Chapter A4
ARM Instructions

This chapter describes the syntax and usage of every ARM® instruction, in the sections:

A4.1 Alphabetical list of ARM instructions

Every ARM instruction is listed on the following pages. Each instruction description shows:

- the instruction encoding
- the instruction syntax
- the version of the ARM architecture where the instruction is valid
- any exceptions that apply
- an example in pseudo-code of how the instruction operates
- notes on usage and special cases.

A4.1.1 General notes

These notes explain the types of information and abbreviations used on the instruction pages.

Addressing modes

Many instructions refer to one of the addressing modes described in Chapter A5 *ARM Addressing Modes*. The description of the referenced addressing mode should be considered an intrinsic part of the instruction description.

In particular:

- The addressing mode's encoding diagram and assembler syntax provide additional details over and above the instruction's encoding diagram and assembler syntax.
- The addressing mode's Operation pseudo-code calculates values used in the instruction's pseudo-code, and in some cases specify additional effects of the instruction.
- All usage notes, operand restrictions, and other notes about the addressing mode apply to the instruction.

Syntax abbreviations

The following abbreviations are used in the instruction pages:

`immed_n`	This is an immediate value, where `n` is the number of bits. For example, an 8-bit immediate value is represented by: `immed_8`
`offset_n`	This is an offset value, where `n` is the number of bits. For example, an 8-bit offset value is represented by: `offset_8` The same construction is used for signed offsets. For example, an 8-bit signed offset is represented by: `signed_offset_8`

Encoding diagram and assembler syntax

For the conventions used, see *Assembler syntax descriptions* on page xxii.

Architecture versions

This gives details of architecture versions where the instruction is valid. For further information on architecture versions, see *Architecture versions and variants* on page xiii.

Exceptions

This gives details of which exceptions can occur during the execution of the instruction. Prefetch Abort is not listed in general, both because it can occur for any instruction and because if an abort occurred during instruction fetch, the instruction bit pattern is not known. (Prefetch Abort is however listed for `BKPT`, since it can generate a Prefetch Abort exception without these considerations applying.)

Operation

This gives a pseudo-code description of what the instruction does. For details of conventions used in this pseudo-code, see *Pseudo-code descriptions of instructions* on page xxi.

Information on usage

Usage sections are included where appropriate to supply suggestions and other information about how to use the instruction effectively.

A4.1.2 ADC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	0	1	S	Rn	Rd	shifter_operand

`ADC` (Add with Carry) adds two values and the Carry flag. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

`ADC` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the addition, and the `C` and `V` flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ADC`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand + C Flag
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand + C Flag)
        V Flag = OverflowFrom(Rn + shifter_operand + C Flag)
```

Usage

Use `ADC` to synthesize multi-word addition. If register pairs R0, R1 and R2, R3 hold 64-bit values (where R0 and R2 hold the least significant words) the following instructions leave the 64-bit sum in R4, R5:

```
    ADDS R4,R0,R2
    ADC  R5,R1,R3
```

If the second instruction is changed from:

```
    ADC  R5,R1,R3
```

to:

```
    ADCS R5,R1,R3
```

the resulting values of the flags indicate:

N	The 64-bit addition produced a negative result.
C	An unsigned overflow occurred.
V	A signed overflow occurred.
Z	The most significant 32 bits are all zero.

The following instruction produces a single-bit Rotate Left with Extend operation (33-bit rotate through the Carry flag) on R0:

```
    ADCS R0,R0,R0
```

See *Data-processing operands - Rotate right with extend* on page A5-17 for information on how to perform a similar rotation to the right.

A4.1.3 ADD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	1	0	0	S	Rn	Rd	shifter operand

`ADD` adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

`ADD` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the addition, and the `C` and `V` flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ADD`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn + shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = CarryFrom(Rn + shifter_operand)
        V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

Use `ADD` to add two values together.

To increment a register value in `Rx` use:

```
ADD Rx, Rx, #1
```

You can perform constant multiplication of `Rx` by 2^n+1 into `Rd` with:

```
ADD Rd, Rx, Rx, LSL #n
```

To form a PC-relative address use:

```
ADD Rd, PC, #offset
```

where the offset must be the difference between the required address and the address held in the PC, where the PC is the address of the `ADD` instruction itself plus 8 bytes.

A4.1.4 AND

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	0	0	S	Rn	Rd	shifter_operand

`AND` performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

`AND` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `AND`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

`AND` is most useful for extracting a field from a register, by ANDing the register with a mask value that has 1s in the field to be extracted, and 0s elsewhere.

A4.1.5 B, BL

31 28	27	26	25	24	23 0
cond	1	0	1	L	signed_immed_24

`B` (Branch) and `BL` (Branch and Link) cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

`BL` also stores a return address in the link register, R14 (also known as LR).

Syntax

```
B{L}{<cond>}  <target_address>
```

where:

`L` Causes the `L` bit (bit 24) in the instruction to be set to 1. The resulting instruction stores a return address in the link register (R14). If L is omitted, the L bit is 0 and the instruction simply branches without storing a return address.

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<target_address>`

Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits.
2. Shifting the result left two bits to form a 32-bit value.
3. Adding this to the contents of the PC, which contains the address of the branch instruction plus 8 bytes.

The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-11 for precise range).

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if L == 1 then
        LR = address of the instruction after the branch instruction
    PC = PC + (SignExtend_30(signed_immed_24) << 2)
```

Usage

Use `BL` to perform a subroutine call. The return from subroutine is achieved by copying R14 to the PC. Typically, this is done by one of the following methods:

- Executing a `BX R14` instruction, on architecture versions that support that instruction.
- Executing a `MOV PC,R14` instruction.
- Storing a group of registers and R14 to the stack on subroutine entry, using an instruction of the form:

  ```
  STMFD R13!,{<registers>,R14}
  ```

  and then restoring the register values and returning with an instruction of the form:

  ```
  LDMFD R13!,{<registers>,PC}
  ```

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) must:

1. Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.
2. Subtract the base address from the target address to form a byte offset. This offset is always a multiple of four, because all ARM instructions are word-aligned.
3. If the byte offset is outside the range –33554432 to +33554428, use an alternative code-generation strategy or produce an error as appropriate.
4. Otherwise, set the signed_immed_24 field of the instruction to bits{25:2] of the byte offset.

Notes

Memory bounds Branching backwards past location zero and forwards over the end of the 32-bit address space is UNPREDICTABLE.

A4.1.6 BIC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	1	0	S	Rn	Rd	shifter_operand

`BIC` (Bit Clear) performs a bitwise AND of one value with the complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the BIC operation.

`BIC` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Causes the S bit, bit[20], in the instruction to be set to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `BIC`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn AND NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `BIC` to clear selected bits in a register. For each bit, `BIC` with 1 clears the bit, and `BIC` with 0 leaves it unchanged.

A4.1.7 BKPT

31 28	27	26	25	24	23	22	21	20	19 8	7 4	3 0
1 1 1 0	0	0	0	1	0	0	1	0	immed	0 1 1 1	immed

BKPT (Breakpoint) causes a software breakpoint to occur. This breakpoint can be handled by an exception handler installed on the Prefetch Abort vector. In implementations that also include debug hardware, the hardware can optionally override this behavior and handle the breakpoint itself. When this occurs, the Prefetch Abort exception context is presented to the debugger.

Syntax

```
BKPT  <immed_16>
```

where:

`<immed_16>`	Is a 16-bit immediate value. The top 12 bits of `<immed_16>` are placed in bits[19:8] of the instruction, and the bottom 4 bits are placed in bits[3:0] of the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Architecture version

Version 5 and above.

Exceptions

Prefetch Abort.

Operation

```
if (not overridden by debug hardware)
    R14_abt   = address of BKPT instruction + 4
    SPSR_abt  = CPSR
    CPSR[4:0] = 0b10111            /* Enter Abort mode */
    CPSR[5]   = 0                  /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                  /* Disable normal interrupts */
    CPSR[8]   = 1                  /* Disable imprecise aborts - v6 only */
    CPSR[9]   = CP15_reg1_EEbit
    if high vectors configured then
        PC    = 0xFFFF000C
    else
        PC    = 0x0000000C
```

Usage

The exact usage of BKPT depends on the debug system being used. A debug system can use the BKPT instruction in two ways:

- Monitor debug-mode. Debug hardware, (optional prior to ARMv6), does not override the normal behavior of the BKPT instruction, and so the Prefetch Abort vector is entered. The IFSR is updated to indicate a debug event, allowing software to distinguish debug events due to BKPT instruction execution from other system Prefetch Aborts.

  When used in this manner, the BKPT instruction must be avoided within abort handlers, as it corrupts R14_abt and SPSR_abt. For the same reason, it must also be avoided within FIQ handlers, since an FIQ interrupt can occur within an abort handler.

- Halting debug-mode. Debug hardware does override the normal behavior of the BKPT instruction and handles the software breakpoint itself. When finished, it typically either resumes execution at the instruction following the BKPT, or replaces the BKPT in memory with another instruction and resumes execution at that instruction.

  When BKPT is used in this manner, R14_abt and SPSR_abt are not corrupted, and so the above restrictions about its use in abort and FIQ handlers do not apply.

Notes

Condition field BKPT is unconditional. If bits[31:28] of the instruction encode a valid condition other than the AL (always) condition, the instruction is UNPREDICTABLE.

Hardware override Debug hardware in an implementation is specifically permitted to override the normal behavior of the BKPT instruction. Because of this, software must not use this instruction for purposes other than those documented by the debug system being used (if any). In particular, software cannot rely on the Prefetch Abort exception occurring, unless either there is guaranteed to be no debug hardware in the system or the debug system specifies that it occurs.

For more information, consult the documentation for the debug system being used.

A4.1.8 BLX (1)

31 30 29 28	27 26 25	24	23 … 0
1 1 1 1	1 0 1	H	signed_immed_24

`BLX` (1) (Branch with Link and Exchange) calls a Thumb® subroutine from the ARM instruction set at an address specified in the instruction.

This form of `BLX` is unconditional (always causing a change in program flow) and preserves the address of the instruction following the branch in the link register (R14). Execution of Thumb instructions begins at the target address.

Syntax

```
BLX  <target_addr>
```

where:

`<target_addr>` Specifies the address of the Thumb instruction to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits
2. Shifting the result left two bits to form a 32-bit value
3. Setting bit[1] of the result of step 2 to the H bit
4. Adding the result of step 3 to the contents of the PC, which contains the address of the branch instruction plus 8.

The instruction can therefore specify a branch of approximately ±32MB (see *Usage* on page A4-17 for precise range).

Architecture version

Version 5 and above. See *The T and J bits* on page A2-15 for further details of operation on non-T variants.

Exceptions

None.

Operation

```
LR = address of the instruction after the BLX instruction
CPSR T bit = 1
PC = PC + (SignExtend(signed_immed_24) << 2) + (H << 1)
```

Usage

To return from a Thumb subroutine called via `BLX` to the ARM caller, use the Thumb instruction:

```
BX   R14
```

as described in *BX* on page A7-32, or use this instruction on subroutine entry:

```
PUSH {<registers>,R14}
```

and this instruction to return:

```
POP  {<registers>,PC}
```

To calculate the correct value of signed_immed_24, the assembler (or other toolkit component) must:

1. Form the base address for this branch instruction. This is the address of the instruction, plus 8. In other words, this base address is equal to the PC value used by the instruction.
2. Subtract the base address from the target address to form a byte offset. This offset is always even, because all ARM instructions are word-aligned and all Thumb instructions are halfword-aligned.
3. If the byte offset is outside the range –33554432 to +33554430, use an alternative code-generation strategy or produce an error as appropriate.
4. Otherwise, set the signed_immed_24 field of the instruction to bits[25:2] of the byte offset, and the H bit of the instruction to bit[1] of the byte offset.

Notes

Condition	Unlike most other ARM instructions, this instruction cannot be executed conditionally.
Bit[24]	This bit is used as bit[1] of the target address.

A4.1.9 BLX (2)

31 30 29 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0 0 1 0	SBO	SBO	SBO	0 0 1 1	Rm

`BLX` (2) calls an ARM or Thumb subroutine from the ARM instruction set, at an address specified in a register.

It sets the CPSR T bit to bit[0] of Rm. This selects the instruction set to be used in the subroutine.

The branch target address is the value of register Rm, with its bit[0] forced to zero.

It sets R14 to a return address. To return from the subroutine, use a `BX R14` instruction, or store R14 on the stack and reload the stored value into the PC.

Syntax

```
BLX{<cond>}  <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rm>`	Is the register containing the address of the target instruction. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction. If R15 is specified for `<Rm>`, the results are UNPREDICTABLE.

Architecture version

Version 5 and above. See *The T and J bits* on page A2-15 for further details of operation on non-T variants.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    target = Rm
    LR = address of instruction after the BLX instruction
    CPSR T bit = target[0]
    PC = target AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

A4.1.10 BX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	0	1	Rm

`BX` (Branch and Exchange) branches to an address, with an optional switch to Thumb state.

Syntax

```
BX{<cond>}  <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rm>`	Holds the value of the branch target address. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

Architecture version

Version 5 and above, and T variants of version 4. See *The T and J bits* on page A2-15 for further details of operation on non-T variants of version 5.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    CPSR T bit = Rm[0]
    PC = Rm AND 0xFFFFFFFE
```

Notes

ARM/Thumb state transfers

If Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15 Register 15 can be specified for `<Rm>`, but doing so is discouraged.

In a `BX R15` instruction, R15 is read as normal for ARM code, that is, it is the address of the `BX` instruction itself plus 8. The result is to branch to the second following word, executing in ARM state. This is precisely the same effect that would have been obtained if a `B` instruction with an offset field of 0 had been executed, or an `ADD PC,PC,#0` or `MOV PC,PC` instruction. In new code, use these instructions in preference to the more complex `BX PC` instruction.

A4.1.11 BXJ

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	1	0	SBO	SBO	SBO	0	0	1	0	Rm

`BXJ` (Branch and change to Jazelle® state) enters Jazelle state if Jazelle is available and enabled. Otherwise `BXJ` behaves exactly as `BX` (see *BX* on page A4-20).

Syntax

```
BXJ{<cond>}  <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rm>`	Holds the value of the branch target address for use if Jazelle state is not available. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

Architecture version

Version 6 and above, plus ARMv5TEJ.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if (JE bit of Main Configuration register) == 0 then
        T Flag = Rm[0]
        PC = Rm AND 0xFFFFFFFE
    else
        jpc = SUB-ARCHITECTURE DEFINED value
        invalidhandler = SUB-ARCHITECTURE DEFINED value
        if (Jazelle Extension accepts opcode at jpc) then
            if (CV bit of Jazelle OS Control register) == 0 then
                PC = invalidhandler
            else
                J Flag = 1
                Start opcode execution at jpc
        else
            if ((CV bit of Jazelle OS Control register) == 0) AND
                            (IMPLEMENTATION DEFINED CONDITION) then
                PC = invalidhandler
            else
                /* Subject to SUB-ARCHITECTURE DEFINED restrictions on Rm: */
                T Flag = Rm[0]
                PC = Rm AND 0xFFFFFFFE
```

Usage

This instruction must only be used if one of the following conditions is true:

- The JE bit of the Main Configuration Register is 0.
- The Enabled Java Virtual Machine in use conforms to all the SUB-ARCHITECTURE DEFINED restrictions of the Jazelle Extension hardware being used.

Notes

ARM/Thumb state transfers

IF (JE bit of Main Configuration register) == 0

AND Rm[1:0] == 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Use of R15 If register 15 is specified for `<Rm>`, the result is UNPREDICTABLE.

Jazelle opcode address

The Jazelle opcode address is determined in a SUB-ARCHITECTURE DEFINED manner, typically from the contents of a specific general-purpose register, the *Jazelle Program Counter* (jpc).

A4.1.12 CDP

31 28	27	26	25	24	23 20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	CRn	CRd	cp_num	opcode_2	0	CRm

`CDP` (Coprocessor Data Processing) tells the coprocessor whose number is cp_num to perform an operation that is independent of ARM registers and memory. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
CDP{<cond>}  <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
CDP2         <coproc>, <opcode_1>, <CRd>, <CRn>, <CRm>, <opcode_2>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`CDP2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.
`<CRd>`	Specifies the destination coprocessor register for the instruction.
`<CRn>`	Specifies the coprocessor register that contains the first operand.
`<CRm>`	Specifies the coprocessor register that contains the second operand.
`<opcode_2>`	Specifies (in a coprocessor-specific manner) which coprocessor operation is to be performed.

Architecture version

`CDP` is in all versions.

`CDP2` is in version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    Coprocessor[cp_num]-dependent operation
```

Usage

Use `CDP` to initiate coprocessor instructions that do not operate on values in ARM registers or in main memory. An example is a floating-point multiply instruction for a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bits[11:8], and bit[4] are architecturally defined. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional for coprocessors 0-13, regardless of the architecture version, and is optional for coprocessors 14 and 15 before ARMv6. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

A4.1.13 CLZ

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	1	0	SBO	Rd	SBO	0	0	0	1	Rm

`CLZ` (Count Leading Zeros) returns the number of binary zero bits before the first binary one bit in a value.

`CLZ` does not update the condition code flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<Rm>`	Specifies the source register for this operation. If R15 is specified for `<Rm>`, the result is UNPREDICTABLE.

Architecture version

Version 5 and above.

Exceptions

None.

Operation

```
if Rm == 0
    Rd = 32
else
    Rd = 31 - (bit position of most significant'1' in Rm)
```

Usage

Use `CLZ` followed by a left shift of `Rm` by the resulting `Rd` value to normalize the value of register `Rm`. This shifts `Rm` so that its most significant 1 bit is in bit[31]. Using `MOVS` rather than `MOV` sets the Z flag in the special case that `Rm` is zero and so does not have a most significant 1 bit:

```
    CLZ   Rd, Rm
    MOVS  Rm, Rm, LSL Rd
```

A4.1.14 CMN

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 1 1	1	Rn	SBZ	shifter_operand

`CMN` (Compare Negative) compares one value with the twos complement of a second value. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

`CMN` updates the condition flags, based on the result of adding the two values.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand.
`<shifter_operand>`	Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `CMN`. Instead, see *Multiply instruction extension space* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn + shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = CarryFrom(Rn + shifter_operand)
    V Flag = OverflowFrom(Rn + shifter_operand)
```

Usage

`CMN` performs a comparison by adding the value of `<shifter_operand>` to the value of register `<Rn>`, and updates the condition code flags (based on the result). This is almost equivalent to subtracting the negative of the second operand from the first operand, and setting the flags on the result.

The difference is that the flag values generated can differ when the second operand is 0 or `0x80000000`. For example, this instruction always leaves the C flag = 1:

```
CMP Rn, #0
```

and this instruction always leaves the C flag = 0:

```
CMN Rn, #0
```

A4.1.15 CMP

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	0	1	0	1	Rn	SBZ	shifter_operand

`CMP` (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

`CMP` updates the condition flags, based on the result of subtracting the second value from the first.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand.
`<shifter_operand>`	Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `CMP`. Instead, see *Multiply instruction extension space* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn - shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = NOT BorrowFrom(Rn - shifter_operand)
    V Flag = OverflowFrom(Rn - shifter_operand)
```

A4.1.16 CPS

31 30 29 28	27 26 25 24 23 22 21 20	19 18	17	16	15 9	8	7	6	5	4 0
1 1 1 1	0 0 0 1 0 0 0 0	imod	mmod	0	SBZ	A	I	F	0	mode

`CPS` (Change Processor State) changes one or more of the mode, A, I, and F bits of the CPSR, without changing the other CPSR bits.

Syntax

```
CPS<effect> <iflags> {, #<mode>}

CPS     #<mode>
```

where:

`<effect>` Specifies what effect is wanted on the interrupt disable bits A, I, and F in the CPSR. This is one of:

`IE`	Interrupt Enable, encoded by imod == 0b10. This sets the specified bits to 0.
`ID`	Interrupt Disable, encoded by imod == 0b11. This sets the specified bits to 1.

If `<effect>` is specified, the bits to be affected are specified by `<iflags>`. These are encoded in the A, I, and F bits of the instruction. The mode can optionally be changed by specifying a mode number as `<mode>`.

If `<effect>` is not specified, then:

- `<iflags>` is not specified and the A, I, and F mask settings are not changed
- the A, I, and F bits of the instruction are zero
- imod = 0b00
- mmod = 0b1
- `<mode>` specifies the new mode number.

`<iflags>` Is a sequence of one or more of the following, specifying which interrupt disable flags are affected:

`a`	Sets the A bit in the instruction, causing the specified effect on the CPSR A (imprecise data abort) bit.
`i`	Sets the I bit in the instruction, causing the specified effect on the CPSR I (IRQ interrupt) bit.
`f`	Sets the F bit in the instruction, causing the specified effect on the CPSR F (FIQ interrupt) bit.

`<mode>` Specifies the number of the mode to change to. If it is present, then mmod == 1 and the mode number is encoded in the mode field of the instruction. If it is omitted, then mmod == 0 and the mode field of the instruction is zero. See *The mode bits* on page A2-14 for details.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if InAPrivilegedMode() then
    if imod[1] == 1 then
        if A == 1 then CPSR[8] = imod[0]
        if I == 1 then CPSR[7] = imod[0]
        if F == 1 then CPSR[6] = imod[0]
    /* else no change to the mask */
    if mmod == 1 then
        CPSR[4:0] = mode
```

Notes

User mode `CPS` has no effect in User mode.

Meaningless bit combinations

The following combinations of imod and mmod are meaningless:

- imod == 0b00, mmod == 0
- imod == 0b01, mmod == 0
- imod == 0b01, mmod == 1

An assembler must not generate them. The effects are UNPREDICTABLE on execution.

Condition Unlike most other ARM instructions, `CPS` cannot be executed conditionally.

Reserved modes An attempt to change mode to a reserved value is UNPREDICTABLE

Examples

```
    CPSIE   a,#31   ; enable imprecise data aborts, change to System mode
    CPSID   if      ; disable interrupts and fast interrupts
    CPS     #16     ; change to User mode
```

A4.1.17 CPY

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11	10	9	8	7	6	5	4	3 0
cond	0	0	0	1	1	0	1	0	SBZ	Rd	0	0	0	0	0	0	0	0	Rm

`CPY` (Copy) copies a value from one register to another. It is a synonym for `MOV`, with no flag setting and no shift. See *MOV* on page A4-68.

Syntax

```
CPY{<cond>}  <Rd>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the source register.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rm
```

A4.1.18 EOR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	0	1	S	Rn	Rd	shifter_operand

`EOR` (Exclusive OR) performs a bitwise Exclusive-OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the exclusive OR operation.

`EOR` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `EOR`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn EOR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `EOR` to invert selected bits in a register. For each bit, `EOR` with 1 inverts that bit, and `EOR` with 0 leaves it unchanged.

A4.1.19 LDC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	P	U	N	W	1	Rn	CRd	cp_num	8_bit_word_offset

`LDC` (Load Coprocessor) loads memory data from a sequence of consecutive memory addresses to a coprocessor.

If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
LDC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
LDC2{L}         <coproc>, <CRd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`LDC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`L`	Sets the N bit (bit[22]) in the instruction to 1 and specifies a long load (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short load.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<CRd>`	Specifies the coprocessor destination register.

`<addressing_mode>`

Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-49. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

`LDC` is in all versions.

`LDC2` is in version 5 and above.

Exceptions

Undefined Instruction, Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    load Memory[address,4] for Coprocessor[cp_num]
    while (NotFinished(Coprocessor[cp_num]))
        address = address + 4
        load Memory[address,4] for Coprocessor[cp_num]
    assert address == end_address
```

Usage

`LDC` is useful for loading coprocessor data from memory.

Notes

Coprocessor fields Only instruction bits[31:23], bits[21:16], and bits[11:0] are ARM architecture-defined. The remaining fields (bit[22] and bits[15:12]) are recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction bits[7:0] are also not defined by the ARM architecture, and can be used to specify additional coprocessor options.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the load coprocessor register instruction ignores the least significant two bits of the address. If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

A4.1.20 LDM (1)

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	0	W	1	Rn	register_list

`LDM` (1) (Load Multiple) loads a non-empty subset, or possibly all, of the general-purpose registers from sequential memory locations. It is useful for block loads, stack operations and procedure exit sequences.

The general-purpose registers loaded can include the PC. If they do, the word loaded for the PC is treated as an address and a branch occurs to that address. In ARMv5 and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a `BX (loaded_value)` instruction had been executed (but see also *The T and J bits* on page A2-15 for operation on non-T variants of ARMv5). In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though the instruction `MOV PC,(loaded_value)` had been executed.

Syntax

`LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers>`

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P, U, and W bits of the instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as the base register `<Rn>` gives an UNPREDICTABLE result.

`!` Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If `!` is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in `<registers>`, it changes when a value is loaded into it.)

`<registers>`

Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`). If the PC is specified in the register list (opcode bit[15] is set), the instruction causes a branch to the address (data) loaded into the PC.

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if register_list[15] == 1 then
        value = Memory[address,4]
        if (architecture version 5 or above) then
            pc = value AND 0xFFFFFFFE
            T Bit = value[0]
        else
            pc = value AND 0xFFFFFFFC
        address = address + 4
    assert end_address == address - 4
```

Notes

Operand restrictions

If the base register `<Rn>` is specified in `<registers>`, and base register write-back is specified, the final value of `<Rn>` is UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the Load Multiple instructions ignore the least significant two bits of the address. If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of a value loaded for R15 are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Memory access restrictions* on page B2-13for details.

A4.1.21 LDM (2)

31 28	27 26 25	24	23	22	21	20	19 16	15	14 0
cond	1 0 0	P	U	1	0	1	Rn	0	register_list

`LDM` (2) loads User mode registers when the processor is in a privileged mode. This is useful when performing process swaps, and in instruction emulators. `LDM` (2) loads a non-empty subset of the User mode general-purpose registers from sequential memory locations.

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>, <registers_without_pc>^
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the `LDM` instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as `<Rn>` gives an UNPREDICTABLE result.

`<registers_without_pc>`

Is a list of registers, separated by commas and surrounded by { and }. This list must not include the PC, and specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`).

For each of i=0 to 14, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

`^` For an `LDM` instruction that does not load the PC, this indicates that User mode registers are to be loaded.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 14
        if register_list[i] == 1
            Ri_usr = Memory[address,4]
            address = address + 4
    assert end_address == address - 4
```

Notes

Write-back Setting bit[21] (the W bit) has UNPREDICTABLE results.

User and System mode

This form of LDM is UNPREDICTABLE in User mode or System mode.

Base register mode The base register is read from the current processor mode registers, not the User mode registers.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the Load Multiple instructions ignore the least significant two bits of the address. If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Memory access restrictions* on page B2-13 for details.

Banked registers In ARM architecture versions earlier than ARMv6, this form of LDM must not be followed by an instruction that accesses banked registers. A following NOP is a good way to ensure this.

A4.1.22 LDM (3)

31 28	27	26	25	24	23	22	21	20	19 16	15	14 0
cond	1	0	0	P	U	1	W	1	Rn	1	register_list

`LDM` (3) loads a subset, or possibly all, of the general-purpose registers and the PC from sequential memory locations. Also, the SPSR of the current mode is copied to the CPSR. This is useful for returning from an exception.

The value loaded for the PC is treated as an address and a branch occurs to that address. In ARMv5 and above, and in T variants of version 4, the value copied from the SPSR T bit to the CPSR T bit determines whether execution continues after the branch in ARM state or in Thumb state (but see also *The T and J bits* on page A2-15 for operation on non-T variants of ARMv5). In earlier architecture versions, it continues after the branch in ARM state (the only possibility in those architecture versions).

Syntax

```
LDM{<cond>}<addressing_mode>  <Rn>{!}, <registers_and_pc>^
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`
Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P, U, and W bits of the instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. Using R15 as `<Rn>` gives an UNPREDICTABLE result.

`!` Sets the W bit, and the instruction writes a modified value back to its base register Rn (see *Addressing Mode 4 - Load and Store Multiple* on page A5-41). If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way. (However, if the base register is included in `<registers>`, it changes when a value is loaded into it.)

`<registers_and_pc>`
Is a list of registers, separated by commas and surrounded by { and }. This list must include the PC, and specifies the set of registers to be loaded by the `LDM` instruction.

The registers are loaded in sequence, the lowest-numbered register from the lowest memory address (`start_address`), through to the highest-numbered register from the highest memory address (`end_address`).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise.

`^` For an `LDM` instruction that loads the PC, this indicates that the SPSR of the current mode is copied to the CPSR.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    address = start_address

    for i = 0 to 14
        if register_list[i] == 1 then
            Ri = Memory[address,4]
            address = address + 4

    if CurrentModeHasSPSR() then
        CPSR = SPSR
    else
        UNPREDICTABLE

    value = Memory[address,4]
    PC = value
    address = address + 4
    assert end_address == address - 4
```

Notes

User and System mode

This instruction is UNPREDICTABLE in User or System mode.

Operand restrictions

If the base register `<Rn>` is specified in `<registers_and_pc>`, and base register write-back is specified, the final value of `<Rn>` is UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the Load Multiple instructions ignore the least significant two bits of the address. If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if alignment checking is enabled.

For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

ARM/Thumb state transfers (ARM architecture versions 4T, 5 and above)

If the SPSR T bit is 0 and bit[1] of the value loaded into the PC is 1, the results are UNPREDICTABLE because it is not possible to branch to an ARM instruction at a non word-aligned address. Note that no special precautions against this are needed on normal exception returns, because exception entries always either set the T bit of the SPSR to 1 or bit[1] of the return link value in R14 to 0.

Time order The time order of the accesses to individual words of memory generated by this instruction is not defined. See *Memory access restrictions* on page B2-13 for details.

A4.1.23 LDR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	1	Rn	Rd	addr_mode

`LDR` (Load Register) loads a word from a memory address.

If the PC is specified as register `<Rd>`, the instruction loads a data word which it treats as an address, then branches to that address. In ARMv5T and above, bit[0] of the loaded value determines whether execution continues after this branch in ARM state or in Thumb state, as though a `BX (loaded_value)` instruction had been executed. In earlier versions of the architecture, bits[1:0] of the loaded value are ignored and execution continues in ARM state, as though a `MOV PC,(loaded_value)` instruction had been executed.

Syntax

```
LDR{<cond>}  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        data = Memory[address,4] Rotate_Right (8 * address[1:0])
    else    /* CP15_reg_Ubit == 1 */
        data = Memory[address,4]
    if (Rd is R15) then
        if (ARMv5 or above) then
            PC = data AND 0xFFFFFFFE
            T Bit = data[0]
        else
            PC = data AND 0xFFFFFFFC
    else
        Rd = data
```

Usage

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code. Combined with a suitable addressing mode, `LDR` allows 32-bit memory data to be loaded into a general-purpose register where its value can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address.

To synthesize a Branch with Link, precede the `LDR` instruction with `MOV LR, PC`.

Alignment

ARMv5 and below

If the address is not word-aligned, the loaded value is rotated right by 8 times the value of bits[1:0] of the address. For a little-endian memory system, this rotation causes the addressed byte to occupy the least significant byte of the register. For a big-endian memory system, it causes the addressed byte to occupy bits[31:24] or bits[15:8] of the register, depending on whether bit[0] of the address is 0 or 1 respectively.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

ARMv6 and above

From ARMv6, a byte-invariant mixed-endian format is supported, along with an alignment-checking option. The pseudo-code for the ARMv6 case assumes that unaligned mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

Notes

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Use of R15 If R15 is specified for `<Rd>`, the value of the address of the loaded value must be word aligned. That is, address[1:0] must be 0b00. In addition, for Thumb interworking reasons, R15[1:0] must not be loaded with the value 0b10. If these constraints are not met, the result is UNPREDICTABLE.

ARM/Thumb state transfers (ARM architecture version 5 and above)

If bits[1:0] of a value loaded for R15 are 0b10, the result is UNPREDICTABLE, as branches to non word-aligned addresses are impossible in ARM state.

A4.1.24 LDRB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	1	W	1	Rn	Rd	addr_mode

`LDRB` (Load Register Byte) loads a byte from memory and zero-extends the byte to a 32-bit word.

Syntax

```
LDR{<cond>}B  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If register 15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    Rd = Memory[address,1]
```

Usage

Combined with a suitable addressing mode, `LDRB` allows 8-bit memory data to be loaded into a general-purpose register where it can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.25 LDRBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	1	Rn	Rd	addr_mode

`LDRBT` (Load Register Byte with Translation) loads a byte from memory and zero-extends the byte to a 32-bit word.

If `LDRBT` is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
LDR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
if ConditionPassed(cond) then
    Rd = Memory[address,1]
    Rn = address
```

Usage

`LDRBT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.26 LDRD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	0	Rn	Rd	addr_mode	1	1	0	1	addr_mode

`LDRD` (Load Registers Doubleword) loads a pair of ARM registers from two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

A greater variety of addressing modes is available than for a two-register `LDM`.

Syntax

```
LDR{<cond>}D  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the even-numbered destination register for the memory word addressed by `<addressing_mode>`. The immediately following odd-numbered register is the destination register for the next memory word. If `<Rd>` is R14, which would specify R15 as the second destination register, the instruction is UNPREDICTABLE. If `<Rd>` specifies an odd-numbered register, the instruction is UNDEFINED.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn, and addr_mode bits of the instruction. The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

The address generated by `<addressing_mode>` is the address of the lower of the two words loaded by the `LDRD` instruction. The address of the higher word is generated by adding 4 to this address.

Architecture version

Version 5TE and above, excluding ARMv5TExP.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)

if ConditionPassed(cond) then
    if (Rd is even-numbered) and (Rd is not R14) and
            (address[1:0] == 0b00) and
            ((CP15_reg1_Ubit == 1) or (address[2] == 0)) then
        Rd = Memory[address,4]
        R(d+1) = memory[address+4,4]
    else
        UNPREDICTABLE
```

Notes

Operand restrictions

If `<addressing_mode>` performs base register write-back and the base register `<Rn>` is one of the two destination registers of the instruction, the results are UNPREDICTABLE.

If `<addressing_mode>` specifies an index register `<Rm>`, and `<Rm>` is one of the two destination registers of the instruction, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment Prior to ARMv6, if the memory address is not 64-bit aligned, the data read from memory is UNPREDICTABLE. Alignment checking (taking a data abort), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, a byte-invariant mixed-endian format is supported, along with alignment checking options; modulo4 and modulo8. The pseudo-code for the ARMv6 case assumes that unaligned mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

Time order The time order of the accesses to the two memory words is not architecturally defined. In particular, an implementation is allowed to perform the two 32-bit memory accesses in either order, or to combine them into a single 64-bit memory access.

A4.1.27 LDREX

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 1 0 0	1	Rn	Rd	SBO	1 0 0 1	SBO

`LDREX` (Load Register Exclusive) loads a register from memory, and:

- if the address has the Shared memory attribute, marks the physical address as exclusive access for the executing processor in a shared monitor
- causes the executing processor to indicate an active inclusive access in the local monitor.

Syntax

```
LDREX{<cond>} <Rd>, [<Rn>]
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the memory word addressed by `<Rd>`.
`<Rn>`	Specifies the register containing the address.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    Rd = Memory[Rn,4]
    physical_address = TLB(Rn)
    if Shared(Rn) == 1 then
        MarkExclusiveGlobal(physical_address,processor_id,4)
    MarkExclusiveLocal(physical_address,processor_id,4)
    /* See Summary of operation on page A2-49 */
```

Usage

Use `LDREX` in combination with `STREX` to implement inter-process communication in shared memory multiprocessor systems. For more information see *Synchronization primitives* on page A2-44. The mechanism can also be used locally to ensure that an atomic load-store sequence occurs with no intervening context switch.

Notes

Use of R15 If register 15 is specified for `<Rd>` or `<Rn>`, the result is UNPREDICTABLE.

Data Abort If a data abort occurs during a `LDREX` it is UNPREDICTABLE whether the MarkExclusiveGlobal() and MarkExclusiveLocal() operations are executed. Rd is not updated.

Alignment If CP15 register 1(A,U) != (0,0) and Rd<1:0> != 0b00, an alignment exception will be taken.

There is no support for unaligned Load Exclusive. If Rd<1:0> != 0b00 and (A,U) = (0,0), the result is UNPREDICTABLE.

Memory support for exclusives

The behavior of `LDREX` in regions of shared memory that do not support exclusives (for example, have no exclusives monitor implemented) is UNPREDICTABLE.

A4.1.28 LDRH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	1	Rn	Rd	addr_mode	1	0	1	1	addr_mode

`LDRH` (Load Register Halfword) loads a halfword from memory and zero-extends it to a 32-bit word.

Syntax

```
LDR{<cond>}H  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0 then
            data = Memory[address,2]
        else
            data = UNPREDICTABLE
    else    /* CP15_reg1_Ubit == 1 */
        data = Memory[address,2]
    Rd = ZeroExtend(data[15:0])
```

Usage

Used with a suitable addressing mode, `LDRH` allows 16-bit memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing to facilitate position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment Prior to ARMv6, if the memory address is not halfword aligned, the data read from memory is UNPREDICTABLE. Alignment checking (taking a data abort when address[0] != 0), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, a byte-invariant mixed-endian format is supported, along with an alignment checking option. The pseudo-code for the ARMv6 case assumes that mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment, see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

A4.1.29 LDRSB

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	1	Rn	Rd	addr_mode	1 1 0 1	addr_mode

`LDRSB` (Load Register Signed Byte) loads a byte from memory and sign-extends the byte to a 32-bit word.

Syntax

```
LDR{<cond>}SB  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    data = Memory[address,1]
    Rd = SignExtend(data)
```

Usage

Use `LDRSB` with a suitable addressing mode to load 8-bit signed memory data into a general-purpose register where it can be manipulated.

You can perform PC-relative addressing by using the PC as the base register. This facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.30 LDRSH

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0	P	U	I	W	1	Rn	Rd	addr_mode	1 1 1 1	addr_mode

`LDRSH` (Load Register Signed Halfword) loads a halfword from memory and sign-extends the halfword to a 32-bit word.

If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
LDR{<cond>}SH  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

Version 4 and above.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0 then
            data = Memory[address,2]
        else
            data = UNPREDICTABLE
    else    /* CP15_reg1_Ubit == 1 */
        data = Memory[address,2]
    Rd = SignExtend(data[15:0])
```

Usage

Used with a suitable addressing mode, `LDRSH` allows 16-bit signed memory data to be loaded into a general-purpose register where its value can be manipulated.

Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment Prior to ARMv6, if the memory address is not halfword aligned, the data read from memory is UNPREDICTABLE. Alignment checking (taking a data abort when address[0] != 0), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, a byte-invariant mixed-endian format is supported, along with an alignment checking option. The pseudo-code for the ARMv6 case assumes that mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment, see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

A4.1.31 LDRT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	0	1	1	Rn	Rd	addr_mode

`LDRT` (Load Register with Translation) loads a word from memory.

If `LDRT` is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
LDR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register for the loaded value. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        Rd = Memory[address,4] Rotate_Right (8 * address[1:0])
    else    /* CP15_reg1_Ubit == 1 */
        Rd = Memory[address,4]
```

Usage

`LDRT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>` the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment As for `LDR`, see *LDR* on page A4-43.

A4.1.32 MCR

31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	0	CRn	Rd	cp_num	opcode_2	1	CRm

`MCR` (Move to Coprocessor from ARM Register) passes the value of register `<Rd>` to the coprocessor whose number is cp_num.

If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MCR{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MCR2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`MCR2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Is a coprocessor-specific opcode.
`<Rd>`	Is the ARM register whose value is transferred to the coprocessor. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<CRn>`	Is the destination coprocessor register.
`<CRm>`	Is an additional destination or source coprocessor register.
`<opcode_2>`	Is a coprocessor-specific opcode. If it is omitted, `<opcode_2>` is assumed to be 0.

Architecture version

`MCR` is in all versions.

`MCR2` is in version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
```

Usage

Use `MCR` to initiate a coprocessor operation that acts on a value from an ARM register. An example is a fixed-point to floating-point conversion instruction for a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bit[20], bits[15:8], and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional for coprocessors 0-13, regardless of the architecture version, and is optional for coprocessors 14 and 15 before ARMv6. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

A4.1.33 MCRR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 4	3 0
cond	1	1	0	0	0	1	0	0	Rn	Rd	cp_num	opcode	CRm

`MCRR` (Move to Coprocessor from two ARM Registers) passes the values of two ARM registers to a coprocessor.

If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MCRR{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MCRR2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`MCRR2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the `cp_num` field of the instruction. The standard generic coprocessor names are p0, p1, …, p15.
`<opcode>`	Is a coprocessor-specific opcode.
`<Rd>`	Is the first ARM register whose value is transferred to the coprocessor. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<Rn>`	Is the second ARM register whose value is transferred to the coprocessor. If R15 is specified for `<Rn>`, or Rn = Rd, the result is UNPREDICTABLE.
`<CRm>`	Is the destination coprocessor register.

Architecture version

`MCRR` is in version 5TE and above, excluding ARMv5TExP.

`MCRR2` is in version 6 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    send Rd value to Coprocessor[cp_num]
    send Rn value to Coprocessor[cp_num]
```

Usage

Use `MCRR` to initiate a coprocessor operation that acts on values from two ARM registers. An example for a floating-point coprocessor is an instruction to transfer a double-precision floating-point number held in two ARM registers to a floating-point register.

Notes

Coprocessor fields

Only instruction bits[31:8] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional for coprocessors 0-13, regardless of the architecture version, and is optional for coprocessors 14 and 15 before ARMv6. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

Order of transfers

If a coprocessor uses these instructions, it defines how each of the values of `<Rd>` and `<Rn>` is used. There is no architectural requirement for the two register transfers to occur in any particular time order. It is IMPLEMENTATION DEFINED whether Rd is transferred before Rn, after Rn, or at the same time as Rn.

A4.1.34 MLA

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	0	0	1	S	Rd	Rn	Rs	1	0	0	1	Rm

`MLA` (Multiply Accumulate) multiplies two signed or unsigned 32-bit values, and adds a third 32-bit value. The least significant 32 bits of the result are written to the destination register.

`MLA` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<Rd>`	Specifies the destination register.
`<Rm>`	Holds the value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the value to be multiplied with the value of `<Rm>`.
`<Rn>`	Contains the value that is added to the product of `<Rs>` and `<Rm>`.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs + Rn)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Signed and unsigned The `MLA` instruction produces only the lower 32 bits of the 64-bit product. Therefore, `MLA` gives the same answer for multiplication of both signed and unsigned numbers.

C flag The `MLAS` instruction is defined to leave the C flag unchanged in ARMv5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after an `MLAS` instruction.

Operand restriction Specifying the same register for `<Rd>` and `<Rm>` was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed that all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

A4.1.35 MOV

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 1 0 1	S	SBZ	Rd	shifter_operand

`MOV` (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

`MOV` can optionally update the condition code flags, based on the result.

Syntax

```
MOV{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<shifter_operand>`

Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `MOV`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `MOV` to:

- Move a value from one register to another.
- Put a constant value into a register.
- Perform a shift without any other arithmetic or logical operation. Use a left shift by *n* to multiply by 2^n.
- When the PC is the destination of the instruction, a branch occurs. The instruction:

  ```
  MOV PC, LR
  ```

  can therefore be used to return from a subroutine (see instructions *B, BL* on page A4-10). In T variants of architecture 4 and in architecture 5 and above, the instruction `BX LR` must be used in place of `MOV PC, LR`, as the `BX` instruction automatically switches back to Thumb state if appropriate (but see also *The T and J bits* on page A2-15 for operation on non-T variants of ARM architecture version 5).
- When the PC is the destination of the instruction and the S bit is set, a branch occurs and the SPSR of the current mode is copied to the CPSR. This means that you can use a `MOVS PC, LR` instruction to return from some types of exception (see *Exceptions* on page A2-16).

A4.1.36 MRC

31 28	27	26	25	24	23 21	20	19 16	15 12	11 8	7 5	4	3 0
cond	1	1	1	0	opcode_1	1	CRn	Rd	cp_num	opcode_2	1	CRm

`MRC` (Move to ARM Register from Coprocessor) causes a coprocessor to transfer a value to an ARM register or to the condition flags.

If no coprocessors can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MRC{<cond>}  <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
MRC2         <coproc>, <opcode_1>, <Rd>, <CRn>, <CRm>{, <opcode_2>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`MRC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<opcode_1>`	Is a coprocessor-specific opcode.
`<Rd>`	Specifies the destination ARM register for the instruction. If R15 is specified for `<Rd>`, the condition code flags are updated instead of a general-purpose register.
`<CRn>`	Specifies the coprocessor register that contains the first operand.
`<CRm>`	Is an additional coprocessor source or destination register.
`<opcode_2>`	Is a coprocessor-specific opcode. If it is omitted, `<opcode_2>` is assumed to be 0.

Architecture version

`MRC` is in all versions.

`MRC2` is in version 5 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    data = value from Coprocessor[cp_num]
    if Rd is R15 then
        N flag = data[31]
        Z flag = data[30]
        C flag = data[29]
        V flag = data[28]
    else /* Rd is not R15 */
        Rd = data
```

Usage

`MRC` has two uses:

1. If `<Rd>` specifies R15, the condition code flags bits are updated from the top four bits of the value from the coprocessor specified by `<coproc>` (to allow conditional branching on the status of a coprocessor) and the other 28 bits are ignored.

   An example of this use would be to transfer the result of a comparison performed by a floating-point coprocessor to the ARM's condition flags.

2. Otherwise the instruction writes into register `<Rd>` a value from the coprocessor specified by `<coproc>`.

   An example of this use is a floating-point to integer conversion instruction in a floating-point coprocessor.

Notes

Coprocessor fields Only instruction bits[31:24], bit[20], bits[15:8] and bit[4] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional for coprocessors 0-13, regardless of the architecture version, and is optional for coprocessors 14 and 15 before ARMv6. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

A4.1.37 MRRC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 4	3 0
cond	1	1	0	0	0	1	0	1	Rn	Rd	cp_num	opcode	CRm

`MRRC` (Move to two ARM registers from Coprocessor) causes a coprocessor to transfer values to two ARM registers.

If no coprocessors can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
MRRC{<cond>}  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
MRRC2  <coproc>, <opcode>, <Rd>, <Rn>, <CRm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`MRRC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the `cp_num` field of the instruction. The standard generic coprocessor names are p0, p1, …, p15.
`<opcode>`	Is a coprocessor-specific opcode.
`<Rd>`	Is the first destination ARM register. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<Rn>`	Is the second destination ARM register. If R15 is specified for `<Rn>`, the result is UNPREDICTABLE.
`<CRm>`	Is the coprocessor register which supplies the data to be transferred.

Architecture version

`MRRC` is in version 5TE and above, excluding ARMv5TExP.

`MRRC2` is in version 6 and above.

Exceptions

Undefined Instruction.

Operation

```
if ConditionPassed(cond) then
    Rd = first value from Coprocessor[cp_num]
    Rn = second value from Coprocessor[cp_num]
```

Usage

Use `MRRC` to initiate a coprocessor operation that writes values to two ARM registers. An example for a floating-point coprocessor is an instruction to transfer a double-precision floating-point number held in a floating-point register to two ARM registers.

Notes

Operand restrictions

Specifying the same register for `<Rd>` and `<Rn>` has UNPREDICTABLE results.

Coprocessor fields

Only instruction bits[31:8] are defined by the ARM architecture. The remaining fields are recommendations, for compatibility with ARM Development Systems.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional for coprocessors 0-13, regardless of the architecture version, and is optional for coprocessors 14 and 15 before ARMv6. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

Order of transfers

If a coprocessor uses these instructions, it defines which value is written to `<Rd>` and which value to `<Rn>`. There is no architectural requirement for the two register transfers to occur in any particular time order. It is IMPLEMENTATION DEFINED whether Rd is transferred before Rn, after Rn, or at the same time as Rn.

A4.1.38 MRS

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	0	1	0	R	0	0	SBO	Rd	SBZ

`MRS` (Move PSR to general-purpose register) moves the value of the CPSR or the SPSR of the current mode into a general-purpose register. In the general-purpose register, the value can be examined or manipulated with normal data-processing instructions.

Syntax

```
MRS{<cond>}  <Rd>, CPSR
MRS{<cond>}  <Rd>, SPSR
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if R == 1 then
        Rd = SPSR
    else
        Rd = CPSR
```

Usage

The `MRS` instruction is commonly used for three purposes:

- As part of a read/modify/write sequence for updating a PSR. For more details, see *MSR* on page A4-76.
- When an exception occurs and there is a possibility of a nested exception of the same type occurring, the SPSR of the exception mode is in danger of being corrupted. To deal with this, the SPSR value must be saved before the nested exception can occur, and later restored in preparation for the exception return. The saving is normally done by using an `MRS` instruction followed by a store instruction. Restoring the SPSR uses the reverse sequence of a load instruction followed by an `MSR` instruction.
- In process swap code, the programmers' model state of the process being swapped out must be saved, including relevant PSR contents, and similar state of the process being swapped in must be restored. Again, this involves the use of `MRS`/store and load/`MSR` instruction sequences.

Notes

User mode SPSR Accessing the SPSR when in User mode or System mode is UNPREDICTABLE.

A4.1.39 MSR

Immediate operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 0
cond	0 0 1 1 0	R	1 0	field_mask	SBO	rotate_imm	8_bit_immediate

Register operand:

31 28	27 26 25 24 23	22	21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 1 0	R	1 0	field_mask	SBO	SBZ	0 0 0 0	Rm

`MSR` (Move to Status Register from ARM Register) transfers the value of a general-purpose register or an immediate constant to the CPSR or the SPSR of the current mode.

Syntax

```
MSR{<cond>}  CPSR_<fields>, #<immediate>
MSR{<cond>}  CPSR_<fields>, <Rm>
MSR{<cond>}  SPSR_<fields>, #<immediate>
MSR{<cond>}  SPSR_<fields>, <Rm>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<fields>` Is a sequence of one or more of the following:

`c`	sets the control field mask bit (bit 16)
`x`	sets the extension field mask bit (bit 17)
`s`	sets the status field mask bit (bit 18)
`f`	sets the flags field mask bit (bit 19).

`<immediate>` Is the immediate value to be transferred to the CPSR or SPSR. Allowed immediate values are 8-bit immediates (in the range `0x00` to `0xFF`) and values that can be obtained by rotating them right by an even amount in the range 0 to 30. These immediate values are the same as those allowed in the immediate form as shown in *Data-processing operands - Immediate* on page A5-6.

`<Rm>` Is the general-purpose register to be transferred to the CPSR or SPSR.

Architecture version

All.

Exceptions

None.

Operation

There are four categories of PSR bits, according to rules about updating them, see *Types of PSR bits* on page A2-11 for details.

The pseudo-code uses four bit mask constants to identify these categories of PSR bits. The values of these masks depend on the architecture version, see Table A4-1.

Table A4-1 Bit mask constants

Architecture versions	UnallocMask	UserMask	PrivMask	StateMask
4	0x0FFFFF20	0xF0000000	0x0000000F	0x00000000
4T, 5T	0x0FFFFF00	0xF0000000	0x0000000F	0x00000020
5TE, 5TExP	0x07FFFF00	0xF8000000	0x0000000F	0x00000020
5TEJ	0x06FFFF00	0xF8000000	0x0000000F	0x01000020
6	0x06F0FC00	0xF80F0200	0x000001DF	0x01000020

```
if ConditionPassed(cond) then
    if opcode[25] == 1 then
        operand = 8_bit_immediate Rotate_Right (rotate_imm * 2)
    else
        operand = Rm
    if (operand AND UnallocMask) !=0 then
        UNPREDICTABLE              /* Attempt to set reserved bits */
    byte_mask = (if field_mask[0] == 1 then 0x000000FF else 0x00000000) OR
                (if field_mask[1] == 1 then 0x0000FF00 else 0x00000000) OR
                (if field_mask[2] == 1 then 0x00FF0000 else 0x00000000) OR
                (if field_mask[3] == 1 then 0xFF000000 else 0x00000000)
    if R == 0 then
        if InAPrivilegedMode() then
            if (operand AND StateMask) != 0 then
                UNPREDICTABLE      /* Attempt to set non-ARM execution state */
            else
                mask = byte_mask AND (UserMask OR PrivMask)
        else
            mask = byte_mask AND UserMask
        CPSR = (CPSR AND NOT mask) OR (operand AND mask)
    else  /* R == 1 */
        if CurrentModeHasSPSR() then
            mask = byte_mask AND (UserMask OR PrivMask OR StateMask)
            SPSR = (SPSR AND NOT mask) OR (operand AND mask)
        else
            UNPREDICTABLE
```

Usage

Use `MSR` to update the value of the condition code flags, interrupt enables, or the processor mode.

You must normally update the value of a PSR by moving the PSR to a general-purpose register (using the `MRS` instruction), modifying the relevant bits of the general-purpose register, and restoring the updated general-purpose register value back into the PSR (using the `MSR` instruction). For example, a good way to switch the ARM to Supervisor mode from another privileged mode is:

```
MRS   R0,CPSR                  ; Read CPSR
BIC   R0,R0,#0x1F              ; Modify by removing current mode
ORR   R0,R0,#0x13              ; and substituting Supervisor mode
MSR   CPSR_c,R0                ; Write the result back to CPSR
```

For maximum efficiency, `MSR` instructions should only write to those fields that they can potentially change. For example, the last instruction in the above code can only change the CPSR control field, as all bits in the other fields are unchanged since they were read from the CPSR by the first instruction. So it writes to CPSR_c, not CPSR_fsxc or some other combination of fields.

However, if the *only* reason that an `MSR` instruction cannot change a field is that no bits are currently allocated to the field, then the field must be written, to ensure future compatibility.

You can use the immediate form of `MSR` to set any of the fields of a PSR, but you must take care to use the read-modify-write technique described above. The immediate form of the instruction is equivalent to reading the PSR concerned, replacing all the bits in the fields concerned by the corresponding bits of the immediate constant and writing the result back to the PSR. The immediate form must therefore only be used when the intention is to modify all the bits in the specified fields and, in particular, must not be used if the specified fields include any as-yet-unallocated bits. Failure to observe this rule might result in code which has unanticipated side effects on future versions of the ARM architecture.

As an exception to the above rule, it is legitimate to use the immediate form of the instruction to modify the flags byte, despite the fact that bits[26:25] of the PSRs have no allocated function at present. For example, you can use `MSR` to set all four flags (and clear the Q flag if the processor implements the Enhanced DSP extension):

```
MSR    CPSR_f,#0xF0000000
```

Any functionality allocated to bits[26:25] in a future version of the ARM architecture will be designed so that such code does not have unexpected side effects. Several bits must not be changed to reserved values or the results are UNPREDICTABLE. For example, an attempt to write a reserved value to the mode bits (4:0), or changing the J-bit (24).

Notes

The R bit Bit[22] of the instruction is 0 if the CPSR is to be written and 1 if the SPSR is to be written.

User mode CPSR

Any writes to privileged or execution state bits are ignored.

User mode SPSR

Accessing the SPSR when in User mode is UNPREDICTABLE.

System mode SPSR

Accessing the SPSR when in System mode is UNPREDICTABLE.

Obsolete field specification

The `CPSR`, `CPSR_flg`, `CPSR_ctl`, `CPSR_all`, `SPSR`, `SPSR_flg`, `SPSR_ctl` and `SPSR_all` forms of PSR field specification have been superseded by the `csxf` format shown on page A4-76.

`CPSR`, `SPSR`, `CPSR_all` and `SPSR_all` produce a field mask of 0b1001.

`CPSR_flg` and `SPSR_flg` produce a field mask of 0b1000.

`CPSR_ctl` and `SPSR_ctl` produce a field mask of 0b0001.

The T bit or J bit

The `MSR` instruction must not be used to alter the T bit or the J bit in the CPSR. If such an attempt is made, the results are UNPREDICTABLE.

Addressing modes

The immediate and register forms are specified in precisely the same way as the immediate and unshifted register forms of Addressing Mode 1 (see *Addressing Mode 1 - Data-processing operands* on page A5-2). All other forms of Addressing Mode 1 yield UNPREDICTABLE results.

A4.1.40 MUL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 0 0 0	S	Rd	SBZ	Rs	1 0 0 1	Rm

`MUL` (Multiply) multiplies two signed or unsigned 32-bit values. The least significant 32 bits of the result are written to the destination register.

`MUL` can optionally update the condition code flags, based on the result.

Syntax

```
MUL{<cond>}{S}  <Rd>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Specifies the register that contains the first value to be multiplied.
`<Rs>`	Holds the value to be multiplied with the value of `<Rm>`.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = unaffected in v5 and above, UNPREDICTABLE in v4 and earlier
        V Flag = unaffected
```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Signed and unsigned	Because the `MUL` instruction produces only the lower 32 bits of the 64-bit product, `MUL` gives the same answer for multiplication of both signed and unsigned numbers.
C flag	The `MULS` instruction is defined to leave the C flag unchanged in ARM architecture version 5 and above. In earlier versions of the architecture, the value of the C flag was UNPREDICTABLE after a `MULS` instruction.
Operand restriction	Specifying the same register for `<Rd>` and `<Rm>` was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

A4.1.41 MVN

31 … 28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11 … 0
cond	0	0	I	1	1	1	1	S	SBZ	Rd	shifter_operand

`MVN` (Move Not) generates the logical ones complement of a value. The value can be either an immediate value or a value from a register, and can be shifted before the MVN operation.

`MVN` can optionally update the condition code flags, based on the result.

Syntax

```
MVN{<cond>}{S}  <Rd>, <shifter_operand>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<shifter_operand>`

Specifies the operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `MVN`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = NOT shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `MVN` to:

- form a bit mask
- take the ones complement of a value.

A4.1.42 ORR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	1	0	0	S	Rn	Rd	shifter_operand

`ORR` (Logical OR) performs a bitwise (inclusive) OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the OR operation.

`ORR` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see *Addressing Mode 1 - Data-processing operands* on page A5-2). The V flag and the rest of the CPSR are unaffected.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `ORR`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn OR shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = shifter_carry_out
        V Flag = unaffected
```

Usage

Use `ORR` to set selected bits in a register. For each bit, OR with 1 sets the bit, and OR with 0 leaves it unchanged.

A4.1.43 PKHBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 7	6		4	3 0
cond	0	1	1	0	1	0	0	0	Rn	Rd	shift_imm	0	0	1	Rm

`PKHBT` (Pack Halfword Bottom Top) combines the bottom (least significant) halfword of its first operand with the top (most significant) halfword of its shifted second operand. The shift is a left shift, by any amount from 0 to 31.

Syntax

```
PKHBT {<cond>} <Rd>, <Rn>, <Rm> {, LSL #<shift_imm>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand. Bits[15:0] of this operand become bits[15:0] of the result of the operation.
`<Rm>`	Specifies the register that contains the second operand. This is shifted left by the specified amount, then bits[31:16] of this operand become bits[31:16] of the result of the operation.
`<shift_imm>`	Specifies the amount by which `<Rm>` is to be shifted left. This is a value from 0 to 31. If the shift specifier is omitted, a left shift by 0 is used.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0] = Rn[15:0]
    Rd[31:16] = (Rm Logical_Shift_Left shift_imm)[31:16]
```

Usage

To construct the word in Rd consisting of the top half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, use:

```
PKHBT   Rd, Rb, Ra
```

To construct the word in Rd consisting of the bottom half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, use:

```
PKHBT   Rd, Rb, Ra, LSL #16
```

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.44 PKHTB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 7	6	5	4	3 0
cond	0	1	1	0	1	0	0	0	Rn	Rd	shift_imm	1	0	1	Rm

PKHTB (Pack Halfword Top Bottom) combines the top (most significant) halfword of its first operand with the bottom (least significant) halfword of its shifted second operand. The shift is an arithmetic right shift, by any amount from 1 to 32.

Syntax

```
PKHTB {<cond>} <Rd>, <Rn>, <Rm> {, ASR #<shift_imm>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand. Bits[31:16] of this operand become bits[31:16] of the result of the operation.
`<Rm>`	Specifies the register that contains the second operand. This is shifted right arithmetically by the specified amount, then bits[15:0] of this operand become bits[15:0] of the result of the operation.
`<shift_imm>`	Specifies the amount by which `<Rm>` is to be shifted right. A shift by 32 is encoded as `shift_imm == 0`.

If the shift specifier is omitted, the assembler converts the instruction to `PKHBT Rd, Rm, Rn`. This produces the same effect as an arithmetic shift right by 0.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ASR #0` here. It is equivalent to omitting the shift specifier.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift_imm == 0 then          /* ASR #32 case */
        if Rm[31] == 0 then
            Rd[15:0] = 0x0000
        else
            Rd[15:0] = 0xFFFF
    else
        Rd[15:0] = (Rm Arithmetic_Shift_Right shift_imm)[15:0]
    Rd[31:16] = Rn[31:16]
```

Usage

To construct the word in Rd consisting of the top half of register Ra and the top half of register Rb as its most and least significant halfwords respectively, use:

```
    PKHTB   Rd, Ra, Rb, ASR #16
```

You can use this to truncate a Q31 number in Rb, and put the result into the bottom half of Rd. You can scale the Rb value by using a different shift amount.

To construct the word in Rd consisting of the top half of register Ra and the bottom half of register Rb as its most and least significant halfwords respectively, you can use:

```
    PKHTB   Rd, Ra, Rb
```

The assembler converts this into:

```
    PKHBT   Rd, Rb, Ra
```

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.45 PLD

31 30 29 28	27 26	25	24	23	22 21 20	19 16	15 14 13 12	11 0
1 1 1 1	0 1	I	1	U	1 0 1	Rn	1 1 1 1	addr_mode

`PLD` (Preload Data) signals the memory system that memory accesses from a specified address are likely in the near future. The memory system can respond by taking actions which are expected to speed up the memory accesses when they do occur, such as pre-loading the cache line containing the specified address into the cache. `PLD` is a *hint* instruction, aimed at optimizing memory system performance. It has no architecturally-defined effect, and memory systems that do not support this optimization can ignore it. On such memory systems, `PLD` acts as a `NOP`.

Syntax

```
PLD  <addressing_mode>
```

where:

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It specifies the I, U, Rn, and addr_mode bits of the instruction. Only addressing modes with P == 1 and W == 0 are available for this instruction. Pre-indexed and post-indexed addressing modes have P == 0 or W == 1 and so are not available.

Architecture version

Version 5TE and above, excluding ARMv5TExP.

Exceptions

None.

Operation

```
/* No change occurs to programmer's model state, but where
* appropriate, the memory system is signaled that memory accesses
* to the specified address are likely in the near future.
*/
```

Notes

Condition Unlike most other ARM instructions, `PLD` cannot be executed conditionally.

Write-back Clearing bit[24] (the P bit) or setting bit[21] (the W bit) has UNPREDICTABLE results.

Data Aborts This instruction never signals a precise Data Abort generated by the VMSA MMU, PMSA MPU or by the rest of the memory system. Other memory system exceptions caused as a side-effect of this operation might be reported using an imprecise Data Abort or by some other exception mechanism.

Alignment There are no alignment restrictions on the address generated by `<addressing_mode>`. If an implementation contains a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), it must not generate an alignment exception for any `PLD` instruction.

A4.1.46 QADD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	0	0	Rn	Rd	SBZ	0	1	0	1	Rm

`QADD` (Saturating Add) performs integer addition. It saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs, `QADD` sets the Q flag in the CPSR.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rn>`	Specifies the register that contains the second operand.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + Rn, 32)
    if SignedDoesSat(Rm + Rn, 32) then
        Q Flag = 1
```

Usage

As well as performing saturated integer and Q31 additions, you can use `QADD` in combination with an `SMUL<x><y>`, `SMULW<y>`, or `SMULL` instruction to produce multiplications of Q15 and Q31 numbers. Three examples are:

- To multiply the Q15 numbers in the bottom halves of R0 and R1 and place the Q31 result in R2, use:

  ```
  SMULBB  R2, R0, R1
  QADD    R2, R2, R2
  ```

- To multiply the Q31 number in R0 by the Q15 number in the top half of R1 and place the Q31 result in R2, use:

  ```
  SMULWT  R2, R0, R1
  QADD    R2, R2, R2
  ```

- To multiply the Q31 numbers in R0 and R1 and place the Q31 result in R2, use:

  ```
  SMULL   R3, R2, R0, R1
  QADD    R2, R2, R2
  ```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.
Condition flags	`QADD` does not affect the N, Z, C, or V flags.

A4.1.47 QADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	0	0	1	Rm

`QADD16` performs two 16-bit integer additions. It saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

`QADD16` does not affect any flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  + Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[31:16], 16)
```

Usage

Use `QADD16` in similar ways to the `SADD16` instruction, but for signed saturated arithmetic. `QADD16` does not set the GE bits for use with `SEL`. See *SADD16* on page A4-119 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.48 QADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	1	0	0	1	Rm

`QADD8` performs four 8-bit integer additions. It saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

`QADD8` does not affect any flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   + Rm[7:0],   8)
    Rd[15:8]  = SignedSat(Rn[15:8]  + Rm[15:8],  8)
    Rd[23:16] = SignedSat(Rn[23:16] + Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] + Rm[31:24], 8)
```

Usage

Use `QADD8` in similar ways to the `SADD8` instruction, but for signed saturated arithmetic. `QADD8` does not set the GE bits for use with `SEL`. See *SADD8* on page A4-121 for more details.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.49 QADDSUBX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	0	1	1	Rm

`QADDSUBX` (Saturating Add and Subtract with Exchange) performs one 16-bit integer addition and one 16-bit subtraction. It saturates the results to the 16-bit signed integer range $-2^{15} \le x \le 2^{15} - 1$. `QADDSUBX` exchanges the two halfwords of the second operand before it performs the arithmetic.

`QADDSUBX` does not affect any flags.

Syntax

```
QADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:16] = SignedSat(Rn[31:16] + Rm[15:0],  16)
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[31:16], 16)
```

Usage

You can use `QADDSUBX` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
    QADDSUBX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra + i * Rb).

`QADDSUBX` does not set the Q flag, even if saturation occurs on either operation.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.50 QDADD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	0	0	Rn	Rd	SBZ	0	1	0	1	Rm

`QDADD` (Saturating Double and Add) doubles its second operand, then adds the result to its first operand.

Both the doubling and the addition have their results saturated to the 32-bit signed integer range $-2^{31} \le x \le 2^{31} - 1$.

If saturation occurs in either operation, the instruction sets the Q flag in the CPSR.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rn>`	Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated addition.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm + SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm + SignedSat(Rn*2, 32), 32) or
        SignedDoesSat(Rn*2, 32) then
        Q Flag = 1
```

Usage

The primary use for this instruction is to generate multiply-accumulate operations on Q15 and Q31 numbers, by placing it after an integer multiply instruction. Three examples are:

- To multiply the Q15 numbers in the top halves of R4 and R5 and add the product to the Q31 number in R6, use:

```
SMULTT  R0, R4, R5
QDADD   R6, R6, R0
```

- To multiply the Q15 number in the bottom half of R2 by the Q31 number in R3 and add the product to the Q31 number in R7, use:

```
SMULWB  R0, R3, R2
QDADD   R7, R7, R0
```

- To multiply the Q31 numbers in R2 and R3 and add the product to the Q31 number in R4, use:

```
SMULL   R0, R1, R2, R3
QDADD   R4, R4, R1
```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.
Condition flags	The `QDADD` instruction does not affect the N, Z, C, or V flags.

A4.1.51 QDSUB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	1	0	Rn	Rd	SBZ	0	1	0	1	Rm

`QDSUB` (Saturating Double and Subtract) doubles its second operand, then subtracts the result from its first operand.

Both the doubling and the subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs in either operation, `QDSUB` sets the Q flag in the CPSR.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rn>`	Specifies the register whose value is to be doubled, saturated, and used as the second operand for the saturated subtraction.

`Rm` and `Rn` are in reversed order in the assembler syntax, compared with the majority of ARM instructions.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - SignedSat(Rn*2, 32), 32)
    if SignedDoesSat(Rm - SignedSat(Rn*2, 32), 32) or
        SignedDoesSat(Rn*2, 32) then
        Q Flag = 1
```

Usage

The primary use for this instruction is to generate multiply-subtract operations on Q15 and Q31 numbers, by placing it after an integer multiply instruction. Three examples are:

- To multiply the Q15 numbers in the top half of R4 and the bottom half of R5, and subtract the product from the Q31 number in R6, use:

```
SMULTB  R0, R4, R5
QDSUB   R6, R6, R0
```

- To multiply the Q15 number in the bottom half of R2 by the Q31 number in R3 and subtract the product from the Q31 number in R7, use:

```
SMULWB  R0, R3, R2
QDSUB   R7, R7, R0
```

- To multiply the Q31 numbers in R2 and R3 and subtract the product from the Q31 number in R4, use:

```
SMULL   R0, R1, R2, R3
QDSUB   R4, R4, R1
```

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

Condition flags The `QDSUB` instruction does not affect the N, Z, C, or V flags.

A4.1.52 QSUB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	1	0	Rn	Rd	SBZ	0	1	0	1	Rm

`QSUB` (Saturating Subtract) performs integer subtraction. It saturates the result to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$.

If saturation occurs, `QSUB` sets the Q flag in the CPSR.

Syntax

```
QSUB{<cond>}  <Rd>, <Rm>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rn>`	Specifies the register that contains the second operand.

`Rm` and `Rn` are in reversed order in the assembler syntax, compared with the majority of ARM instructions.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = SignedSat(Rm - Rn, 32)
    if SignedDoesSat(Rm - Rn, 32) then
        Q Flag = 1
```

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.
Condition flags	`QSUB` does not affect the N, Z, C, or V flags.

A4.1.53 QSUB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	1	1	1	Rm

`QSUB16` performs two 16-bit subtractions. It saturates the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$.

`QSUB16` does not affect any flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[31:16], 16)
```

Usage

Use `QSUB16` in similar ways to the `SSUB16` instruction, but for signed saturated arithmetic. `QSUB16` does not set the GE bits for use with `SEL`. See *SSUB16* on page A4-180 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.54 QSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	1	1	1	1	Rm

`QSUB8` performs four 8-bit subtractions. It saturates the results to the 8-bit signed integer range $-2^7 \leq x \leq 2^7 - 1$.

`QSUB8` does not affect any flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = SignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = SignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = SignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = SignedSat(Rn[31:24] - Rm[31:24], 8)
```

Usage

Use `QSUB8` in similar ways to `SSUB8`, but for signed saturated arithmetic. `QSUB8` does not set the GE bits for use with `SEL`. See *SSUB8* on page A4-182 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.55 QSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	0	Rn	Rd	SBO	0	1	0	1	Rm

`QSUBADDX` (Saturating Subtract and Add with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction, saturating the results to the 16-bit signed integer range $-2^{15} \leq x \leq 2^{15} - 1$. It exchanges the two halfwords of the second operand before it performs the arithmetic. `QSUBADDX` does not affect any flags.

Syntax

```
QSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:16] = SignedSat(Rn[31:16] - Rm[15:0],  16)
    Rd[15:0]  = SignedSat(Rn[15:0]  + Rm[31:16], 16)
```

Usage

You can use `QSUBADDX` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
QSUBADDX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra – i * Rb).

`QSUBADDX` does not set the Q flag, even if saturation occurs on either operation.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.56 REV

31 28	27 23	22	21 20	19 16	15 12	11 8	7	6 4	3 0
cond	0 1 1 0 1	0	1 1	SBO	Rd	SBO	0	0 1 1	Rm

REV (Byte-Reverse Word) reverses the byte order in a 32-bit register.

Syntax

```
REV{<cond>} Rd, Rm
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:24] = Rm[ 7: 0]
    Rd[23:16] = Rm[15: 8]
    Rd[15: 8] = Rm[23:16]
    Rd[ 7: 0] = Rm[31:24]
```

Usage

Use REV to convert 32-bit big-endian data into little-endian data, or 32-bit little-endian data into big-endian data.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.57 REV16

31 28	27 23	22	21 20	19 16	15 12	11 8	7	6 4	3 0
cond	0 1 1 0 1	0	1 1	SBO	Rd	SBO	1	0 1 1	Rm

REV16 (Byte-Reverse Packed Halfword) reverses the byte order in each 16-bit halfword of a 32-bit register.

Syntax

```
REV16{<cond>} Rd, Rm
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    Rd[31:24] = Rm[23:16]
    Rd[23:16] = Rm[31:24]
```

Usage

Use `REV16` to convert 16-bit big-endian data into little-endian data, or 16-bit little-endian data into big-endian data.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.58 REVSH

31 28	27 23	22	21 20	19 16	15 12	11 8	7	6 4	3 0
cond	0 1 1 0 1	1	1 1	SBO	Rd	SBO	1	0 1 1	Rm

`REVSH` (Byte-Reverse Signed Halfword) reverses the byte order in the lower 16-bit halfword of a 32-bit register, and sign extends the result to 32-bits.

Syntax

```
REVSH{<cond>} Rd, Rm
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.

Architecture version

Version 6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15: 8] = Rm[ 7: 0]
    Rd[ 7: 0] = Rm[15: 8]
    if Rm[7] == 1 then
        Rd[31:16] = 0xFFFF
    else
        Rd[31:16] = 0x0000
```

Usage

Use `REVSH` to convert either:

- 16-bit signed big-endian data into 32-bit signed little-endian data
- 16-bit signed little-endian data into 32-bit signed big-endian data.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.59 RFE

31 30 29 28	27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8	7 0
1 1 1 1	1 0 0	P	U	0	W	1	Rn	SBZ	1 0 1 0	SBZ

`RFE` (Return From Exception) loads the PC and the CPSR from the word at the specified address and the following word respectively.

Syntax

```
RFE<addressing_mode> <Rn>{!}
```

where:

`<addressing_mode>`

Is similar to the `<addressing_mode>` in `LDM` and `STM` instructions, see *Addressing Mode 4 - Load and Store Multiple* on page A5-41, but with the following differences:

- The number of registers to load is 2.
- The register list is `{PC, CPSR}`.

`<Rn>` Specifies the base register to be used by `<addressing_mode>`. If R15 is specified as the base register, the result is UNPREDICTABLE.

`!` If present, sets the W bit. This causes the instruction to write a modified value back to its base register, in a manner similar to that specified for *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If `!` is omitted, the W bit is 0 and the instruction does not change the base register.

Architecture version

Version 6 and above.

Exceptions

Data Abort.

Usage

While `RFE` supports different base registers, a general usage case is where Rn == sp (the stack pointer), held in R13. The instruction can then be used as the return method associated with instructions `SRS` and `CPS`. See *New instructions to improve exception handling* on page A2-28 for more details.

Operation

```
address = start_address
value = Memory[address,4]
If InAPrivilegedMode() then
    CPSR = Memory[address+4,4]
else
    UNPREDICTABLE
PC = value

assert end_address == address + 8
```

where `start_address` and `end_address` are determined as described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41, except that `Number_Of_Set_Bits_in(register_list)` evaluates to 2, rather than depending on bits[15:0] of the instruction.

Notes

Data Abort For details of the effects of this instruction if a Data Abort occurs, see *Data Abort (data access memory abort)* on page A2-21.

Non word-aligned addresses

In ARMv6, an address with bits[1:0] != 0b00 causes an alignment exception if the CP15 register 1 bits U==1 or A==1, otherwise `RFE` behaves as if bits[1:0] are 0b00.

In earlier implementations, if they include a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), an address with bits[1:0] != 0b00 causes an alignment exception if the CP15 register 1 bit A==1, otherwise `RFE` behaves as if bits[1:0] are 0b00.

Time order The time order of the accesses to individual words of memory generated by `RFE` is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

User mode `RFE` is UNPREDICTABLE in User mode.

Condition Unlike most other ARM instructions, `RFE` cannot be executed conditionally.

ARM/Thumb State transfers

If the CPSR T bit as loaded is 0 and bit[1] of the value loaded into the PC is 1, the results are UNPREDICTABLE because it is not possible to branch to an ARM instruction at a non word-aligned address.

A4.1.60 RSB

31–28	27	26	25	24	23	22	21	20	19–16	15–12	11–0
cond	0	0	I	0	0	1	1	S	Rn	Rd	shifter_operand

`RSB` (Reverse Subtract) subtracts a value from a second value.

The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the subtraction. This is the reverse of the normal order of operands in ARM assembler language.

`RSB` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the second operand.

`<shifter_operand>`

Specifies the first operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `RSB`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn)
        V Flag = OverflowFrom(shifter_operand - Rn)
```

Usage

The following instruction stores the negation (twos complement) of `Rx` in `Rd`:

```
    RSB Rd, Rx, #0
```

You can perform constant multiplication (of `Rx`) by 2^n-1 (into `Rd`) with:

```
    RSB Rd, Rx, Rx, LSL #n
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: `SBC` and `RSC` use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.61 RSC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 1 1 1	S	Rn	Rd	shifter_operand

`RSC` (Reverse Subtract with Carry) subtracts one value from another, taking account of any borrow from a preceding less significant subtraction. The normal order of the operands is reversed, to allow subtraction from a shifted register value, or from an immediate value.

`RSC` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the second operand.

`<shifter_operand>`

Specifies the first operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `RSC`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = shifter_operand - Rn - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(shifter_operand - Rn - NOT(C Flag))
        V Flag = OverflowFrom(shifter_operand - Rn - NOT(C Flag))
```

Usage

Use `RSC` to synthesize multi-word subtraction, in cases where you need the order of the operands reversed to allow subtraction from a shifted register value, or from an immediate value.

Example

You can negate the 64-bit value in R0,R1 using the following sequence (R0 holds the least significant word), which stores the result in R2,R3:

```
    RSBS    R2,R0,#0
    RSC     R3,R1,#0
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: `SBC` and `RSC` use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.62 SADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	0	0	1	Rm

`SADD16` (Signed Add) performs two 16-bit signed integer additions. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if sum >= 0 then 0b11 else 0
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
```

Usage

Use the `SADD16` instruction to speed up operations on arrays of halfword data. For example, consider the instruction sequence:

```
LDR     R3, [R0], #4
LDR     R5, [R1], #4
SADD16  R3, R3, R5
STR     R3, [R2], #4
```

This performs the same operations as the instruction sequence:

```
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
LDRH    R3, [R0], #2
LDRH    R4, [R1], #2
ADD     R3, R3, R4
STRH    R3, [R2], #2
```

The first sequence uses half as many instructions and typically half as many cycles as the second sequence.

You can also use `SADD16` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SADD16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra + Rb.

`SADD16` sets the GE flags according to the results of each addition. You can use these in a following `SEL` instruction. See *SEL* on page A4-127.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.63 SADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	1	0	0	1	Rm

`SADD8` performs four 8-bit signed integer additions. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]     /* Signed addition */
    Rd[7:0]   = sum[7:0]
    GE[0]     = if sum >= 0 then 1 else 0
    sum       = Rn[15:8] + Rm[15:8]   /* Signed addition */
    Rd[15:8]  = sum[7:0]
    GE[1]     = if sum >= 0 then 1 else 0
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16] = sum[7:0]
    GE[2]     = if sum >= 0 then 1 else 0
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[7:0]
    GE[3]     = if sum >= 0 then 1 else 0
```

Usage

Use SADD8 to speed up operations on arrays of byte data. This is similar to the way you can use the SADD16 instruction. See the usage subsection for *SADD16* on page A4-119 for details.

SADD8 sets the GE flags according to the results of each addition. You can use these in a following SEL instruction, see *SEL* on page A4-127.

Notes

Use of R15	Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.64 SADDSUBX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 0 1	Rn	Rd	SBO	0 0 1 1	Rm

`SADDSUBX` (Signed Add and Subtract with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction. It exchanges the two halfwords of the second operand before it performs the arithmetic. It sets the GE bits in the CPSR according to the results of the additions.

Syntax

```
SADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]     /* Signed addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if sum >= 0 then 0b11 else 0
    diff      = Rn[15:0] - Rm[31:16]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
```

Usage

You can use SADDSUBX for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SADDSUBX  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra + (i * Rb).

SADDSUBX sets the GE flags according to the results the operation. You can use these in a following SEL instruction, see *SEL* on page A4-127.

Notes

Use of R15	Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.65 SBC

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	0 1 1 0	S	Rn	Rd	shifter_operand

`SBC` (Subtract with Carry) subtracts the value of its second operand and the value of NOT(Carry flag) from the value of its first operand. The first operand comes from a register. The second operand can be either an immediate value or a value from a register, and can be shifted before the subtraction.

Use `SBC` to synthesize multi-word subtraction.

`SBC` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `SBC`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand - NOT(C Flag)
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand - NOT(C Flag))
        V Flag = OverflowFrom(Rn - shifter_operand - NOT(C Flag))
```

Usage

If register pairs R0,R1 and R2,R3 hold 64-bit values (R0 and R2 hold the least significant words), the following instructions leave the 64-bit difference in R4,R5:

```
    SUBS    R4,R0,R2
    SBC     R5,R1,R3
```

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: `SBC` and `RSC` use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.66 SEL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	0	0	Rn	Rd	SBO	1	0	1	1	Rm

`SEL` (Select) selects each byte of its result from either its first operand or its second operand, according to the values of the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = if GE[0] == 1 then Rn[7:0]   else Rm[7:0]
    Rd[15:8]  = if GE[1] == 1 then Rn[15:8]  else Rm[15:8]
    Rd[23:16] = if GE[2] == 1 then Rn[23:16] else Rm[23:16]
    Rd[31:24] = if GE[3] == 1 then Rn[31:24] else Rm[31:24]
```

Usage

Use `SEL` after instructions such as `SADD8`, `SADD16`, `SSUB8`, `SSUB16`, `UADD8`, `UADD16`, `USUB8`, `USUB16`, `SADDSUBX`, `SSUBADDX`, `UADDSUBX` and `USUBADDX`, that set the GE flags. For example, the following sequence of instructions sets each byte of Rd equal to the unsigned minimum of the corresponding bytes of Ra and Rb:

```
USUB8  Rd, Ra, Rb
SEL    Rd, Rb, Ra
```

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.67 SETEND

31 30 29 28	27 26 25 24 23 22 21 20	19 18 17	16	15 10	9	8	7 4	3 0
1 1 1 1	0 0 0 1 0 0 0 0	0 0 0	1	SBZ	E	SBZ	0 0 0 0	SBZ

`SETEND` modifies the CPSR E bit, without changing any other bits in the CPSR.

Syntax

`SETEND <endian_specifier>`

where:

`<endian_specifier>`

Is one of:

`BE`	Sets the E bit in the instruction. This sets the CPSR E bit.
`LE`	Clears the E bit in the instruction. This clears the CPSR E bit.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
CPSR = CPSR with specified E bit modification
```

Usage

Use `SETEND` to change the byte order for data accesses. You can use `SETEND` to increase the efficiency of access to a series of big-endian data fields in an otherwise little-endian application, or to a series of little-endian data fields in an otherwise big-endian application.

Notes

Condition Unlike most other ARM instructions, `SETEND` cannot be executed conditionally.

A4.1.68 SHADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	0	0	1	Rm

`SHADD16` (Signed Halving Add) performs two 16-bit signed integer additions, and halves the results. It has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0] /* Signed addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Signed addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use `SHADD16` for similar purposes to `SADD16` (see *SADD16* on page A4-119). `SHADD16` averages the operands. `It` does not set any flags, as overflow is not possible.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.69 SHADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	1	0	0	1	Rm

`SHADD8` performs four 8-bit signed integer additions, and halves the results. It has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0] /* Signed addition */
    Rd[7:0]  = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8] /* Signed addition */
    Rd[15:8] = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16] /* Signed addition */
    Rd[23:16]  = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24] /* Signed addition */
    Rd[31:24] = sum[8:1]
```

Usage

Use `SHADD8` similar purposes to `SADD16` (see *SADD16* on page A4-119). `SHADD8` averages the operands. It does not set any flags, as overflow is not possible.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.70 SHADDSUBX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 0 1 1	Rn	Rd	SBO	0 0 1 1	Rm

`SHADDSUBX` (Signed Halving Add and Subtract with Exchange) performs one 16-bit signed integer addition and one 16-bit signed integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

`SHADDSUBX` has no effect on the GE flags.

Syntax

```
SHADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]     /* Signed addition */
    Rd[31:16] = sum[16:1]
    diff      = Rn[15:0] - Rm[31:16]     /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
```

Usage

Use `SHADDSUBX` for similar purposes to `SADDSUBX`, but when you want the results halved. See *SADDSUBX* on page A4-123 for further details.

`SHADDSUBX` does not set any flags, as overflow is not possible.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.71 SHSUB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	1	1	1	Rm

SHSUB16 (Signed Halving Subtract) performs two 16-bit signed integer subtractions, and halves the results.

SHSUB16 has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use `SHSUB16` to speed up operations on arrays of halfword data. This is similar to the way you can use `SADD16`. See the usage subsection for *SADD16* on page A4-119 for details.

You can also use `SHSUB16` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SHSUB16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = (Ra - Rb)/2.

`SHSUB16` does not set any flags, as overflow is not possible.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.72 SHSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	1	1	1	1	Rm

`SHSUB8` performs four 8-bit signed integer subtractions, and halves the results.

`SHSUB8` has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]      /* Signed subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]    /* Signed subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]  /* Signed subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]  /* Signed subtraction */
    Rd[31:24] = diff[8:1]
```

Usage

Use `SHSUB8` to speed up operations on arrays of byte data. This is similar to the way you can use `SADD16` to speed up operations on halfword data. See the usage subsection for *SADD16* on page A4-119 for details.

`SHSUB8` does not set any flags, as overflow is not possible.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.73 SHSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	1	1	Rn	Rd	SBO	0	1	0	1	Rm

`SHSUBADDX` (Signed Halving Subtract and Add with Exchange) performs one 16-bit signed integer subtraction and one 16-bit signed integer addition, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

`SHSUBADDX` has no effect on the GE flags.

Syntax

```
SHSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]     /* Signed subtraction */
    Rd[31:16] = diff[16:1]
    sum       = Rn[15:0]  + Rm[31:16]    /* Signed addition */
    Rd[15:0]  = sum[16:1]
```

Usage

Use SHSUBADDX for similar purposes to SSUBADDX, but when you want the results halved. See *SSUBADDX* on page A4-184 for further details.

SHSUBADDX does not set any flags, as overflow is not possible.

Notes

Use of R15	Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.74 SMLA<x><y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	0	0	Rd	Rn	Rs	1	y	x	0	Rm

`SMLA<x><y>` (Signed multiply-accumulate `BB`, `BT`, `TB`, and `TT`) performs a signed multiply-accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is added to a 32-bit accumulate value and the result is written to the destination register.

If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the CPSR. It is not possible for overflow to occur during the multiplication.

Syntax

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

where:

`<x>`	Specifies which half of the source register `<Rm>` is used as the first multiply operand. If `<x>` is `B`, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rm>` is used. If `<x>` is `T`, then x == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rm>` is used.
`<y>`	Specifies which half of the source register `<Rs>` is used as the second multiply operand. If `<y>` is `B`, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rs>` is used. If `<y>` is `T`, then y == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rs>` is used.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the source register whose bottom or top half (selected by `<x>`) is the first multiply operand.
`<Rs>`	Specifies the source register whose bottom or top half (selected by `<y>`) is the second multiply operand.
`<Rn>`	Specifies the register which contains the accumulate value.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (operand1 * operand2) + Rn
    if OverflowFrom((operand1 * operand2) + Rn) then
        Q Flag = 1
```

Usage

In addition to its straightforward uses for integer multiply-accumulates, these instructions sometimes provide a faster alternative to Q15 × Q15 + Q31 → Q31 multiply-accumulates synthesized from `SMUL<x><y>` and `QDADD` instructions. The main circumstances under which this is possible are:

- if it is known that saturation and/or overflow cannot occur during the calculation
- if saturation and/or overflow can occur during the calculation but the Q flag is going to be used to detect this and take remedial action if it does occur.

For example, the following code produces the dot product of the four Q15 numbers in R0 and R1 by the four Q15 numbers in R2 and R3:

```
SMULBB  R4, R0, R2
QADD    R4, R4, R4
SMULTT  R5, R0, R2
QDADD   R4, R4, R5
SMULBB  R5, R1, R3
QDADD   R4, R4, R5
SMULTT  R5, R1, R3
QDADD   R4, R4, R5
```

In the absence of saturation, the following code provides a faster alternative:

```
SMULBB  R4, R0, R2
SMLATT  R4, R0, R2, R4
SMLABB  R4, R1, R3, R4
SMLATT  R4, R1, R3, R4
QADD    R4, R4, R4
```

Furthermore, if saturation and/or overflow occurs in this second sequence, it sets the Q flag. This allows remedial action to be taken, such as scaling down the data values and repeating the calculation.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.
Condition flags	The `SMLA<x><y>` instructions do not affect the N, Z, C, or V flags.

A4.1.75 SMLAD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	0	0	0	Rd	Rn	Rs	0	0	X	1	Rm

`SMLAD` (Signed Multiply Accumulate Dual) performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications.

Syntax

```
SMLAD{X}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`X`	Sets the X bit of the instruction to 1, and the multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0, and the multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rs>`	Specifies the register that contains the second operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = Rn + product1 + product2
    if OverflowFrom(Rn + product1 + product2) then
        Q flag = 1
```

Usage

Use SMLAD to accumulate the sums of products of 16-bit data, with a 32-bit accumulator. This instruction enables you to do this at approximately twice the speed otherwise possible. This is useful in many applications, for example in filters.

You can use the X option for calculating the imaginary part for similar filters acting on complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15 Specifying R15 for register <Rd>, <Rm>, or <Rs> has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register <Rn>.

Encoding If the <Rn> field of the instruction contains 0b1111, the instruction is an SMUAD instruction instead, see *SMUAD* on page A4-164.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

N, Z, C and V flags The SMLAD instruction leaves the N, Z, C and V flags unchanged.

A4.1.76 SMLAL

31 28	27 26 25 24 23 22 21	20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 0 0 0 1 1 1	S	RdHi	RdLo	Rs	1 0 0 1	Rm

`SMLAL` (Signed Multiply Accumulate Long) multiplies two signed 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

`SMLAL` can optionally update the condition code flags, based on the result.

Syntax

```
SMLAL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Supplies the lower 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the upper 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo /* Signed multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`SMLAL` multiplies signed variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15 Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Operand restriction `<RdHi>` and `<RdLo>` must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either `<RdHi>` and `<Rm>`, or `<RdLo>` and `<Rm>`, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags `SMLALS` is defined to leave the C and V flags unchanged in ARMv5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after an `SMLALS` instruction.

A4.1.77 SMLAL<x><y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	0	0	RdHi	RdLo	Rs	1	y	x	0	Rm

`SMLAL<x><y>` (Signed Multiply-Accumulate Long `BB`, `BT`, `TB`, and `TT`) performs a signed multiply-accumulate operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and added to the 64-bit accumulate value held in `<RdHi>` and `<RdLo>`, and the result is written back to `<RdHi>` and `<RdLo>`.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64}.

Syntax

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

where:

`<x>`	Specifies which half of the source register `<Rm>` is used as the first multiply operand. If `<x>` is `B`, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rm>` is used. If `<x>` is `T`, then x == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rm>` is used.
`<y>`	Specifies which half of the source register `<Rs>` is used as the second multiply operand. If `<y>` is `B`, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rs>` is used. If `<y>` is `T`, then y == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rs>` is used.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<RdLo>`	Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.
`<RdHi>`	Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.
`<Rm>`	Specifies the source register whose bottom or top half (selected by `<x>`) is the first multiply operand.
`<Rs>`	Specifies the source register whose bottom or top half (selected by `<y>`) is the second multiply operand.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    RdLo = RdLo + (operand1 * operand2)
    RdHi = RdHi + (if (operand1*operand2) < 0 then 0xFFFFFFFF else 0)
                + CarryFrom(RdLo + (operand1 * operand2))
```

Usage

These instructions allow a long sequence of multiply-accumulates of signed 16-bit integers or Q15 numbers to be performed, with sufficient *guard bits* to ensure that the result cannot overflow the 64-bit destination in practice. It would take more than 2^{33} consecutive multiply-accumulates to cause such overflow.

If the overall calculation does not overflow a signed 32-bit number, then `<RdLo>` holds the result of the calculation.

A simple test to determine whether such a calculation has overflowed `<RdLo>` is to execute the instruction:

```
  CMP     <RdHi>, <RdLo>, ASR #31
```

at the end of the calculation. If the Z flag is set, `<RdLo>` holds an accurate final result. If the Z flag is clear, the final result has overflowed a signed 32-bit destination.

Notes

Use of R15	Specifying R15 for register `<RdLo>`, `<RdHi>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	If `<RdLo>` and `<RdHi>` are the same register, the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Condition flags	The `SMLAL<x><y>` instructions do not affect the N, Z, C, V, or Q flags.

A4.1.78 SMLALD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	1	0	0	RdHi	RdLo	Rs	0	0	X	1	Rm

`SMLALD` (Signed Multiply Accumulate Long Dual) performs two signed 16 x 16-bit multiplications. It adds the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Syntax

```
SMLALD{X}{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`X`	Sets the X bit of the instruction to 1, and the multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0, and the multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<RdLo>`	Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.
`<RdHi>`	Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    accvalue[31:0]  = RdLo
    accvalue[63:32] = RdHi
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    result = accvalue + product1 + product2   /* Signed addition */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

Usage

Use SMLALD in similar ways to SMLAD, but when you require a 64-bit accumulator instead of a 32-bit accumulator. On most implementations, this runs more slowly. See the usage section for *SMLAD* on page A4-144 for further details.

Notes

Use of R15	Specifying R15 for register <RdLo>, <RdHi>, <Rm>, or <Rs> has UNPREDICTABLE results.
Operand restriction	If <RdLo> and <RdHi> are the same register, the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	SMLALD leaves all the flags unchanged.

A4.1.79 SMLAW<y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	1	0	Rd	Rn	Rs	1	y	0	0	Rm

`SMLAW<y>` (Signed Multiply-Accumulate Word `B` and `T`) performs a signed multiply-accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored. If overflow occurs during the addition of the accumulate value, the instruction sets the Q flag in the CPSR. No overflow can occur during the multiplication, because of the use of the top 32 bits of the 48-bit product.

Syntax

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

where:

`<y>`	Specifies which half of the source register `<Rs>` is used as the second multiply operand. If `<y>` is `B`, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rs>` is used. If `<y>` is `T`, then y == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rs>` is used.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the source register which contains the 32-bit first multiply operand.
`<Rs>`	Specifies the source register whose bottom or top half (selected by `<y>`) is the second multiply operand.
`<Rn>`	Specifies the register which contains the accumulate value.

Architecture version

Version 5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] + Rn     /* Signed multiplication */
    if OverflowFrom((Rm * operand2)[47:16] + Rn) then
        Q Flag = 1
```

Usage

In addition to their straightforward uses for integer multiply-accumulates, these instructions sometimes provide a faster alternative to Q31 × Q15 + Q31 → Q31 multiply-accumulates synthesized from `SMULW<y>` and `QDADD` instructions. The circumstances under which this is possible and the benefits it provides are very similar to those for the `SMLA<x><y>` instructions. See *Usage* on page A4-143 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Condition flags	The `SMLAW<y>` instructions do not affect the N, Z, C, or V flags.

A4.1.80 SMLSD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	0	0	0	Rd	Rn	Rs	0	1	X	1	Rm

`SMLSD` (Signed Multiply Subtract accumulate Dual) performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 32-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets the Q flag if the accumulate operation overflows. Overflow cannot occur during the multiplications or subtraction.

Syntax

```
SMLSD{X}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`X`	Sets the X bit of the instruction to 1, and the multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0, and the multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    diffofproducts = product1 - product2      /* Signed subtraction */
    Rd = Rn + diffofproducts
    if OverflowFrom(Rn + diffofproducts) then
        Q flag = 1
```

Usage

You can use `SMLSD` for calculating the real part in filters with 32-bit accumulators, acting on complex numbers with 16-bit real and 16-bit imaginary parts.

See also the usage section for *SMLAD* on page A4-144.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is an `SMUSD` instruction instead, see *SMUSD* on page A4-172.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

N, Z, C and V flags `SMLSD` leaves the N, Z, C and V flags unchanged.

A4.1.81 SMLSLD

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	1	0	0	RdHi	RdLo	Rs	0	1	X	1	Rm

`SMLSLD` (Signed Multiply Subtract accumulate Long Dual) performs two signed 16 x 16-bit multiplications. It adds the difference of the products to a 64-bit accumulate operand.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Syntax

```
SMLSLD{X}{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`X`	Sets the X bit of the instruction to 1, and the multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0, and the multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<RdLo>`	Supplies the lower 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the lower 32 bits of the 64-bit result.
`<RdHi>`	Supplies the upper 32 bits of the 64-bit accumulate value to be added to the product, and is the destination register for the upper 32 bits of the 64-bit result.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    accvalue[31:0]  = RdLo
    accvalue[63:32] = RdHi
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    result = accvalue + product1 - product2   /* Signed subtraction */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

Usage

The instruction has similar uses to those of the `SMLSD` instruction (see the Usage section for *SMLSD* on page A4-154), but when 64-bit accumulators are required rather than 32-bit accumulators. On most implementations, the resulting filter will not run as fast as a version using `SMLSD`, but it has many more guard bits against overflow.

See also the usage section for *SMLAD* on page A4-144.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	If `<RdLo>` and `<RdHi>` are the same register, the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	`SMLSD` leaves all the flags unchanged.

A4.1.82 SMMLA

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	1	0	1	Rd	Rn	Rs	0	0	R	1	Rm

`SMMLA` (Signed Most significant word Multiply Accumulate) multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and adds an accumulate value.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

Syntax

```
SMMLA{R}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`R`	Sets the R bit of the instruction to 1. The multiplication is rounded. If the R is omitted, sets the R bit to 0. The multiplication is truncated.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    value = Rm * Rs                    /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) + value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) + value)[63:32]
```

Usage

Provides fast multiplication for 32-bit fractional arithmetic. For example, the multiplies take two Q31 inputs and give a Q30 result (where Q*n* is a fixed point number with *n* bits of fraction).

A short discussion on fractional arithmetic is provided in *Saturated Q15 and Q31 arithmetic* on page A2-69.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is an `SMMUL` instruction instead, see *SMMUL* on page A4-162.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

Flags `SMMLA` leaves all the flags unchanged.

A4.1.83 SMMLS

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	1	0	1	Rd	Rn	Rs	1	1	R	1	Rm

`SMMLS` (Signed Most significant word Multiply Subtract) multiplies two signed 32-bit values, extracts the most significant 32 bits of the result, and subtracts it from an accumulate value.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the accumulated value before the high word is extracted.

Syntax

```
SMMLS{R}{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`R`	Sets the R bit of the instruction to 1. The multiplication is rounded. If the R is omitted, sets the R bit to 0. The multiplication is truncated.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    value = Rm * Rs                    /* Signed multiplication */
    if R == 1 then
        Rd = ((Rn<<32) - value + 0x80000000)[63:32]
    else
        Rd = ((Rn<<32) - value)[63:32]
```

Usage

Provides fast multiplication for 32-bit fractional arithmetic. For example, the multiplies take two Q31 inputs and give a Q30 result (where Q*n* is a fixed point number with *n* bits of fraction).

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, `<Rs>`, or `<Rn>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	`SMMLS` leaves all the flags unchanged.

A4.1.84 SMMUL

31 28	27 26 25 24 23 22 21 20	19 16	15 14 13 12	11 8	7 6	5	4	3 0
cond	0 1 1 1 0 1 0 1	Rd	1 1 1 1	Rs	0 0	R	1	Rm

`SMMUL` (Signed Most significant word Multiply) multiplies two signed 32-bit values, and extracts the most significant 32 bits of the result.

Optionally, you can specify that the result is rounded instead of being truncated. In this case, the constant `0x80000000` is added to the product before the high word is extracted.

Syntax

```
SMMUL{R}{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

`R`	Sets the R bit of the instruction to 1. The multiplication is rounded. If the R is omitted, sets the R bit to 0. The multiplication is truncated.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if R == 1 then
        value = Rm * Rs + 0x80000000  /* Signed multiplication */
    else
        value = Rm * Rs               /* Signed multiplication */
    Rd = value[63:32]
```

Usage

You can use `SMMUL` in combination with `QADD` or `QDADD` to perform Q31 multiplies and multiply-accumulates. It has two advantages over a combination of `SMULL` with `QADD` or `QDADD`:

- you can round the product
- no scratch register is required for the least significant half of the product.

You can also use `SMMUL` in optimized Fast Fourier Transforms and similar algorithms.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	`SMMUL` leaves all the flags unchanged.

A4.1.85 SMUAD

31 28	27	26	25	24	23	22	21	20	19 16	15	14	13	12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	0	0	0	Rd	1	1	1	1	Rs	0	0	X	1	Rm

`SMUAD` (Signed Dual Multiply Add) performs two signed 16 x 16-bit multiplications. It adds the products together, giving a 32-bit result.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets the Q flag if the addition overflows. The multiplications cannot overflow.

Syntax

```
SMUAD{X}{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

`X`	Sets the X bit of the instruction to 1, and the multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0, and the multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rs>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = product1 + product2
    if OverflowFrom(product1 + product2) then
        Q flag = 1
```

Usage

Use `SMUAD` for the first pair of multiplications in a sequence that uses the `SMLAD` instruction for the following multiplications, see *SMLAD* on page A4-144.

You can use the X option for calculating the imaginary part of a product of complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
N, Z, C and V flags	`SMUAD` leaves the N, Z, C and V flags unchanged.

A4.1.86 SMUL<x><y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	1	0	Rd	SBZ	Rs	1	y	x	0	Rm

`SMUL<x><y>` (Signed Multiply `BB`, `BT`, `TB`, or `TT`) performs a signed multiply operation. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. No overflow is possible during this instruction.

Syntax

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

where:

`<x>`	Specifies which half of the source register `<Rm>` is used as the first multiply operand. If `<x>` is `B`, then x == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rm>` is used. If `<x>` is `T`, then x == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rm>` is used.
`<y>`	Specifies which half of the source register `<Rs>` is used as the second multiply operand. If `<y>` is `B`, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of `<Rs>` is used. If `<y>` is `T`, then y == 1 in the instruction encoding and the top half (bits[31:16]) of `<Rs>` is used.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the source register whose bottom or top half (selected by `<x>`) is the first multiply operand.
`<Rs>`	Specifies the source register whose bottom or top half (selected by `<y>`) is the second multiply operand.

Architecture version

ARMv5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (x == 0) then
        operand1 = SignExtend(Rm[15:0])
    else /* x == 1 */
        operand1 = SignExtend(Rm[31:16])

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = operand1 * operand2
```

Usage

In addition to its straightforward uses for integer multiplies, this instruction can be used in combination with `QADD`, `QDADD`, and `QDSUB` to perform multiplies, multiply-accumulates, and multiply-subtracts on Q15 numbers. See the *Usage* sections on page A4-93, page A4-100, and page A4-102 for examples.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Condition flags	`SMUL<x><y>` does not affect the N, Z, C, V, or Q flags.

A4.1.87 SMULL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	1	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

`SMULL` (Signed Multiply Long) multiplies two 32-bit signed values to produce a 64-bit result.

`SMULL` can optionally update the condition code flags, based on the 64-bit result.

Syntax

```
SMULL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32] /* Signed multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

SMULL multiplies signed variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15 Specifying R15 for register <RdHi>, <RdLo>, <Rm>, or <Rs> has UNPREDICTABLE results.

Operand restriction <RdHi> and <RdLo> must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either <RdHi> and <Rm>, or <RdLo> and <Rm>, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the <Rs> operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags SMULLS is defined to leave the C and V flags unchanged in ARMv5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after an SMULLS instruction.

A4.1.88 SMULW<y>

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	1	0	Rd	SBZ	Rs	1	y	1	0	Rm

SMULW<y> (Signed Multiply Word B and T) performs a signed multiply operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity, with the latter being taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

No overflow is possible during this instruction.

Syntax

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

where:

<y>	Specifies which half of the source register <Rs> is used as the second multiply operand. If <y> is B, then y == 0 in the instruction encoding and the bottom half (bits[15:0]) of <Rs> is used. If <y> is T, then y == 1 in the instruction encoding and the top half (bits[31:16]) of <Rs> is used.
<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rm>	Specifies the source register which contains the 32-bit first operand.
<Rs>	Specifies the source register whose bottom or top half (selected by <y>) is the second operand.

Architecture version

ARMv5TE and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if (y == 0) then
        operand2 = SignExtend(Rs[15:0])
    else /* y == 1 */
        operand2 = SignExtend(Rs[31:16])

    Rd = (Rm * operand2)[47:16] /* Signed multiplication */
```

Usage

In addition to its straightforward uses for integer multiplies, this instruction can be used in combination with `QADD`, `QDADD`, and `QDSUB` to perform multiplies, multiply-accumulates and multiply-subtracts between Q31 and Q15 numbers. See the *Usage* sections on page A4-93, page A4-100, and page A4-102 for examples.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	`SMULW<y>` leaves all the flags unchanged.

A4.1.89 SMUSD

31 28	27	26	25	24	23	22	21	20	19 16	15	14	13	12	11 8	7	6	5	4	3 0
cond	0	1	1	1	0	0	0	0	Rd	1	1	1	1	Rs	0	1	X	1	Rm

`SMUSD` (Signed Dual Multiply Subtract) performs two signed 16 x 16-bit multiplications. It subtracts one product from the other, giving a 32-bit result.

Optionally, you can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

Overflow cannot occur.

Syntax

```
SMUSD{X}{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

`X`	Sets the X bit of the instruction to 1. The multiplications are bottom x top and top x bottom. If the X is omitted, sets the X bit to 0. The multiplications are bottom x bottom and top x top.
`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first multiply operand.
`<Rs>`	Specifies the register that contains the second multiply operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if X == 1 then
        operand2 = Rs Rotate_Right 16
    else
        operand2 = Rs
    product1 = Rm[15:0]  * operand2[15:0]     /* Signed multiplication */
    product2 = Rm[31:16] * operand2[31:16]    /* Signed multiplication */
    Rd = product1 - product2                  /* Signed subtraction */
```

Usage

You can use `SMUSD` for calculating the real part of a complex product of complex numbers with 16-bit real and 16-bit imaginary parts.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.
Flags	`SMUSD` leaves all the flags unchanged.

A4.1.90 SRS

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 12	11 10 9 8	7 5	4 0
1 1 1 1	1 0 0	P	U	1	W	0	1 1 0 1	SBZ	0 1 0 1	SBZ	mode

`SRS` (Store Return State) stores the R14 and SPSR of the current mode to the word at the specified address and the following word respectively. The address is determined from the banked version of R13 belonging to a specified mode.

Syntax

```
SRS<addressing_mode> #<mode>{!}
```

where:

`<addressing_mode>`

Is similar to the `<addressing_mode>` in `LDM` and `STM` instructions, see *Addressing Mode 4 - Load and Store Multiple* on page A5-41, but with the following differences:

- The base register, `Rn`, is the banked version of `R13` for the mode specified by `<mode>`, rather than the current mode.
- The number of registers to store is 2.
- The register list is `{R14, SPSR}`, with both `R14` and the `SPSR` being the versions belonging to the current mode.

`<mode>` Specifies the number of the mode whose banked register is used as the base register for `<addressing_mode>`. The mode number is the 5-bit encoding of the chosen mode in a PSR, as described in *The mode bits* on page A2-14.

`!` If present, sets the W bit. This causes the instruction to write a modified value back to its base register, in a manner similar to that specified for *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If `!` is omitted, the W bit is 0 and the instruction does not change the base register.

Architecture version

ARMv6 and above.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
address = start_address
Memory[address,4] = R14
if Shared(address) then         /* from ARMv6 */
    physical_address = TLB(address)
    ClearExclusiveByAddress(physical_address,processor_id,4)
if CurrentModeHasSPSR() then
    Memory[address+4,4] = SPSR
    if Shared(address+4) then    /* from ARMv6 */
        physical_address = TLB(address+4)
        ClearExclusiveByAddress(physical_address,processor_id,4)
else
    UNPREDICTABLE
assert end_address == address + 8
```

where start_address and end_address are determined as described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41, with the following modifications:

- Number_Of_Set_Bits_in(register_list) evaluates to 2, rather than depending on bits[15:0] of the instruction.
- Rn is the banked version of R13 belonging to the mode specified by the instruction, rather than being the version of R13 of the current mode.

Notes

Data Abort For details of the effects of this instruction if a Data Abort occurs, see *Data Abort (data access memory abort)* on page A2-21.

Non word-aligned addresses

In ARMv6, an address with bits[1:0] != 0b00 causes an alignment exception if CP15 register 1 bits U==1 or A==1. Otherwise, SRS behaves as if bits[1:0] are 0b00.

Time order The time order of the accesses to individual words of memory generated by SRS is not architecturally defined. Do not use this instruction on memory-mapped I/O locations where access order matters.

User and System modes

SRS is UNPREDICTABLE in User and System modes, because they do not have SPSRs.

Note

In User mode, SRS must not give access to any banked registers belonging to other modes. This would constitute a security hole.

Condition Unlike most other ARM instructions, SRS cannot be executed conditionally.

A4.1.91 SSAT

31 28	27 26 25 24 23 22 21	20 16	15 12	11 7	6	5 4	3 0
cond	0 1 1 0 1 0 1	sat_imm	Rd	shift_imm	sh	0 1	Rm

`SSAT` (Signed Saturate) saturates a signed value to a signed range. You can choose the bit position at which saturation occurs.

You can apply a shift to the value before the saturation occurs.

The Q flag is set if the operation saturates.

Syntax

```
SSAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<immed>`	Specifies the bit position for saturation, in the range 1 to 32. It is encoded in the `sat_imm` field of the instruction as `<immed>-1`.
`<Rm>`	Specifies the register that contains the signed value to be saturated.
`<shift>`	Specifies the optional shift. If present, it must be one of:

- `LSL #N`. `N` must be in the range 0 to 31.
  This is encoded as `sh` == 0 and `shift_imm` == `N`.
- `ASR #N`. `N` must be in the range 1 to 32. This is encoded as `sh` == 1 and either `shift_imm` == 0 for `N` == 32, or `shift_imm == N` otherwise.

If `<shift>` is omitted, `LSL #0` is used.

Return

The value returned in Rd is:

$-2^{(n-1)}$	if X is $< -2^{(n-1)}$
X	if $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$
$2^{(n-1)} - 1$	if $X > 2^{(n-1)} - 1$

where n is `<immed>`, and X is the shifted value from Rm.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = SignedSat(operand, sat_imm + 1)
    if SignedDoesSat(operand, sat_imm + 1) then
        Q Flag = 1
```

Usage

You can use SSAT in various DSP algorithms that require scaling and saturation of signed data.

Notes

Use of R15 Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.92 SSAT16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	1	0	sat_imm	Rd	SBO	0	0	1	1	Rm

`SSAT16` saturates two 16-bit signed values to a signed range. You can choose the bit position at which saturation occurs. The Q flag is set if either halfword operation saturates.

Syntax

```
SSAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<immed>`	Specifies the bit position for saturation. This lies in the range 1 to 16. It is encoded in the `sat_imm` field of the instruction as `<immed>-1`.
`<Rm>`	Specifies the register that contains the signed value to be saturated.

Return

The value returned in each half of Rd is:

$-2^{(n-1)}$	if X is $< -2^{(n-1)}$
X	if $-2^{(n-1)} <= X <= 2^{(n-1)} - 1$
$2^{(n-1)} - 1$	if $X > 2^{(n-1)} - 1$

where n is `<immed>`, and X is the value from the corresponding half of Rm.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = SignedSat(Rm[15:0],  sat_imm + 1)
    Rd[31:16] = SignedSat(Rm[31:16], sat_imm + 1)
    if SignedDoesSat(Rm[15:0],  sat_imm + 1)
                              OR SignedDoesSat(Rm[31:16], sat_imm + 1) then
        Q Flag = 1
```

Usage

You can use `SSAT16` in various DSP algorithms that require saturation of signed data.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.93 SSUB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	1	1	1	Rm

`SSUB16` (Signed Subtract) performs two 16-bit signed integer subtractions. It sets the GE bits in the CPSR according to the results of the subtractions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Signed subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if diff >= 0 then 0b11 else 0
    diff      = Rn[31:16] - Rm[31:16]  /* Signed subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if diff >= 0 then 0b11 else 0
```

Usage

Use `SSUB16` to speed up operations on arrays of halfword data. This is similar to the way you can use `SADD16`. See the usage subsection for *SADD16* on page A4-119 for details.

You can also use `SSUB16` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SSUB16  Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra - Rb.

`SSUB16` sets the GE flags according to the results of each subtraction. You can use these in a following `SEL` instruction. See *SEL* on page A4-127 for further information.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.94 SSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	1	1	1	1	Rm

`SSUB8` performs four 8-bit signed integer subtractions. It sets the GE bits in the CPSR according to the results of the subtractions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]      /* Signed subtraction */
    Rd[7:0]   = diff[7:0]
    GE[0]     = if diff >= 0 then 1 else 0
    diff      = Rn[15:8] - Rm[15:8]    /* Signed subtraction */
    Rd[15:8]  = diff[7:0]
    GE[1]     = if diff >= 0 then 1 else 0
    diff      = Rn[23:16] - Rm[23:16]  /* Signed subtraction */
    Rd[23:16] = diff[7:0]
    GE[2]     = if diff >= 0 then 1 else 0
    diff      = Rn[31:24] - Rm[31:24]  /* Signed subtraction */
    Rd[31:24] = diff[7:0]
    GE[3]     = if diff >= 0 then 1 else 0
```

Usage

Use `SSUB8` to speed up operations on arrays of byte data. This is similar to the way you can use `SADD16` to speed up operations on halfword data. See the usage subsection for *SADD16* on page A4-119 for details.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.95 SSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	0	0	1	Rn	Rd	SBO	0	1	0	1	Rm

`SSUBADDX` (Signed Subtract and Add with Exchange) performs one 16-bit signed integer subtraction and one 16-bit signed integer addition. It exchanges the two halfwords of the second operand before it performs the arithmetic.

`SSUBADDX` sets the GE bits in the CPSR according to the results.

Syntax

```
SSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]     /* Signed subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if diff >= 0 then 0b11 else 0
    sum       = Rn[15:0] + Rm[31:16]     /* Signed addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if sum >= 0 then 0b11 else 0
```

Usage

You can use `SSUBADDX` for operations on complex numbers that are held as pairs of 16-bit integers or Q15 numbers. If you hold the real and imaginary parts of a complex number in the bottom and top half of a register respectively, then the instruction:

```
SSUBADDX Rd, Ra, Rb
```

performs the complex arithmetic operation Rd = Ra - i * Rb.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.96 STC

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7 0
cond	1	1	0	P	U	N	W	0	Rn	CRd	cp_num	8_bit_word_offset

`STC` (Store Coprocessor) stores data from a coprocessor to a sequence of consecutive memory addresses. If no coprocessors indicate that they can execute the instruction, an Undefined Instruction exception is generated.

Syntax

```
STC{<cond>}{L}  <coproc>, <CRd>, <addressing_mode>
STC2{L}         <coproc>, <CRd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`STC2`	Causes the condition field of the instruction to be set to 0b1111. This provides additional opcode space for coprocessor designers. The resulting instructions can only be executed unconditionally.
`L`	Sets the N bit (bit[22]) in the instruction to 1 and specifies a long store (for example, double-precision instead of single-precision data transfer). If L is omitted, the N bit is 0 and the instruction specifies a short store.
`<coproc>`	Specifies the name of the coprocessor, and causes the corresponding coprocessor number to be placed in the cp_num field of the instruction. The standard generic coprocessor names are p0, p1, ..., p15.
`<CRd>`	Specifies the coprocessor source register.
`<addressing_mode>`	Is described in *Addressing Mode 5 - Load and Store Coprocessor* on page A5-49. It determines the P, U, Rn, W and 8_bit_word_offset bits of the instruction. The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

`STC` is in all versions.

`STC2` is in ARMv5 and above.

Exceptions

Undefined Instruction, Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    address = start_address
    Memory[address,4] = value from Coprocessor[cp_num]
    if Shared(address) then /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
    while (NotFinished(coprocessor[cp_num]))
        address = address + 4
        Memory[address,4] = value from Coprocessor[cp_num]
        if Shared(address) then    /* from ARMv6 */
            physical_address = TLB(address)
            ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See Summary of operation on page A2-49 */
    assert address == end_address
```

Usage

STC is useful for storing coprocessor data to memory. The L (long) option controls the N bit and could be used to distinguish between a single- and double-precision transfer for a floating-point store instruction.

Notes

Coprocessor fields Only instruction bits[31:23], bits[21:16} and bits[11:0] are defined by the ARM architecture. The remaining fields (bit[22] and bits[15:12]) are recommendations, for compatibility with ARM Development Systems.

In the case of the Unindexed addressing mode (P==0, U==1, W==0), instruction bits[7:0] are also not ARM architecture-defined, and can be used to specify additional coprocessor options.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0 the store coprocessor register instructions ignore the least significant two bits of `address`. For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Unimplemented coprocessor instructions

Hardware coprocessor support is optional, regardless of the architecture version. An implementation can choose to implement a subset of the coprocessor instructions, or no coprocessor instructions at all. Any coprocessor instructions that are not implemented instead cause an Undefined Instruction exception.

A4.1.97 STM (1)

31 28	27 26 25	24	23	22	21	20	19 16	15 0
cond	1 0 0	P	U	0	W	0	Rn	register_list

`STM` (1) (Store Multiple) stores a non-empty subset (or possibly all) of the general-purpose registers to sequential memory locations.

Syntax

```
STM{<cond>}<addressing_mode>  <Rn>{!}, <registers>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<addressing_mode>`	Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P, U, and W bits of the instruction.
`<Rn>`	Specifies the base register used by `<addressing_mode>`. If R15 is specified as `<Rn>`, the result is UNPREDICTABLE.
`!`	Sets the W bit, causing the instruction to write a modified value back to its base register Rn as specified in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. If ! is omitted, the W bit is 0 and the instruction does not change its base register in this way.
`<registers>`	Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction. The registers are stored in sequence, the lowest-numbered register to the lowest memory address (`start_address`), through to the highest-numbered register to the highest memory address (`end_address`). For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE. If R15 is specified in `<registers>`, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-9.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1 then
            Memory[address,4] = Ri
            address = address + 4
            if Shared(address) then   /* from ARMv6 */
                physical_address = TLB(address)
                ClearExclusiveByAddress(physical_address,processor_id,4)
                /* See Summary of operation on page A2-49 */
    assert end_address == address - 4
```

Usage

`STM` is useful as a block store instruction (combined with `LDM` it allows efficient block copy) and for stack operations. A single `STM` used in the sequence of a procedure can push the return address and general-purpose register values on to the stack, updating the stack pointer in the process.

Notes

Operand restrictions

If `<Rn>` is specified in `<registers>` and base register write-back is specified:

- If `<Rn>` is the lowest-numbered register specified in `<registers>`, the original value of `<Rn>` is stored.
- Otherwise, the stored value of `<Rn>` is UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the `STM[1]` instruction ignores the least significant two bits of `address`. For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault.

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Memory access restrictions* on page B2-13 for details.

A4.1.98 STM (2)

31 28	27	26	25	24	23	22	21	20	19 16	15 0
cond	1	0	0	P	U	1	0	0	Rn	register_list

`STM` (2) stores a subset (or possibly all) of the User mode general-purpose registers to sequential memory locations.

Syntax

```
STM{<cond>}<addressing_mode>  <Rn>, <registers>^
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<addressing_mode>`

Is described in *Addressing Mode 4 - Load and Store Multiple* on page A5-41. It determines the P and U bits of the instruction. Only the forms of this addressing mode with W == 0 are available for this form of the `STM` instruction.

`<Rn>` Specifies the base register used by `<addressing_mode>`. If R15 is specified as the base register `<Rn>`, the result is UNPREDICTABLE.

`<registers>` Is a list of registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored by the `STM` instruction.

The registers are stored in sequence, the lowest-numbered register to the lowest memory address (`start_address`), through to the highest-numbered register to the highest memory address (`end_address`).

For each of i=0 to 15, bit[i] in the register_list field of the instruction is 1 if Ri is in the list and 0 otherwise. If bits[15:0] are all zero, the result is UNPREDICTABLE.

If R15 is specified in `<registers>` the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-9.

`^` For an `STM` instruction, indicates that User mode registers are to be stored.

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    address = start_address
    for i = 0 to 15
        if register_list[i] == 1
            Memory[address,4] = Ri_usr
            address = address + 4
            if Shared(address) then    /* from ARMv6 */
                physical_address = TLB(address)
                ClearExclusiveByAddress(physical_address,processor_id,4)
                /* See Summary of operation on page A2-49 */
    assert end_address == address - 4
```

Usage

Use `STM` (2) to store the User mode registers when the processor is in a privileged mode (useful when performing process swaps, and in instruction emulators).

Notes

Write-back Setting bit 21, the W bit, has UNPREDICTABLE results.

User and System mode

This instruction is UNPREDICTABLE in User or System mode.

Base register mode For the purpose of address calculation, the base register is read from the current processor mode registers, not the User mode registers.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Non word-aligned addresses

For CP15_reg1_Ubit == 0, the `STM[2]` instruction ignores the least significant two bits of `address`. For CP15_reg1_Ubit == 1, all non-word aligned accesses cause an alignment fault

Alignment If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

Time order The time order of the accesses to individual words of memory generated by this instruction is only defined in some circumstances. See *Memory access restrictions* on page B2-13 for details.

Banked registers In ARM architecture versions earlier than ARMv6, this form of `STM` must not be followed by an instruction that accesses banked registers (a following `NOP` is a good way to ensure this).

A4.1.99 STR

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	0	W	0	Rn	Rd	addr_mode

STR (Store Register) stores a word from a register to memory.

Syntax

```
STR{<cond>}  <Rd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the source register for the operation. If R15 is specified for `<Rd>`, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-9.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    Memory[address,4] = Rd
    if Shared(address) then    /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
        /* See Summary of operation on page A2-49 */
```

Usage

Combined with a suitable addressing mode, `STR` stores 32-bit data from a general-purpose register into memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment Prior to ARMv6, `STR` ignores the least significant two bits of the address. This is different from the `LDR` behavior. Alignment checking (taking a data abort when address[1:0] != 0b00), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, a byte- invariant mixed-endian format is supported, along with an alignment checking option. The pseudo-code for the ARMv6 case assumes that unaligned mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment see *Endian support* on page A2-30and *Unaligned access support* on page A2-38.

A4.1.100 STRB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	P	U	1	W	0	Rn	Rd	addr_mode

`STRB` (Store Register Byte) stores a byte from the least significant byte of a register to memory.

Syntax

```
STR{<cond>}B  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, P, U, W, Rn and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
    if Shared(address) then      /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See Summary of operation on page A2-49 */
```

Usage

Combined with a suitable addressing mode, `STRB` writes the least significant byte of a general-purpose register to memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Notes

Operand restrictions

If `<addressing_mode>` specifies base register write-back, and the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.101 STRBT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	1	1	0	Rn	Rd	addr_mode

`STRBT` (Store Register Byte with Translation) stores a byte from the least significant byte of a register to memory. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor were in User mode.

Syntax

```
STR{<cond>}BT  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    Memory[address,1] = Rd[7:0]
    if Shared(address) then      /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See Summary of operation on page A2-49 */
```

Usage

STRBT can be used by a (privileged) exception handler that is emulating a memory access instruction which would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

A4.1.102 STRD

31 28	27 26 25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0 0 0	P	U	I	W	0	Rn	Rd	addr_mode	1	1	1	1	addr_mode

`STRD` (Store Registers Doubleword) stores a pair of ARM registers to two consecutive words of memory. The pair of registers is restricted to being an even-numbered register and the odd-numbered register that immediately follows it (for example, R10 and R11).

A greater variety of addressing modes is available than for a two-register `STM`.

Syntax

```
STR{<cond>}D  <Rd>, <addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the even-numbered register that is stored to the memory word addressed by `<addressing_mode>`. The immediately following odd-numbered register is stored to the next memory word. If `<Rd>` is R14, which would specify R15 as the second source register, the instruction is UNPREDICTABLE.

If `<Rd>` specifies an odd-numbered register, the instruction is UNDEFINED.

`<addressing_mode>`

Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn, and addr_mode bits of the instruction.

The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

The address generated by `<addressing_mode>` is the address of the lower of the two words stored by the `STRD` instruction. The address of the higher word is generated by adding 4 to this address.

Architecture version

ARMv5TE and above, excluding ARMv5TExP.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    if (Rd is even-numbered) and (Rd is not R14) and
            (address[1:0] == 0b00) and
            ((CP15_reg1_Ubit == 1) or (address[2] == 0)) then
        Memory[address,4] = Rd
        Memory[address+4,4] = R(d+1)
    else
        UNPREDICTABLE
    if Shared(address) then    /* from ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
    if Shared(address+4)
        physical_address = TLB(address+4)
        ClearExclusiveByAddress(physical_address,processor_id,4)
```

Notes

Operand restrictions

If `<addressing_mode>` performs base register write-back and the base register `<Rn>` is one of the two source registers of the instruction, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment Prior to ARMv6, if the memory address is not 64-bit aligned, the instruction is UNPREDICTABLE. Alignment checking (taking a data abort), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, a byte-invariant mixed-endian format is supported, along with alignment checking options; modulo4 and modulo8. The pseudo-code for the ARMv6 case assumes that unaligned mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit.

For more details on endianness and alignment, see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

Time order The time order of the accesses to the two memory words is not architecturally defined. In particular, an implementation is allowed to perform the two 32-bit memory accesses in either order, or to combine them into a single 64-bit memory access.

A4.1.103 STREX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	1	0	0	0	Rn	Rd	SBO	1	0	0	1	Rm

`STREX` (Store Register Exclusive) performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

Syntax

```
STREX{<cond>} <Rd>, <Rm>, [<Rn>]
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies 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.
`<Rm>`	Specifies the register containing the word to be stored to memory.
`<Rn>`	Specifies the register containing the address.

Architecture version

ARMv6 and above.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
if ConditionPassed(cond) then
    processor_id = ExecutingProcessor()
    physical_address = TLB(Rn)
    if IsExclusiveLocal(physical_address, processor_id, 4) then
        if Shared(Rn) == 1 then
            if IsExclusiveGlobal(physical_address, processor_id, 4) then
                Memory[Rn,4] = Rm
                Rd = 0
                ClearExclusiveByAddress(physical_address,processor_id,4)
            else
                Rd = 1
        else
            Memory[Rn,4] = Rm
            Rd = 0
    else
        Rd = 1
    ClearExclusiveLocal(processor_id)
    /* See Summary of operation on page A2-49 */
    /* The notes take precedence over any implied atomicity or
       order of events indicated in the pseudo-code */
```

Usage

Use STREX in combination with LDREX to implement inter-process communication in multiprocessor and shared memory systems. See *LDREX* on page A4-52 for further information.

Notes

Use of R15 Specifying R15 for register <Rd>, <Rn>, or <Rm> has UNPREDICTABLE results.

Operand restrictions

<Rd> must be distinct from both <Rm> and <Rn>, otherwise the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21. If a Data Abort occurs during execution of a STREX instruction:

- memory is not updated
- <Rd> is not updated.

Alignment If CP15 register 1(A,U) != (0,0) and Rd<1:0> != 0b00, an alignment exception will be taken.

There is no support for unaligned Load Exclusive. If Rd<1:0> != 0b00 and (A,U) = (0,0), the result is UNPREDICTABLE

A4.1.104 STRH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	P	U	I	W	0	Rn	Rd	addr_mode	1	0	1	1	addr_mode

`STRH` (Store Register Halfword) stores a halfword from the least significant halfword of a register to memory. If the address is not halfword-aligned, the result is UNPREDICTABLE.

Syntax

```
STR{<cond>}H  <Rd>, <addressing_mode>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the source register for the operation. If R15 is specified for `<Rd>`, the result is UNPREDICTABLE.
`<addressing_mode>`	Is described in *Addressing Mode 3 - Miscellaneous Loads and Stores* on page A5-33. It determines the P, U, I, W, Rn and addr_mode bits of the instruction. The syntax of all forms of `<addressing_mode>` includes a *base register* `<Rn>`. Some forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        if address[0] == 0b0 then
            Memory[address,2] = Rd[15:0]
        else
            Memory[address,2] = UNPREDICTABLE
    else    /* CP15_reg1_Ubit ==1  */
        Memory[address,2] = Rd[15:0]
    if Shared(address) then    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,2)
        /* See Summary of operation on page A2-49 */
```

Usage

Combined with a suitable addressing mode, STRH allows 16-bit data from a general-purpose register to be stored to memory. Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Notes

Operand restrictions	If <addressing_mode> specifies base register write-back, and the same register is specified for <Rd> and <Rn>, the results are UNPREDICTABLE.
Data Abort	For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.
Alignment	Prior to ARMv6, if the memory address is not halfword aligned, the instruction is UNPREDICTABLE. Alignment checking (taking a data abort when address[0] != 0), and support for a big-endian (BE-32) data format are implementation options. From ARMv6, a byte-invariant mixed-endian format is supported, along with an alignment checking option. The pseudo-code for the ARMv6 case assumes that mixed-endian support is configured, with the endianness of the transfer defined by the CPSR E-bit. For more details on endianness and alignment, see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

A4.1.105 STRT

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	1	I	0	U	0	1	0	Rn	Rd	addr_mode

`STRT` (Store Register with Translation) stores a word from a register to memory. If the instruction is executed when the processor is in a privileged mode, the memory system is signaled to treat the access as if the processor was in User mode.

Syntax

```
STR{<cond>}T  <Rd>, <post_indexed_addressing_mode>
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the source register for the operation. If R15 is specified for `<Rd>`, the value stored is IMPLEMENTATION DEFINED. For more details, see *Reading the program counter* on page A2-9.

`<post_indexed_addressing_mode>`

Is described in *Addressing Mode 2 - Load and Store Word or Unsigned Byte* on page A5-18. It determines the I, U, Rn and addr_mode bits of the instruction. Only post-indexed forms of Addressing Mode 2 are available for this instruction. These forms have P == 0 and W == 0, where P and W are bit[24] and bit[21] respectively. This instruction uses P == 0 and W == 1 instead, but the addressing mode is the same in all other respects.

The syntax of all forms of `<post_indexed_addressing_mode>` includes a *base register* `<Rn>`. All forms also specify that the instruction modifies the base register value (this is known as *base register write-back*).

Architecture version

All.

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    Memory[address,4] = Rd
    if Shared(address) then    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,4)
            /* See Summary of operation on page A2-49 */
```

Usage

`STRT` can be used by a (privileged) exception handler that is emulating a memory access instruction that would normally execute in User mode. The access is restricted as if it had User mode privilege.

Notes

User mode If this instruction is executed in User mode, an ordinary User mode access is performed.

Operand restrictions

If the same register is specified for `<Rd>` and `<Rn>`, the results are UNPREDICTABLE.

Data Abort For details of the effects of the instruction if a Data Abort occurs, see *Effects of data-aborted instructions* on page A2-21.

Alignment As for `STR`, see *STR* on page A4-193.

If an implementation includes a System Control coprocessor (see Chapter B3 *The System Control Coprocessor*), and alignment checking is enabled, an address with bits[1:0] != 0b00 causes an alignment exception.

A4.1.106 SUB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	0	0	1	0	S	Rn	Rd	shifter_operand

`SUB` (Subtract) subtracts one value from a second value.

The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

`SUB` can optionally update the condition code flags, based on the result.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`S` Sets the S bit (bit[20]) in the instruction to 1 and specifies that the instruction updates the CPSR. If `S` is omitted, the S bit is set to 0 and the CPSR is not changed by the instruction. Two types of CPSR update can occur when `S` is specified:

- If `<Rd>` is not R15, the `N` and `Z` flags are set according to the result of the subtraction, and the `C` and `V` flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively. The rest of the CPSR is unchanged.
- If `<Rd>` is R15, the SPSR of the current mode is copied to the CPSR. This form of the instruction is UNPREDICTABLE if executed in User mode or System mode, because these modes do not have an SPSR.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `SUB`. Instead, see *Extending the instruction set* on page A3-32 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd = Rn - shifter_operand
    if S == 1 and Rd == R15 then
        if CurrentModeHasSPSR() then
            CPSR = SPSR
        else UNPREDICTABLE
    else if S == 1 then
        N Flag = Rd[31]
        Z Flag = if Rd == 0 then 1 else 0
        C Flag = NOT BorrowFrom(Rn - shifter_operand)
        V Flag = OverflowFrom(Rn - shifter_operand)
```

Usage

Use `SUB` to subtract one value from another. To decrement a register value (in `Ri`) use:

```
    SUB     Ri, Ri, #1
```

`SUBS` is useful as a loop counter decrement, as the loop branch can test the flags for the appropriate termination condition, without the need for a separate compare instruction:

```
    SUBS    Ri, Ri, #1
```

This both decrements the loop counter in Ri and checks whether it has reached zero.

You can use `SUB`, with the PC as its destination register and the S bit set, to return from interrupts and various other types of exception. See *Exceptions* on page A2-16 for more details.

Notes

C flag If `S` is specified, the C flag is set to:

`1`	if no borrow occurs
`0`	if a borrow does occur.

In other words, the C flag is used as a NOT(borrow) flag. This inversion of the borrow condition is used by subsequent instructions: `SBC` and `RSC` use the C flag as a NOT(borrow) operand, performing a normal subtraction if C == 1 and subtracting one more than usual if C == 0.

The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

A4.1.107 SWI

31 28	27	26	25	24	23 0
cond	1	1	1	1	immed_24

`SWI` (Software Interrupt) causes a SWI exception (see *Exceptions* on page A2-16).

Syntax

```
SWI{<cond>}  <immed_24>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<immed_24>`	Is a 24-bit immediate value that is put into bits[23:0] of the instruction. This value is ignored by the ARM processor, but can be used by an operating system SWI exception handler to determine what operating system service is being requested (see *Usage* on page A4-211 below for more details).

Architecture version

All.

Exceptions

Software interrupt.

Operation

```
if ConditionPassed(cond) then
    R14_svc   = address of next instruction after the SWI instruction
    SPSR_svc  = CPSR
    CPSR[4:0] = 0b10011             /* Enter Supervisor mode */
    CPSR[5]   = 0                   /* Execute in ARM state */
    /* CPSR[6] is unchanged */
    CPSR[7]   = 1                   /* Disable normal interrupts */
    /* CPSR[8] is unchanged */
    CPSR[9] = CP15_reg1_EEbit
    if high vectors configured then
        PC    = 0xFFFF0008
    else
        PC    = 0x00000008
```

Usage

`SWI` is used as an operating system service call. The method used to select which operating system service is required is specified by the operating system, and the SWI exception handler for the operating system determines and provides the requested service. Two typical methods are:

- The 24-bit immediate in the instruction specifies which service is required, and any parameters needed by the selected service are passed in general-purpose registers.
- The 24-bit immediate in the instruction is ignored, general-purpose register R0 is used to select which service is wanted, and any parameters needed by the selected service are passed in other general-purpose registers.

A4.1.108 SWP

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	0	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

SWP (Swap) swaps a word between registers and memory. SWP loads a word from the memory address given by the value of register <Rn>. The value of register <Rm> is then stored to the memory address given by the value of <Rn>, and the original loaded value is written to register <Rd>. If the same register is specified for <Rd> and <Rm>, this instruction swaps the value of the register and the value at the memory address.

Syntax

```
SWP{<cond>}  <Rd>, <Rm>, [<Rn>]
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register for the instruction.
<Rm>	Contains the value that is stored to memory.
<Rn>	Contains the memory address to load from.

Architecture version

All (deprecated in ARMv6).

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    if (CP15_reg1_Ubit == 0) then
        temp = Memory[address,4] Rotate_Right (8 * address[1:0])
        Memory[address,4] = Rm
        Rd = temp
    else /* CP15_reg1_Ubit ==1 */
        temp = Memory[address,4]
        Memory[address,4] = Rm
        Rd = temp
    if Shared(address) then   /* ARMv6 */
```

```
physical_address = TLB(address)
ClearExclusiveByAddress(physical_address,processor_id,4)
/* See Summary of operation on page A2-49 */
```

Usage

You can use SWP to implement semaphores. This instruction is deprecated in ARMv6. Software should migrate to using the Load/Store exclusive instructions described in *Synchronization primitives* on page A2-44.

Notes

Use of R15 If R15 is specified for <Rd>, <Rn>, or <Rm>, the result is UNPREDICTABLE.

Operand restrictions

If the same register is specified as <Rn> and <Rm>, or <Rn> and <Rd>, the result is UNPREDICTABLE.

Data Abort If a precise Data Abort is signaled on either the load access or the store access, the loaded value is not written to <Rd>. If a precise Data Abort is signaled on the load access, the store access does not occur.

Alignment Prior to ARMv6, the alignment rules are the same as for an LDR on the read (see *LDR* on page A4-43) and an STR on the write (see *STR* on page A4-193). Alignment checking (taking a data abort when address[1:0] != 0b00), and support for a big-endian (BE-32) data format are implementation options.

From ARMv6, if CP15 register 1(A,U) != (0,0) and Rn[1:0] != 0b00, an alignment exception is taken. If CP15 register 1(A,U) == (0,0), the behavior is the same as the behavior before ARMv6.

For more details on endianness and alignment see *Endian support* on page A2-30 and *Unaligned access support* on page A2-38.

Memory model considerations

Swap is an atomic operation for all accesses, cached and non-cached.

The swap operation does not include any memory barrier guarantees. For example, it does not guarantee flushing of write buffers, which is an important consideration on multiprocessor systems.

A4.1.109 SWPB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	1	0	1	0	0	Rn	Rd	SBZ	1	0	0	1	Rm

`SWPB` (Swap Byte) swaps a byte between registers and memory. `SWPB` loads a byte from the memory address given by the value of register `<Rn>`. The value of the least significant byte of register `<Rm>` is stored to the memory address given by `<Rn>`, the original loaded value is zero-extended to a 32-bit word, and the word is written to register `<Rd>`. If the same register is specified for `<Rd>` and `<Rm>`, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address.

Syntax

```
SWP{<cond>}B  <Rd>, <Rm>, [<Rn>]
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register for the instruction.
`<Rm>`	Contains the value that is stored to memory.
`<Rn>`	Contains the memory address to load from.

Architecture version

All (deprecated in ARMv6).

Exceptions

Data Abort.

Operation

```
MemoryAccess(B-bit, E-bit)
processor_id = ExecutingProcessor()
if ConditionPassed(cond) then
    temp = Memory[address,1]
    Memory[address,1] = Rm[7:0]
    Rd = temp
    if Shared(address) then    /* ARMv6 */
        physical_address = TLB(address)
        ClearExclusiveByAddress(physical_address,processor_id,1)
        /* See Summary of operation on page A2-49 */
```

Usage

You can use `SWPB` to implement semaphores. This instruction is deprecated in ARMv6. Software should migrate to using the Load /Store exclusive instructions described in *Synchronization primitives* on page A2-44.

Notes

Use of R15 If R15 is specified for `<Rd>`, `<Rn>`, or `<Rm>`, the result is UNPREDICTABLE.

Operand restrictions If the same register is specified as `<Rn>` and `<Rm>`, or `<Rn>` and `<Rd>`, the result is UNPREDICTABLE.

Data Abort If a precise Data Abort is signaled on either the load access or the store access, the loaded value is not written to `<Rd>`. If a precise Data Abort is signaled on the load access, the store access does not occur.

Memory model considerations Swap is an atomic operation for all accesses, cached and non-cached.

The swap operation does not include any memory barrier guarantees. For example, it does not guarantee flushing of write buffers, which is an important consideration on multiprocessor systems.

A4.1.110 SXTAB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	1	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`SXTAB` extracts an 8-bit value from a register, sign extends it to 32 bits, and adds the result to the value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
SXTAB{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

`<rotation>` This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd = Rn + SignExtend(operand2[7:0])
```

Usage

You can use `SXTAB` to eliminate a separate sign-extension instruction in many instruction sequences that act on **`signed char`** values in C/C++.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is an `SXTB` instruction instead, see *SXTB* on page A4-222.

A4.1.111 SXTAB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	0	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`SXTAB16` extracts two 8-bit values from a register, sign extends them to 16 bits each, and adds the results to two 16-bit values from another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

```
SXTAB16{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = Rn[15:0]  + SignExtend(operand2[7:0])
    Rd[31:16] = Rn[31:16] + SignExtend(operand2[23:16])
```

Usage

Use SXTAB16 when you need to keep intermediate values to higher precision while working on arrays of signed byte values. See *UXTAB16* on page A4-276 for an example of a similar usage.

Notes

Use of R15 Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register <Rn>.

Encoding If the <Rn> field of the instruction contains 0b1111, the instruction is an SXTB16 instruction instead, see *SXTB16* on page A4-224.

A4.1.112 SXTAH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	1	1	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`SXTAH` extracts a 16-bit value from a register, sign extends it to 32 bits, and adds the result to a value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
SXTAH{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd       = Rn + SignExtend(operand2[15:0])
```

Usage

You can use SXTAH to eliminate a separate sign-extension instruction in many instruction sequences that act on **signed short** values in C/C++.

Notes

Use of R15 Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register <Rn>.

Encoding If the <Rn> field of the instruction contains 0b1111, the instruction is an SXTH instruction instead, see *SXTH* on page A4-226.

A4.1.113 SXTB

31 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 8	7 6 5 4	3 0
cond	0 1 1 0 1 0 1 0	1 1 1 1	Rd	rotate	SBZ	0 1 1 1	Rm

`SXTB` extracts an 8-bit value from a register and sign extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
SXTB{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[7:0])
```

Usage

Use SXTB to sign-extend a byte to a word, for example in instruction sequences acting on **signed char** values in C/C++.

Notes

Use of R15 Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results

A4.1.114 SXTB16

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	0	0	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

`SXTB16` extracts two 8-bit values from a register and sign extends them to 16 bits each. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

```
SXTB16{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = Rm Rotate_Right(8 * rotate)
    Rd[15:0]  = SignExtend(operand2[7:0])
    Rd[31:16] = SignExtend(operand2[23:16])
```

Usage

Use `SXTB16` when you need to keep intermediate values to higher precision while working on arrays of signed byte values. See *UXTAB16* on page A4-276 for an example of a similar usage.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.115 SXTH

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	0	1	1	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

SXTH extracts a 16-bit value from a register and sign extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
SXTH{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = Rm Rotate_Right(8 * rotate)
    Rd[31:0] = SignExtend(operand2[15:0])
```

Usage

Use `SXTH` to sign-extend a halfword to a word, for example in instruction sequences acting on **`signed short`** values in C/C++.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.116 TEQ

31 28	27 26	25	24 23 22 21	20	19 16	15 12	11 0
cond	0 0	I	1 0 0 1	1	Rn	SBZ	shifter_operand

`TEQ` (Test Equivalence) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically exclusive-ORing the two values, so that subsequent instructions can be conditionally executed.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rn>`	Specifies the register that contains the first operand.
`<shifter_operand>`	Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option sets the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) in the instruction. If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `TEQ`. Instead, see *Multiply instruction extension space* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn EOR shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

Use `TEQ` to test if two values are equal, without affecting the V flag (as `CMP` does). The C flag is also unaffected in many cases. `TEQ` is also useful for testing whether two values have the same sign. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

A4.1.117 TST

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 0
cond	0	0	I	1	0	0	0	1	Rn	SBZ	shifter_operand

`TST` (Test) compares a register value with another arithmetic value. The condition flags are updated, based on the result of logically ANDing the two values, so that subsequent instructions can be conditionally executed.

Syntax

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

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rn>` Specifies the register that contains the first operand.

`<shifter_operand>`

Specifies the second operand. The options for this operand are described in *Addressing Mode 1 - Data-processing operands* on page A5-2, including how each option causes the I bit (bit[25]) and the shifter_operand bits (bits[11:0]) to be set in the instruction.

If the I bit is 0 and both bit[7] and bit[4] of shifter_operand are 1, the instruction is not `TST`. Instead, see *Multiply instruction extension space* on page A3-35 to determine which instruction it is.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    alu_out = Rn AND shifter_operand
    N Flag = alu_out[31]
    Z Flag = if alu_out == 0 then 1 else 0
    C Flag = shifter_carry_out
    V Flag = unaffected
```

Usage

Use TST to determine whether a particular subset of register bits includes at least one set bit. A very common use for TST is to test whether a single bit is set or clear.

A4.1.118 UADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	0	0	1	Rm

`UADD16` (Unsigned Add) performs two 16-bit unsigned integer additions. It sets the GE bits in the CPSR as carry flags for the additions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] + Rm[15:0]
    GE[1:0]   = if CarryFrom16(Rn[15:0] + Rm[15:0]) == 1 then 0b11 else 0
    Rd[31:16] = Rn[31:16] + Rm[31:16]
    GE[3:2]   = if CarryFrom16(Rn[31:16] + Rm[31:16]) == 1 then 0b11 else 0
```

Usage

`UADD16` produces the same result value as `SADD16`. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.119 UADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	1	0	0	1	Rm

`UADD8` performs four 8-bit unsigned integer additions. It sets the GE bits in the CPSR as carry flags for the additions.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = Rn[7:0] + Rm[7:0]
    GE[0]     = CarryFrom8(Rn[7:0] + Rm[7:0])
    Rd[15:8]  = Rn[15:8] + Rm[15:8]
    GE[1]     = CarryFrom8(Rn[15:8] + Rm[15:8])
    Rd[23:16] = Rn[23:16] + Rm[23:16]
    GE[2]     = CarryFrom8(Rn[23:16] + Rm[23:16])
    Rd[31:24] = Rn[31:24] + Rm[31:24]
    GE[3]     = CarryFrom8(Rn[31:24] + Rm[31:24])
```

Usage

`UADD8` produces the same result value as `SADD8`. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.120 UADDSUBX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	0	1	1	Rm

`UADDSUBX` (Unsigned Add and Subtract with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction. It exchanges the two halfwords of the second operand before it performs the arithmetic. It sets the GE bits in the CPSR according to the results of the addition and subtraction.

Syntax

```
UADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]    /* unsigned addition */
    Rd[31:16] = sum[15:0]
    GE[3:2]   = if CarryFrom16(Rn[31:16] + Rm[15:0]) then 0b11 else 0
    diff      = Rn[15:0] - Rm[31:16]    /* unsigned subtraction */
    Rd[15:0]  = diff[15:0]
    GE[1:0]   = if BorrowFrom(Rn[15:0] - Rm[31:16]) then 0b11 else 0
```

Usage

UADDSUBX produces the same result value as SADDSUBX. However, the GE flag values are based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15	Specifying R15 for register <Rd>, <Rm>, or <Rn> has UNPREDICTABLE results.

A4.1.121 UHADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	0	0	1	Rm

`UHADD16` (Unsigned Halving Add) performs two 16-bit unsigned integer additions, and halves the results. It has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[15:0] + Rm[15:0]   /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
    sum       = Rn[31:16] + Rm[31:16] /* Unsigned addition */
    Rd[31:16] = sum[16:1]
```

Usage

Use `UHADD16` for similar purposes to `UADD16` (see *UADD16* on page A4-232). `UHADD16` averages the operands.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.122 UHADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	1	0	0	1	Rm

`UHADD16` performs four 8-bit unsigned integer additions, and halves the results. It has no effect on the GE flags.

Syntax

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

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[7:0] + Rm[7:0]       /* Unsigned addition */
    Rd[7:0]   = sum[8:1]
    sum       = Rn[15:8] + Rm[15:8]     /* Unsigned addition */
    Rd[15:8]  = sum[8:1]
    sum       = Rn[23:16] + Rm[23:16]   /* Unsigned addition */
    Rd[23:16] = sum[8:1]
    sum       = Rn[31:24] + Rm[31:24]   /* Unsigned addition */
    Rd[31:24] = sum[8:1]
```

Usage

Use `UHADD8` for similar purposes to `UADD8` (see *UADD8* on page A4-233). `UHADD8` averages the operands.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.123 UHADDSUBX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	0	0	1	1	Rm

UHADDSUBX (Unsigned Halving Add and Subtract with Exchange) performs one 16-bit unsigned integer addition and one 16-bit unsigned integer subtraction, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

It has no effect on the GE flags.

Syntax

```
UHADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

<cond>	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If <cond> is omitted, the AL (always) condition is used.
<Rd>	Specifies the destination register.
<Rn>	Specifies the register that contains the first operand.
<Rm>	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    sum       = Rn[31:16] + Rm[15:0]  /* Unsigned addition */
    Rd[31:16] = sum[16:1]
    diff      = Rn[15:0] - Rm[31:16]  /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
```

Usage

Use UHADDSUBX for similar purposes to UADDSUBX (see *UADDSUBX* on page A4-235). UHADDSUBX halves the results.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.124 UHSUB16

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 1 1	Rn	Rd	SBO	0 1 1 1	Rm

`UHSUB16` (Unsigned Halving Subtract) performs two 16-bit unsigned integer subtractions, and halves the results. It has no effect on the GE flags.

Syntax

```
UHSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[15:0] - Rm[15:0]    /* Unsigned subtraction */
    Rd[15:0]  = diff[16:1]
    diff      = Rn[31:16] - Rm[31:16]  /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
```

Usage

Use `UHSUB16` for similar purposes to `USUB16` (see *USUB16* on page A4-269). `UHSUB16` gives half the difference instead of the full difference.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.125 UHSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	1	Rn	Rd	SBO	1	1	1	1	Rm

`UHSUB8` performs four 8-bit unsigned integer subtractions, and halves the results. It has no effect on the GE flags.

Syntax

```
UHSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[7:0] - Rm[7:0]      /* Unsigned subtraction */
    Rd[7:0]   = diff[8:1]
    diff      = Rn[15:8] - Rm[15:8]    /* Unsigned subtraction */
    Rd[15:8]  = diff[8:1]
    diff      = Rn[23:16] - Rm[23:16]  /* Unsigned subtraction */
    Rd[23:16] = diff[8:1]
    diff      = Rn[31:24] - Rm[31:24]  /* Unsigned subtraction */
    Rd[31:24] = diff[8:1]
```

Usage

Use `UHSUB8` for similar purposes to `USUB8` (see *USUB8* on page A4-270). `UHSUB8` gives half the difference instead of the full difference.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.126 UHSUBADDX

31 28	27 26 25 24 23 22 21 20	19 16	15 12	11 8	7 6 5 4	3 0
cond	0 1 1 0 0 1 1 1	Rn	Rd	SBO	0 1 0 1	Rm

`UHSUBADDX` (Unsigned Halving Subtract and Add with Exchange) performs one 16-bit unsigned integer subtraction and one 16-bit unsigned integer addition, and halves the results. It exchanges the two halfwords of the second operand before it performs the arithmetic.

It has no effect on the GE flags.

Syntax

```
UHSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]     /* Unsigned subtraction */
    Rd[31:16] = diff[16:1]
    sum       = Rn[15:0]  + Rm[31:16]    /* Unsigned addition */
    Rd[15:0]  = sum[16:1]
```

Usage

Use `UHSUBADDX` for similar purposes to `USUBADDX` (see *USUBADDX* on page A4-272). `UHSUBADDX` gives half the difference and the average instead of the full difference and sum.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.127 UMAAL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	0	1	0	0	RdHi	RdLo	Rs	1	0	0	1	Rm

`UMAAL` (Unsigned Multiply Accumulate Accumulate Long) multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit product. Both the unsigned 32-bit value held in `<RdHi>` and the unsigned 32-bit value held in `<RdLo>` are added to this product, and the sum is written back to `<RdHi>` and `<RdLo>` as a 64-bit value. The flags are not updated.

Syntax

```
UMAAL{<cond>}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<RdLo>`	Supplies one of the 32-bit values to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the other 32-bit value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the unsigned value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the unsigned value to be multiplied with the value of `<Rm>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    result = Rm * Rs + RdLo + RdHi  /* Unsigned multiplication and additions */
    RdLo = result[31:0]
    RdHi = result[63:32]
```

Usage

Adding two 32-bit values to a 32-bit unsigned multiply is a useful function in cryptographic applications.

Notes

Use of R15	Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Operand restriction	If `<RdLo>` and `<RdHi>` are the same register, the results are UNPREDICTABLE.
Early termination	If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

A4.1.128 UMLAL

31 … 28	27	26	25	24	23	22	21	20	19 … 16	15 … 12	11 … 8	7	6	5	4	3 … 0
cond	0	0	0	0	1	0	1	S	RdHi	RdLo	Rs	1	0	0	1	Rm

`UMLAL` (Unsigned Multiply Accumulate Long) multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit product. This product is added to the 64-bit value held in `<RdHi>` and `<RdLo>`, and the sum is written back to `<RdHi>` and `<RdLo>`. The condition code flags are optionally updated, based on the result.

Syntax

```
UMLAL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiply-accumulate. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Supplies the lower 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the lower 32 bits of the result.
`<RdHi>`	Supplies the upper 32 bits of the value to be added to the product of `<Rm>` and `<Rs>`, and is the destination register for the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdLo = (Rm * Rs)[31:0] + RdLo    /* Unsigned multiplication */
    RdHi = (Rm * Rs)[63:32] + RdHi + CarryFrom((Rm * Rs)[31:0] + RdLo)
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`UMLAL` multiplies unsigned variables to produce a 64-bit result, which is added to the 64-bit value in the two destination general-purpose registers. The result is written back to the two destination general-purpose registers.

Notes

Use of R15 Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Operand restriction `<RdHi>` and `<RdLo>` must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either `<RdHi>` and `<Rm>`, or `<RdLo>` and `<Rm>`, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags `UMLALS` is defined to leave the C and V flags unchanged in ARMv5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a `UMLALS` instruction.

A4.1.129 UMULL

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	0	0	0	1	0	0	S	RdHi	RdLo	Rs	1	0	0	1	Rm

`UMULL` (Unsigned Multiply Long) multiplies the unsigned value of register `<Rm>` with the unsigned value of register `<Rs>` to produce a 64-bit result. The upper 32 bits of the result are stored in `<RdHi>`. The lower 32 bits are stored in `<RdLo>`. The condition code flags are optionally updated, based on the 64-bit result.

Syntax

```
UMULL{<cond>}{S}  <RdLo>, <RdHi>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`S`	Causes the S bit (bit[20]) in the instruction to be set to 1 and specifies that the instruction updates the CPSR by setting the N and Z flags according to the result of the multiplication. If `S` is omitted, the S bit of the instruction is set to 0 and the entire CPSR is unaffected by the instruction.
`<RdLo>`	Stores the lower 32 bits of the result.
`<RdHi>`	Stores the upper 32 bits of the result.
`<Rm>`	Holds the signed value to be multiplied with the value of `<Rs>`.
`<Rs>`	Holds the signed value to be multiplied with the value of `<Rm>`.

Architecture version

All.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    RdHi = (Rm * Rs)[63:32]    /* Unsigned multiplication */
    RdLo = (Rm * Rs)[31:0]
    if S == 1 then
        N Flag = RdHi[31]
        Z Flag = if (RdHi == 0) and (RdLo == 0) then 1 else 0
        C Flag = unaffected    /* See "C and V flags" note */
        V Flag = unaffected    /* See "C and V flags" note */
```

Usage

`UMULL` multiplies unsigned variables to produce a 64-bit result in two general-purpose registers.

Notes

Use of R15 Specifying R15 for register `<RdHi>`, `<RdLo>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

Operand restriction `<RdHi>` and `<RdLo>` must be distinct registers, or the results are UNPREDICTABLE.

Specifying the same register for either `<RdHi>` and `<Rm>`, or `<RdLo>` and `<Rm>`, was previously described as producing UNPREDICTABLE results. There is no restriction in ARMv6, and it is believed all relevant ARMv4 and ARMv5 implementations do not require this restriction either, because high performance multipliers read all their operands prior to writing back any results.

Early termination If the multiplier implementation supports early termination, it must be implemented on the value of the `<Rs>` operand. The type of early termination used (signed or unsigned) is IMPLEMENTATION DEFINED.

C and V flags `UMULLS` is defined to leave the C and V flags unchanged in ARMv5 and above. In earlier versions of the architecture, the values of the C and V flags were UNPREDICTABLE after a `UMULLS` instruction.

A4.1.130 UQADD16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	0	0	1	Rm

`UQADD16` (Unsigned Saturating Add) performs two 16-bit integer additions. It saturates the results to the 16-bit unsigned integer range $0 \le x \le 2^{16} - 1$. It has no effect on the GE flags.

Syntax

```
UQADD16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[31:16], 16)
```

Usage

Use `UQADD16` in similar ways to `UADD16`, but for unsigned saturated arithmetic. `UQADD16` does not set the GE bits for use with `SEL`. See *UADD16* on page A4-232 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.131 UQADD8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	1	0	0	1	Rm

`UQADD8` performs four 8-bit integer additions. It saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$. It has no effect on the GE flags.

Syntax

```
UQADD8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   + Rm[7:0],   8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  + Rm[15:8],  8)
    Rd[23:16] = UnsignedSat(Rn[23:16] + Rm[23:16], 8)
    Rd[31:24] = UnsignedSat(Rn[31:24] + Rm[31:24], 8)
```

Usage

Use `UQADD8` in similar ways to `UADD8`, but for unsigned saturated arithmetic. `UQADD8` does not set the GE bits for use with `SEL`. See *UADD8* on page A4-233 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.132 UQADDSUBX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	0	1	1	Rm

`UQADDSUBX` (Unsigned Saturating Add and Subtract with Exchange) performs one 16-bit integer addition and one 16-bit subtraction. It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$. It exchanges the two halfwords of the second operand before it performs the arithmetic. It has no effect on the GE flags.

Syntax

```
UQADDSUBX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[31:16], 16)
    Rd[31:16] = UnsignedSat(Rn[31:16] + Rm[15:0],  16)
```

Usage

Use `UQADDSUBX` in similar ways to `UADDSUBX`, but for unsigned saturated arithmetic. `UQADDSUBX` does not set the GE bits for use with `SEL`. See *UADDSUBX* on page A4-235 for more details.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.133 UQSUB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	1	1	1	Rm

`UQSUB16` (Unsigned Saturating Subtract) performs two 16-bit subtractions. It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$. It has no effect on the GE flags.

Syntax

```
UQSUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rn[15:0]  - Rm[15:0],  16)
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[31:16], 16)
```

Usage

Use `UQSUB16` in similar ways to `USUB16`, but for unsigned saturated arithmetic. `UQSUB16` does not set the GE bits for use with `SEL`. See *SSUB16* on page A4-180 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.134 UQSUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	1	1	1	1	Rm

`UQSUB8` performs four 8-bit subtractions. It saturates the results to the 8-bit unsigned integer range $0 \leq x \leq 2^8 - 1$. It has no effect on the GE flags.

Syntax

```
UQSUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = UnsignedSat(Rn[7:0]   - Rm[7:0],   8)
    Rd[15:8]  = UnsignedSat(Rn[15:8]  - Rm[15:8],  8)
    Rd[23:16] = UnsignedSat(Rn[23:16] - Rm[23:16], 8)
    Rd[31:24] = UnsignedSat(Rn[31:24] - Rm[31:24], 8)
```

Usage

Use `UQSUB8` in similar ways to `USUB8`, but for unsigned saturated arithmetic. `UQSUB8` does not set the GE bits for use with `SEL`. See *SSUB8* on page A4-182 for more details.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.135 UQSUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	1	0	Rn	Rd	SBO	0	1	0	1	Rm

UQSUBADDX (Unsigned Saturating Subtract and Add with Exchange) performs one 16-bit integer subtraction and one 16-bit integer addition. It saturates the results to the 16-bit unsigned integer range $0 \leq x \leq 2^{16} - 1$. It exchanges the two halfwords of the second operand before it performs the arithmetic. It has no effect on the GE flags.

Syntax

```
UQSUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:16] = UnsignedSat(Rn[31:16] - Rm[15:0],  16)
    Rd[15:0]  = UnsignedSat(Rn[15:0]  + Rm[31:16], 16)
```

Usage

You can use UQSUBADDX in similar ways to USUBADDX, but for unsigned saturated arithmetic. UQSUBADDX does not set the GE bits for use with SEL. See *UADDSUBX* on page A4-235 for more details.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.136 USAD8

31 28	27	26	25	24	23	22	21	20	19 16	15	14	13	12	11 8	7	6	5	4	3 0
cond	0	1	1	1	1	0	0	0	Rd	1	1	1	1	Rs	0	0	0	1	Rm

`USAD8` (Unsigned Sum of Absolute Differences) performs four unsigned 8-bit subtractions, and adds the absolute values of the differences together.

Syntax

```
USAD8{<cond>}  <Rd>, <Rm>, <Rs>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first operand.
`<Rs>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then        /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then      /* Unsigned comparison */
        diff2 = Rs[15:8] - Rm[15:8]
    else
        diff2 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then    /* Unsigned comparison */
        diff3 = Rs[23:16] - Rm[23:16]
    else
        diff3 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then    /* Unsigned comparison */
        diff4 = Rs[31:24] - Rm[31:24]
    else
        diff4 = Rm[31:24] - Rs[31:24]

    Rd = ZeroExtend(diff1) + ZeroExtend(diff2)
                       + ZeroExtend(diff3) + ZeroExtend(diff4]
```

Usage

You can use `USAD8` to process the first four bytes in a video motion estimation calculation.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.

A4.1.137 USADA8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	1	1	0	0	0	Rd	Rn	Rs	0	0	0	1	Rm

`USADA8` (Unsigned Sum of Absolute Differences and Accumulate) performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

Syntax

```
USADA8{<cond>}  <Rd>, <Rm>, <Rs>, <Rn>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the first main operand.
`<Rs>`	Specifies the register that contains the second main operand.
`<Rn>`	Specifies the register that contains the accumulate operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then

    if Rm[7:0] < Rs[7:0] then        /* Unsigned comparison */
        diff1 = Rs[7:0] - Rm[7:0]
    else
        diff1 = Rm[7:0] - Rs[7:0]

    if Rm[15:8] < Rs[15:8] then      /* Unsigned comparison */
        diff2 = Rs[15:8] - Rm[15:8]
    else
        diff2 = Rm[15:8] - Rs[15:8]

    if Rm[23:16] < Rs[23:16] then    /* Unsigned comparison */
        diff3 = Rs[23:16] - Rm[23:16]
    else
        diff3 = Rm[23:16] - Rs[23:16]

    if Rm[31:24] < Rs[31:24] then    /* Unsigned comparison */
        diff4 = Rs[31:24] - Rm[31:24]
    else
        diff4 = Rm[31:24] - Rs[31:24]

    Rd = Rn + ZeroExtend(diff1) + ZeroExtend(diff2)
                       + ZeroExtend(diff3) + ZeroExtend(diff4]
```

Usage

You can use `USADA8` in video motion estimation calculations.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rs>` has UNPREDICTABLE results.
Encoding	If the `<Rn>` field of the instruction contains `0b1111`, the instruction is a `USAD8` instruction instead, see *USAD8* on page A4-261.

A4.1.138 USAT

31 28	27	26	25	24	23	22	21	20 16	15 12	11 7	6	5	4	3 0
cond	0	1	1	0	1	1	1	sat_imm	Rd	shift_imm	sh	0	1	Rm

`USAT` (Unsigned Saturate) saturates a signed value to an unsigned range. You can choose the bit position at which saturation occurs.

You can apply a shift to the value before the saturation occurs.

The Q flag is set if the operation saturates.

Syntax

```
USAT{<cond>}  <Rd>, #<immed>, <Rm>{, <shift>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<immed>`	Specifies the bit position for saturation. This lies in the range 0 to 31. It is encoded in the `sat_imm` field of the instruction.
`<Rm>`	Specifies the register that contains the signed value to be saturated.
`<shift>`	Specifies the optional shift. If present, it must be one of: • `LSL #N`. `N` must be in the range 0 to 31. This is encoded as `sh` == 0 and `shift_imm` == `N`. • `ASR #N`. `N` must be in the range 1 to 32. This is encoded as `sh` == 1 and either `shift_imm` == 0 for `N` == 32, or `shift_imm == N` otherwise. If `<shift>` is omitted, `LSL #0` is used.

Return

The value returned in Rd is:

0	if X is < 0
X	if $0 <= X < 2^n$
$\mathbf{2^n - 1}$	if $X > 2^n - 1$

where n is `<immed>`, and X is the shifted value from Rm.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    if shift == 1 then
        if shift_imm == 0 then
            operand = (Rm Artihmetic_Shift_Right 32)[31:0]
        else
            operand = (Rm Artihmetic_Shift_Right shift_imm)[31:0]
    else
        operand = (Rm Logical_Shift_Left shift_imm)[31:0]
    Rd = UnsignedSat(operand, sat_imm)        /* operand treated as signed */
    if UnsignedDoesSat(operand, sat_imm) then
        Q Flag = 1
```

Usage

You can use USAT in various DSP algorithms, such as calculating a pixel color component, that require scaling and saturation of signed data to an unsigned destination.

Notes

Use of R15	Specifying R15 for register <Rd> or <Rm> has UNPREDICTABLE results.

A4.1.139 USAT16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	1	0	sat_imm	Rd	SBO	0	0	1	1	Rm

`USAT16` saturates two signed 16-bit values to an unsigned range. You can choose the bit position at which saturation occurs. The Q flag is set if either halfword operation saturates.

Syntax

```
USAT16{<cond>}  <Rd>, #<immed>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<immed>`	Specifies the bit position for saturation. This lies in the range 0 to 15. It is encoded in the `sat_imm` field of the instruction.
`<Rm>`	Specifies the register that contains the signed value to be saturated.

Return

The value returned in each half of Rd is:

0	if X is < 0
X	if $0 <= X < 2^n$
$2^n - 1$	if $X > 2^n - 1$

where n is `<immed>`, and X is the value from the corresponding half of Rm.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = UnsignedSat(Rm[15:0],  sat_imm) // Rm[15:0]  treated as signed
    Rd[31:16] = UnsignedSat(Rm[31:16], sat_imm) // Rm[31:16] treated as signed
    if UnsignedDoesSat(Rm[15:0], sat_imm)
                              OR UnsignedDoesSat(Rm[31:16], sat_imm) then
        Q Flag = 1
```

Usage

You can use `USAT16` in various DSP algorithms, such as calculating a pixel color component, that require saturation of signed data to an unsigned destination.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

A4.1.140 USUB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	1	1	1	Rm

`USUB16` (Unsigned Subtract) performs two 16-bit unsigned integer subtractions. It sets the GE bits in the CPSR as borrow bits for the subtractions.

Syntax

```
USUB16{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[15:0]  = Rn[15:0] - Rm[15:0]
    GE[1:0]   = if BorrowFrom(Rn[15:0] - Rm[15:0]) then 0 else 0b11
    Rd[31:16] = Rn[31:16] - Rm[31:16]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[31:16]) then 0 else 0b11
```

Usage

`USUB16` produces the same result as `SSUB16` (see *SSUB16* on page A4-180), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.141 USUB8

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	1	1	1	1	Rm

`USUB8` performs four 8-bit unsigned integer subtractions. It sets the GE bits in the CPSR as borrow bits for the subtractions.

Syntax

```
USUB8{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[7:0]   = Rn[7:0] - Rm[7:0]
    GE[0]     = NOT BorrowFrom(Rn[7:0] - Rm[7:0])
    Rd[15:8]  = Rn[15:8] - Rm[15:8]
    GE[1]     = NOT BorrowFrom(Rn[15:8] - Rm[15:8])
    Rd[23:16] = Rn[23:16] - Rm[23:16]
    GE[2]     = NOT BorrowFrom(Rn[23:16] - Rm[23:16])
    Rd[31:24] = Rn[31:24] - Rm[31:24]
    GE[3]     = NOT BorrowFrom(Rn[31:24] - Rm[31:24])
```

Usage

`USUB8` produces the same result as `SSUB8` (see *SSUB8* on page A4-182), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15 Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.142 USUBADDX

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 8	7	6	5	4	3 0
cond	0	1	1	0	0	1	0	1	Rn	Rd	SBO	0	1	0	1	Rm

`USUBADDX` (Unsigned Subtract and Add with Exchange) performs one 16-bit unsigned integer subtraction and one 16-bit unsigned integer addition.

It exchanges the two halfwords of the second operand before it performs the arithmetic.

It sets the GE bits in the CPSR as borrow and carry bits.

Syntax

```
USUBADDX{<cond>}  <Rd>, <Rn>, <Rm>
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    diff      = Rn[31:16] - Rm[15:0]    /* unsigned subtraction */
    Rd[31:16] = diff[15:0]
    GE[3:2]   = if BorrowFrom(Rn[31:16] - Rm[15:0]) then 0b11 else 0
    sum       = Rn[15:0] + Rm[31:16]    /* unsigned addition */
    Rd[15:0]  = sum[15:0]
    GE[1:0]   = if CarryFrom16(Rn[15:0] + Rm[31:16]) then 0b11 else 0
```

Usage

`USUBADDX` produces the same result as `SSUBADDX` (see *SSUBADDX* on page A4-184), but produces GE bit values based on unsigned arithmetic instead of signed arithmetic.

Notes

Use of R15	Specifying R15 for register `<Rd>`, `<Rm>`, or `<Rn>` has UNPREDICTABLE results.

A4.1.143 UXTAB

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	1	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`UXTAB` extracts an 8-bit value from a register, zero extends it to 32 bits, and adds the result to the value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
UXTAB{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
    Rd = Rn + operand2
```

Usage

You can use `UXTAB` to eliminate a separate sign-extension instruction in many instruction sequences that act on **`unsigned char`** values in C/C++.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is an `UXTB` instruction instead, see *UXTB* on page A4-280.

A4.1.144 UXTAB16

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	0	0	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`UXTAB16` extracts two 8-bit values from a register, zero extends them to 16 bits each, and adds the results to the two values from another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

```
UXTAB16{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rn>`	Specifies the register that contains the first operand.
`<Rm>`	Specifies the register that contains the second operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2  = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
    Rd[15:0]  = Rn[15:0]  + operand2[15:0]
    Rd[31:16] = Rn[31:16] + operand2[23:16]
```

Usage

Use `UXTAB16` to keep intermediate values to higher precision while working on arrays of unsigned byte values.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is a `UXTB16` instruction instead, see *UXTB16* on page A4-282.

A4.1.145 UXTAH

31 28	27	26	25	24	23	22	21	20	19 16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	1	1	Rn	Rd	rotate	SBZ	0	1	1	1	Rm

`UXTAH` extracts a 16-bit value from a register, zero extends it to 32 bits, and adds the result to a value in another register. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
UXTAH{<cond>}  <Rd>, <Rn>, <Rm>{, <rotation>}
```

where:

`<cond>` Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.

`<Rd>` Specifies the destination register.

`<Rn>` Specifies the register that contains the first operand.

`<Rm>` Specifies the register that contains the second operand.

`<rotation>` This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    operand2 = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff
    Rd       = Rn + operand2
```

Usage

You can use `UXTAH` to eliminate a separate zero-extension instruction in many instruction sequences that act on **`unsigned short`** values in C/C++.

Notes

Use of R15 Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results.

Note

Your assembler must fault the use of R15 for register `<Rn>`.

Encoding If the `<Rn>` field of the instruction contains `0b1111`, the instruction is a `UXTH` instruction instead, see *UXTH* on page A4-284.

A4.1.146 UXTB

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	1	0	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

`UXTB` extracts an 8-bit value from a register and zero extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit value.

Syntax

```
UXTB{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x000000ff
```

Usage

Use `UXTB` to zero extend a byte to a word, for example in instruction sequences acting on **`unsigned char`** values in C/C++.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.147 UXTB16

31 28	27 26 25 24 23 22 21 20	19 18 17 16	15 12	11 10	9 8	7 6 5 4	3 0
cond	0 1 1 0 1 1 0 0	1 1 1 1	Rd	rotate	SBZ	0 1 1 1	Rm

`UXTB16` extracts two 8-bit values from a register and zero extends them to 16 bits each. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

Syntax

```
UXTB16{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x00ff00ff
```

Usage

Use `UXTB16` to zero extend a byte to a halfword, for example in instruction sequences acting on **`unsigned char`** values in C/C++.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results

A4.1.148 UXTH

31 28	27	26	25	24	23	22	21	20	19	18	17	16	15 12	11 10	9 8	7	6	5	4	3 0
cond	0	1	1	0	1	1	1	1	1	1	1	1	Rd	rotate	SBZ	0	1	1	1	Rm

`UXTH` extracts a 16-bit value from a register and zero extends it to 32 bits. You can specify a rotation by 0, 8, 16, or 24 bits before extracting the 16-bit value.

Syntax

```
UXTH{<cond>}  <Rd>, <Rm>{, <rotation>}
```

where:

`<cond>`	Is the condition under which the instruction is executed. The conditions are defined in *The condition field* on page A3-3. If `<cond>` is omitted, the `AL` (always) condition is used.
`<Rd>`	Specifies the destination register.
`<Rm>`	Specifies the register that contains the operand.
`<rotation>`	This can be any one of:

- `ROR #8`. This is encoded as 0b01 in the rotate field.
- `ROR #16`. This is encoded as 0b10 in the rotate field.
- `ROR #24`. This is encoded as 0b11 in the rotate field.
- Omitted. This is encoded as 0b00 in the rotate field.

Note

If your assembler accepts shifts by `#0` and treats them as equivalent to no shift or `LSL #0`, then it must accept `ROR #0` here. It is equivalent to omitting `<rotation>`.

Architecture version

ARMv6 and above.

Exceptions

None.

Operation

```
if ConditionPassed(cond) then
    Rd[31:0] = (Rm Rotate_Right(8 * rotate)) AND 0x0000ffff
```

Usage

Use `UXTH` to zero extend a halfword to a word, for example in instruction sequences acting on **`unsigned short`** values in C/C++.

Notes

Use of R15	Specifying R15 for register `<Rd>` or `<Rm>` has UNPREDICTABLE results