the value of <Rn> to form the address.

A.1.147 STREX

STREX (Store register exclusive). Performs a store to a location marked for exclusive access, returning a status value if the store succeeded. Byte, halfword, word and doubleword variants are provided.

Syntax

```
STREX{cond} Rd, Rt, [Rn {, #offset}]
STREXB{cond} Rd, Rt, [Rn]
STREXH{cond} Rd, Rt, [Rn]
STREXD{cond} Rd, Rt, Rt2, [Rn]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register for the return status.

Rt is the register to store.

Rt2 is the second register for doubleword stores.

Rn is the register holding the address.

offset is an optional value, permitted in Thumb only.

A.1.148 SUB

SUB (Subtract) subtracts the value Operand2 from Rn (or subtracts imm12 from Rn).

Syntax

```
SUB{S}{cond} {Rd,} Rn, <Operand2>
```

```
SUB{cond}{Rd,}, Rn, #imm12 (Only available in Thumb)
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

imm12 is in the range 0-4095.

A.1.149 SVC

SVC (SuperVisor Call) causes an SVC exception (was called SWI in older documentation).

Syntax

```
SVC{cond} #imm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

imm is an integer in the range 0 - 0xFFFFFF (ARM) or 0 - 0xFF (Thumb). This integer is not used by the processor itself, but can be used by exception handler code.

A.1.150 SWP

SWP (Swap registers and memory) performs the following two actions. Data from memory is loaded into Rt. Rt2 is saved to memory, at the address given by Rn. Use of this instruction is deprecated and its use is disabled by default.

Syntax

```
SWP{B}{cond} Rt, Rt2, [Rn]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

B is an optional suffix. If specified, a byte is swapped. If not present, a word is specified.

Rt is the destination register.

Rt2 is the source register and can be the same as Rt.

Rn is the register holding the address and cannot be the same as Rt or Rt2.

A.1.151 SXT

SXT (Signed Extend) extracts the specified byte and extends to 32-bit.

Syntax

```
SXT<extend>{cond} {Rd,} Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.152 SXTA

SXTA (Signed Extend and Add) extracts the specified byte, adds the value from Rn and extends to 32-bit.

Syntax

```
SXTA<extend>{cond} {Rd,} Rn, Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the value to be added.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.153 SYS

SYS (System coprocessor instruction) is used to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers.

Syntax

```
SYS{cond} instruction {,Rn}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`instruction` is a write-only system coprocessor register name.

`Rn` is the register holding the operand.

A.1.154 TBB

`TBB` (Table Branch Byte) causes a PC-relative forward branch using a table of single byte offsets. `Rn` provides a pointer to the table, and `Rm` supplies an index into the table. The branch length is twice the value of the byte returned from the table. The target of the branch table must be in the same execution state. There is no ARM or 16-bit Thumb version of this instruction.

Syntax

```
TBB [Rn, Rm]
```

where:

`Rn` is the base register that holds the address of the table of branch lengths.

`Rm` is a register that holds the index into the table.

A.1.155 TBH

`TBH` (Table Branch Halfword) causes a PC-relative forward branch using a table of halfword offsets. `Rn` provides a pointer to the table, and `Rm` supplies an index into the table. The branch length is twice the value of the halfword returned from the table. The target of the branch table must be in the same execution state.

There is no ARM or 16-bit Thumb version of this instruction.

Syntax

```
TBH [Rn, Rm, LSL #1]
```

where:

`Rn` is the base register that holds the address of the table of branch lengths.

`Rm` is a register that holds the index into the table.

A.1.156 TEQ

`TEQ` (Test Equivalence) does a bitwise AND operation on the value in `Rn` and the value of `Operand2`. This is the same as an `ANDS` instruction, except that the result is discarded.

Syntax

```
TEQ{cond} Rn, <Operand2>
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.157 TST

TST (Test) does an Exclusive OR operation on the value in Rn and the value of Operand2. This is the same as an EORS instruction, except that the result is discarded.

Syntax

```
TST{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.158 UADD8

UADD8 (Unsigned bytewise Add) does an unsigned bytewise addition (4 adds).

Syntax

```
UADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.159 UADD16

UADD16 (Unsigned halfword-wise Add) does an unsigned halfword-wise addition (2 adds).

Syntax

```
UADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.160 UASX

UASX (Unsigned Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords.

Syntax

```
UASX{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.161 UBFX

`UBFX` (Unsigned Bit Field Extract) writes adjacent bits from one register into the least significant bits of a second register and zero extends to 32 bits.

Syntax

```
UBFX{cond} Rd, Rn, #lsb, #width
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register that contains the bits to be extracted.

`lsb` specifies the least significant bit of the bitfield.

`width` is the width of the bitfield.

A.1.162 UDIV

`UDIV` (Unsigned Divide). divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. This instruction is not present in all variants of the ARMv7-A architecture.

Syntax

```
UDIV{cond}{q} {Rd,} Rn, Rm
```

where:

`cond` is the optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N` (narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W` (wide), specifies that the assembler must select a 32-bit encoding for the instruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding.

`Rd` the destination register.

`Rn` is the register that contains the dividend.

`Rm` is the register that contains the divisor.

A.1.163 UHADD8

UHADD8 (Unigned Halving bytewise Add) does an unsigned bytewise addition (4 adds) and halves the results.

Syntax

```
UHADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.164 UHADD16

UHADD16 (Unsigned Halving halfword-wise Add) does an unsigned halfword-wise addition (2 adds) and halves the results.

Syntax

```
UHADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.165 UHASX

UHASX (Unsigned Halving Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and halves the results.

Syntax

```
UHASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.166 UHSAX

UHSAX (Unsigned Halving Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and halves the results.

Syntax

```
UHSAX{cond} {Rd}, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.167 UHSUB8

UHSUB8 (Unsigned Halving bytewise Subtraction) does an unsigned bytewise subtraction (4 subtracts) and halves the results.

Syntax

```
UHSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-32.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.168 UHSUB16

UHSUB16 (Unsigned Halving halfword-wise Subtract) does an unsigned halfword-wise subtraction (2 subtracts) and halves the result.

Syntax

```
UHSUB16{cond} {Rd}, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.169 UMAAL

UMAAL (Unsigned Multiply Accumulate Long; 64 <= 32 + 32 + 32 x 32) multiplies Rn and Rm (treated as unsigned integers) adds the two 32-bit values in RdHi and RdLo, and stores the 64-bit result to RdLo, RdHi.

Syntax

```
UMAAL{cond} RdLo, RdHi, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination accumulator registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.170 UMLAL

UMLAL (Unsigned Multiply Accumulate 64 <= 64 + 32 x 32) multiplies Rn and Rm (treated as unsigned integers) and adds the 64-bit result to the 64-bit unsigned integer contained in RdHi and RdLo.

Syntax

```
UMLAL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination accumulator registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.171 UMULL

UMULL (Unsigned Multiply; 64 <= 32 x 32) multiplies Rn and Rm (treated as unsigned integers) and stores the least significant 32 bits of the result in RdLo, and the most significant 32 bits of the result in RdHi.

Syntax

```
UMULL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.172 UQADD8

UQADD8 (Saturating Unsigned bytewise Add) does an unsigned bytewise addition (4 adds) and saturates the results to the unsigned range $0 \leq x \leq 2^8-1$. The Q flag is not affected by this instruction.

Syntax

```
UQADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.173 UQADD16

UQADD16 (Saturating Unsigned halfword-wise Add) does an unsigned halfword-wise addition (2 adds) and saturates the results to the unsigned range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
UQADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.174 UQASX

UQASX (Saturating Unsigned Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and saturates the results to the unsigned range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
UQASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.175 UQSAX

UQSAX (Saturating Unsigned Subtract Add Exchange) exchanges the halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and saturates the results to the signed range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.176 UQSUB8

UQSUB8 (Saturating Unsigned bytewise Subtract) does bytewise subtraction (4 subtracts), with saturation of the results to the unsigned range $0 \leq x \leq 28-1$. The Q flag is not affected by this instruction.

Syntax

```
UQSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.177 UQSUB16

UQSUB16 (Saturating Unsigned halfword Subtract) does halfword-wise subtraction (two subtracts), with saturation of the results to the unsigned range $0 \leq x \leq 216-1$. The Q flag is not affected by this instruction.

Syntax

```
UQSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.178 USAD8

USAD8 (Unsigned Sum of Absolute Differences) finds the 4 differences between the unsigned values in corresponding bytes of Rn and Rm and adds the absolute values of the 4 differences, and stores the result in Rd.

Syntax

```
USAD8{cond} Rd, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

A.1.179 USADA8

USADA8 (Unsigned Sum of Absolute Differences Accumulate) finds the 4 differences between the unsigned values in corresponding bytes of Rn and Rm and adds the absolute values of the 4 differences to the value in Ra, and stores the result in Rd.

See *Sum of absolute differences* on page 5-10 for more information.

Syntax

```
USADA8{cond} Rd, Rn, Rm, Ra
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

Ra is the register that holds the accumulate value.

A.1.180 USAT

USAT (Unsigned Saturate) performs a shift and saturates the result to the signed range $0 \leq x \leq 2^{sat}-1$. If saturation occurs, the Q flag is set.

Syntax

```
USAT{cond} Rd, #sat, Rm{, shift}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 0 to 31.

Rm is the register holding the operand.

shift is optional shift amount and can be either ASR #n where n is in the range (1 – 32 ARM state, 1-31 Thumb state) or LSL #n where n is in the range (0 – 31).

A.1.181 USAT16

USAT16 (Unigned Saturate, parallel halfwords) saturates each unsigned halfword to the signed range $0 \leq x \leq 2^{sat}-1$. If saturation occurs, the Q flag is set.

Syntax

```
USAT16{cond} Rd, #sat, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 0 to 31.

Rn is the register holding the operand.

A.1.182 USAX

USAX (Unsigned Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords.

Syntax

```
USAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.183 USUB8

USUB8 (Unsigned bytewise Subtraction) does an unsigned bytewise subtraction (4 subtracts).

Syntax

```
USUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.184 USUB16

USUB16 (Unsigned halfword-wise Subtract) does an unsigned halfword-wise subtraction (2 subtracts).

Syntax

```
USUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.185 UXT

UXT (Unsigned Extend) extracts the specified byte and zero extends to a 32-bit value.

Syntax

```
UXT<extend>{cond} {Rd,} Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.186 UXTA

UXTA (Unsigned Extend and Add) extracts the specified byte, adds the value from Rn and zero extends to a 32-bit value.

Syntax

```
UXTA<extend>{cond} {Rd,} Rn, Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the value to be added.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.187 WFE

WFE (Wait for Event). If the Event Register is not set, WFE suspends execution until one of the following events occurs:

- an IRQ interrupt (even when CPSR I-bit is set)
- an FIQ interrupt (even when CPSR F-bit is set)
- an asynchronous abort (not when masked by the CPSR A-bit)
- Debug Entry request, even when debug is disabled.
- an Event signaled by another processor using the SEV instruction.

If the Event Register is set, WFE clears it and returns immediately.

Syntax

```
WFE{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.188 WFI

`WFI` (Wait For Interrupt) suspends execution until one of the following events occurs:

- An IRQ interrupt (even when CPSR I-bit is set).
- An FIQ interrupt (even when CPSR F-bit is set).
- An asynchronous abort (not when masked by the CPSR A-bit).
- Debug Entry request, even when debug is disabled.

If the Event Register is set, `WFI` clears it and returns immediately.

Syntax

```
WFI{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.189 YIELD

`YIELD` indicates to the hardware that the current thread is performing a task that can be swapped out (for example, a spinlock). Hardware could use this hint to suspend and resume threads in a multithreading system.

Syntax

```
YIELD{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.