

A6.7.1 ADC (register)

Add with Carry (register) adds a register value, the carry flag value, and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ADCS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rdn		

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift\_t, shift\_n) = (SRType\_LSL, 0);

Assembler syntax

ADCS{<q>} {<Rd>,<Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is optionally shifted and used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.2 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ADDS <Rd>, <Rn>, #<imm3>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

**Encoding T2** All versions of the Thumb instruction set.  
ADDS <Rdn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

Assembler syntax

ADDS{<q>} {<Rd>,<Rn>, #<const> All encodings permitted

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus immediate)* on page A6-111. If the PC is specified for <Rn>, see *ADR* on page A6-115.
- <const> The immediate value to be added to the value obtained from <Rn>. The range of permitted values is 0-7 for encoding T1, and 0-255 for encoding T2.  
Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.3 ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. Encoding T1 updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ADD{S} <Rd>, <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0		Rm		Rn					Rd

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setFlags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

**Encoding T2** All versions of the Thumb instruction set.  
ADD <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0				Rm				Rdn

DN ┘

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn); n = d; m = UInt(Rm); setFlags = FALSE; (shift_t, shift_n) = (SRTYPE_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

ADD{S}{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S

If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
- {<q>}

See *Standard assembler syntax fields* on page A6-98.
- <Rd>

The destination register. If <Rd> is omitted, this register is the same as <Rn> and encoding T2 is preferred to encoding T1 if both are available. If <Rd> is specified, encoding T1 is preferred to encoding T2. If R<m> is not the PC, the PC can be used in encoding T2.
- <Rn>

The register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus register)* on page A6-113. If R<m> is not the PC, the PC can be used in encoding T2.
- <Rm>

The register that is used as the second operand. The PC can be used in encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result); // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

A6.7.4 ADD (SP plus immediate)

This instruction adds an immediate value to the SP value, and writes the result to the destination register.

**Encoding T1** All versions of the Thumb instruction set.  
ADD <Rd>,SP,#<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	1			Rd								imm8

d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm8:'00', 32);

**Encoding T2** All versions of the Thumb instruction set.  
ADD SP,SP,#<imm7>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	0							imm7

d = 13; setflags = FALSE; imm32 = ZeroExtend(imm7:'00', 32);

Assembler syntax

ADD{<q>} {<Rd>}, SP, #<const>

where:

- <q> See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is SP.
- <const> The immediate value to be added to the value obtained from <Rn>. Permitted values are multiples of 4 in the range 0-1020 for encoding T1 and multiples of 4 in the range 0-508 for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, imm32, '0');
    R[d] = result;

    // no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.5 ADD (SP plus register)

This instruction adds a register value to the SP value, and writes the result to the destination register.

**Encoding T1** All versions of the Thumb instruction set.  
ADD <Rdm>, SP, <Rdm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0		1	1	0	1	Rdm		

DM —

```
d = UInt(DM:Rdm); m = UInt(DM:Rdm); setFlags = FALSE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

**Encoding T2** All versions of the Thumb instruction set.  
ADD SP, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	1	Rm			1 0 1			

```
if Rm == '1101' then SEE encoding T1;
d = 13; m = UInt(Rm); setFlags = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

ADD{<q>} {<Rd>}, SP, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is SP.
- <Rm> The register that is used as the second operand. This register can be the SP, but such instructions are deprecated and the instruction can only be ADD SP, SP, SP.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(SP, shifted, '0');
    if d == 15 then
        ALUWritePC(result);
    else
        R[d] = result;    // no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.6 ADR

Address to Register adds an immediate value to the PC value, and writes the result to the destination register.

**Encoding T1** All versions of the Thumb instruction set.

ADR <Rd>, <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0			Rd								imm8

d = UInt(Rd); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;

Assembler syntax

ADR{<q>} <Rd>, <label>

Normal syntax

ADD{<q>} <Rd>, PC, #<const>

Alternative syntax

where:

- <q>

See *Standard assembler syntax fields* on page A6-98.
- <Rd>

The destination register.
- <label>

The label of an instruction or literal data item whose address is to be loaded into <Rd>. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.  
  
Only a positive value is permitted with imm32 equal to the offset. Permitted values of the offset are multiples of four in the range 0 to 1020 for encoding T1.

In the alternative syntax forms:

<const>

The offset value for the ADD form. Permitted values are multiples of four in the range 0 to 1020 for encoding T1.

Note

It is recommended that the alternative syntax form is avoided where possible.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC, 4) + imm32) else (Align(PC, 4) - imm32);
    R[d] = result;
```

Exceptions

None.

A6.7.7 AND (register)

This instruction performs a bitwise AND of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ANDS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	0	Rm					Rdn

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift\_t, shift\_n) = (SRType\_LSL, 0);

Assembler syntax

ANDS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.8 ASR (immediate)

Arithmetic Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in copies of its sign bit, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ASRS <Rd>, <Rm>, #<imm5>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0					imm5				Rm		Rd

d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();  
(-, shift\_n) = DecodeImmShift('10', imm5);

Assembler syntax

ASRS{<q>} <Rd>, <Rm>, #<imm5>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the first operand.
- <imm5> The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-101.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_ASR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.9 ASR (register)

Arithmetic Shift Right (register) shifts a register value right by a variable number of bits, shifting in copies of its sign bit, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ASRS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0		Rm		Rdn		

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setFlags = !InITBlock();

Assembler syntax

ASRS{<q>} <Rd>, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- <q> See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rn> The register that contains the first operand.
- <Rm> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRTYPE_ASR, shift_n, APSR.C);
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.10 B

Branch causes a branch to a target address.

**Encoding T1** All versions of the Thumb instruction set.  
B<c> <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1					cond						imm8	

```
if cond == '1110' then UNDEFINED;
if cond == '1111' then SEE SVC;
imm32 = SignExtend(imm8:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

**Encoding T2** All versions of the Thumb instruction set.  
B<c> <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0										imm11	

```
imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

B{<c>}{<q>} <label>

where:

<c>{<q>} See *Standard assembler syntax fields* on page A6-98.

- Note**
- Encoding T1 is conditional.
  - For encoding T1, <c> must not be AL or omitted.
  - For ARMv6-M, for encoding T2, <c> must be AL or omitted.

<label> The label of the instruction that is to be branched to. The assembler calculates the required value of the offset from the PC value of the B instruction to this label, then selects an encoding that sets imm32 to that offset.

Permitted offsets are even numbers in the range -256 to 254 for encoding T1 and -2048 to 2046 for encoding T2.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BranchWritePC(PC + imm32);
```

Exceptions

None.

A6.7.11 BIC (register)

Bit Clear (register) performs a bitwise AND of a register value and the complement of an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.

BICS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	0	Rm			Rdn		

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setFlags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

BICS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND NOT(shifted);
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.



A6.7.12 BKPT

Breakpoint causes a HardFault exception or a debug halt to occur depending on the presence and configuration of the debug support.

**Encoding T1** ARMv5T\*, ARMv6-M, ARMv7-M M profile specific behavior  
BKPT #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	imm8							

imm32 = ZeroExtend(imm8, 32);  
// imm32 is for assembly/disassembly only and is ignored by hardware.

Assembler syntax

BKPT{<q>} {#}<imm8>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <imm8> Specifies an 8-bit value that is stored in the instruction. This value is ignored by the ARM hardware, but can be used by a debugger to store additional information about the breakpoint.

Operation

EncodingSpecificOperations();  
BKPTInstrDebugEvent();

Exceptions

HardFault.

A6.7.13 BL

Branch with Link (immediate) calls a subroutine at a PC-relative address.

**Encoding T1** All versions of the Thumb instruction set.  
BL <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	S	imm10										1	1	J1	1	J2	imm11										

I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Assembler syntax

BL{<q>} <label>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <label> The label of the instruction that is to be branched to.  
The assembler calculates the required value of the offset from the PC value of the BL instruction to this label, then selects an encoding that sets imm32 to that offset. Permitted offsets are even numbers in the range -16777216 to 16777214.

Operation

if ConditionPassed() then  
EncodingSpecificOperations();  
next\_instr\_addr = PC;  
LR = next\_instr\_addr<31:1> : '1';  
BranchWritePC(PC + imm32);

Exceptions

None.

Note

Before the introduction of Thumb-2 technology, J1 and J2 in encodings T1 and T2 were both 1, resulting in a smaller branch range. The instructions could be executed as two separate 16-bit instructions, with the first instruction instr1 setting LR to PC + SignExtend(instr1<10:0>:'000000000000', 32) and the second instruction completing the operation. It is no longer possible to split the BL instruction into two 16-bit instructions in ARMv6T2, ARMv6-M and ARMv7.

A6.7.14 BLX (register)

Branch with Link and Exchange calls a subroutine at an address and instruction set specified by a register. ARMv6-M only supports Thumb execution. An attempt to change the instruction execution state causes an exception on the instruction at the target address.

**Encoding T1** ARMv5T\*, ARMv6-M, ARMv7-M  
BLX <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	1		Rm			(0)	(0)	(0)

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

BLX{<q>} <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rm> The register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    target = R[m];
    next_instr_addr = PC - 2;
    LR = next_instr_addr<31:1> : '1';
    BLXWritePC(target);
```

Exceptions

HardFault.

A6.7.15 BX

Branch and Exchange causes a branch to an address and instruction set specified by a register. ARMv6-M only supports Thumb execution. An attempt to change the instruction execution state causes an exception on the instruction at the target address.

BX can also be used for an exception return, see *Exception return behavior* on page B1-227.

**Encoding T1** All versions of the Thumb instruction set.  
BX <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	1	0		Rm			(0)	(0)	(0)

```
m = UInt(Rm);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
if m==15 then UNPREDICTABLE;
```

Assembler syntax

BX{<q>} <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rm> The register that contains the branch target address and instruction set selection bit.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    BXWritePC(R[m]);
```

Exceptions

HardFault.

A6.7.16 CMN (register)

Compare Negative (register) adds a register value and an optionally-shifted register value. It updates the condition flags based on the result, and discards the result.

**Encoding T1** All versions of the Thumb instruction set.  
CMN <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	1	Rm		Rn			

n = UInt(Rn); m = UInt(Rm);  
(shift\_t, shift\_n) = (SRType\_LSL, 0);

Assembler syntax

CMN{<q>} <Rn>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.17 CMP (immediate)

Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result, and discards the result.

**Encoding T1** All versions of the Thumb instruction set.  
CMP <Rn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Rn		imm8								

n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);

Assembler syntax

CMP{<q>} <Rn>, #<const>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rn> The register that contains the operand.
- <const> The immediate value to be added to the value obtained from <Rn>. The range of permitted values is 0-255 for encoding T1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.18 CMP (register)

Compare (register) subtracts an optionally-shifted register value from a register value. It updates the condition flags based on the result, and discards the result.

**Encoding T1** All versions of the Thumb instruction set.  
CMP <Rn>, <Rm> <Rn> and <Rm> both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	1	0	Rm		Rn			

```
n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

**Encoding T2** All versions of the Thumb instruction set.  
CMP <Rn>, <Rm> <Rn> and <Rm> not both from R0-R7

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	1	N	Rm			Rn			

```
n = UInt(N:Rn); m = UInt(Rm);
(shift_t, shift_n) = (SRTYPE_LSL, 0);
if n < 8 && m < 8 then UNPREDICTABLE;
if n == 15 || m == 15 then UNPREDICTABLE;
```

Assembler syntax

CMP{<q>} <Rn>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rn> The register that contains the first operand. The SP can be used.
- <Rm> The register that is optionally shifted and used as the second operand. The SP can be used, but use of the SP is deprecated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    APSR.V = overflow;
```

Exceptions

None.

A6.7.19 CPS

Change Processor State is a system instruction, see *CPS* on page B4-306.

A6.7.20 CPY

Copy is a pre-UAL synonym for MOV (register).

Assembler syntax

CPY <Rd>, <Rn>

This is equivalent to:

MOV <Rd>, <Rn>

Exceptions

None.

A6.7.21 DMB

Data Memory Barrier acts as a memory barrier. It ensures that all explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the processor.

Encoding T1 ARMv6-M, ARMv7-M

DMB #<option>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	1	(1)	(1)	(1)	(1)	1	0	0	(1)	(1)	(1)	(1)	0	1	0	1				option

// No additional decoding required

Assembler syntax

DMB{<q>} {<opt>}

where:

- <q>

See *Standard assembler syntax fields* on page A6-98.
- <opt>

Specifies an optional limitation on the DMB operation:

SY

DMB operation ensures ordering of all accesses, encoded as option == '1111'.  
Can be omitted.

All other encodings of the option are reserved. The corresponding instructions execute as system (SY) DMB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataMemoryBarrier(option);
```

Exceptions

None.

A6.7.22 DSB

Data Synchronization Barrier acts as a special kind of memory barrier. No instruction in program order after this instruction can execute until this instruction completes. This instruction completes only when both:

- any explicit memory access made before this instruction is complete
- all cache and branch predictor maintenance operations before this instruction complete.

**Encoding T1** ARMv6-M, ARMv7-M  
DSB #<option>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	0	0				option

// No additional decoding required

Assembler syntax

DSB{<q>} {<opt>}

where:

- {<q>}

See *Standard assembler syntax fields* on page A6-98.
- <opt>

Specifies an optional limitation on the DSB operation:

SY

DSB operation ensures completion of all accesses, encoded as option == '1111'.  
Can be omitted.

All other encodings of option are reserved. The corresponding instructions execute as system (SY) DSB operations, but software must not rely on this behavior.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    DataSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.23 EOR (register)

Exclusive OR (register) performs a bitwise Exclusive OR of a register value and an optionally-shifted register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
EORS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
0	1	0	0	0	0	0	0	0	1				Rm																		

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setFlags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

EORS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S

The instruction updates the flags.
- {<q>}

See *Standard assembler syntax fields* on page A6-98.
- <Rd>

The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn>

The register that contains the first operand.
- <Rm>

The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] EOR shifted;
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.24 ISB

Instruction Synchronization Barrier flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory after the instruction has completed. It ensures that the effects of context altering operations, such as those resulting from read or write accesses to the system control space (SCS), that completed before the ISB instruction are visible to the instructions fetched after the ISB. See *Barrier support for system correctness* on page B2-255 for more details.

In addition, the ISB instruction ensures that any branches that appear in program order after it are always written into any branch prediction logic with the context that is visible after the ISB instruction. This is required to ensure correct execution of the instruction stream.

**Encoding T1** ARMv6-M, ARMv7-M  
ISB #<option>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	1	1	(1)	(1)	(1)	(1)	1	0	(0)	0	(1)	(1)	(1)	(1)	0	1	1	0				option

// No additional decoding required

Assembler syntax

ISB{<q>} {<opt>}

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <opt> Specifies an optional limitation on the ISB operation. Permitted values are:
  - SY Full system ISB operation, encoded as option == '1111'. Can be omitted.All other encodings of option are reserved. The corresponding instructions execute as full system ISB operations, but must not be relied on by software.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    InstructionSynchronizationBarrier(option);
```

Exceptions

None.

A6.7.25 LDM, LDMIA, LDMFD

Load Multiple Increment After loads multiple registers from consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address above the last of those locations is written back to the base register when the base register is not part of the register list.

**Encoding T1** All versions of the Thumb instruction set.  
LDM <Rn>!,<registers> <Rn> not included in <registers>  
LDM <Rn>,<registers> <Rn> included in <registers>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1			Rn								register_list

```
n = UInt(Rn); registers = '00000000':register_list; wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

LDM{<q>} <Rn>{!}, <registers>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rn> The base register.
- ! Causes the instruction to write a modified value back to <Rn>. If ! is omitted, the instruction does not change <Rn> in this way.
- <registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address.

LDMIA and LDMFD are pseudo-instructions for LDM. LDMFD refers to its use for popping data from Full Descending stacks.



Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 7
        if registers<i> == '1' then
            R[i] = MemA[address,4];  address = address + 4;

    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.26 LDR (immediate)

Load Register (immediate) calculates an address from a base register value and an immediate offset, loads a word from memory, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDR <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	1	imm5					Rn		Rt			

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

**Encoding T2** All versions of the Thumb instruction set.  
LDR <Rt>, [SP{, #<imm8>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	1	Rt					imm8					

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

LDR{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to the value of <Rn> to form the address. Permitted values are multiples of 4 in the range 0-124 for encoding T1 and multiples of 4 in the range 0-1020 for encoding T2. For the offset addressing syntax, <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = MemU[address,4];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.27 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.

LDR <Rt>, <label>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	Rt			imm8							

```
t = UInt(Rt); imm32 = ZeroExtend(imm8:'00', 32); add = TRUE;
```

Assembler syntax

LDR{<q>} <Rt>, <label>	Normal syntax
LDR{<q>} <Rt>, [PC, #<imm>]	Alternative syntax

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <label> The label of the literal data item that is to be loaded into <Rt>. The assembler calculates the required value of the offset from the PC value of this instruction to the label.  
Permitted values of the offset are multiples of four in the range 0 to 1020 for encoding T1.

In the alternative syntax form:

- <imm> The immediate offset added to the `Align(PC, 4)` value of the instruction to form the address. Permitted values are multiples of four in the range 0 to 1020 for encoding T1.

Note

It is recommended that the alternative syntax form is avoided where possible.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    R[t] = MemU[address,4];
```

Exceptions

HardFault.

A6.7.28 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

The register loaded can be the PC. If it is, the word loaded for the PC is treated as an address or exception return value and a branch occurs. Bit [0] complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit [0] is 0, a HardFault exception occurs.

**Encoding T1** All versions of the Thumb instruction set.

LDR <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	0	Rm			Rn			Rt		

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

LDR{<q>} <Rt>, [<Rn>, <Rm>]

where:

- <q> See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = MemU[address,4];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.29 LDRB (immediate)

Load Register Byte (immediate) calculates an address from a base register value and an immediate offset, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRB <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	1	imm5					Rn			Rt		

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5, 32);  
index = TRUE; add = TRUE; wback = FALSE;

Assembler syntax

LDRB{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to or subtracted from the value of <Rn> to form the address. The range of permitted values is 0-31 for encoding T1. <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.30 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRB <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	0	Rm			Rn			Rt		

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

LDRB{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
```

Exceptions

HardFault.

A6.7.31 LDRH (immediate)

Load Register Halfword (immediate) calculates an address from a base register value and an immediate offset, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRH <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	imm5					Rn			Rt		

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);  
index = TRUE; add = TRUE; wback = FALSE;

Assembler syntax

LDRH{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to the value of <Rn> to form the address. Permitted values are multiples of 2 in the range 0-62 for encoding T1. <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

HardFault.

A6.7.32 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRH <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	0	1	Rm			Rn			Rt		

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

LDRH{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

Exceptions

HardFault.

A6.7.33 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRSB <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	1	Rm		Rn		Rt				

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

LDRSB{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
```

Exceptions

HardFault.

A6.7.34 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. Offset addressing is used, see *Memory accesses* on page A6-103 for more information.

**Encoding T1** All versions of the Thumb instruction set.  
LDRSH <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	1	1	1	Rm		Rn		Rt				

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

LDRSH{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The destination register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Exceptions

HardFault.

A6.7.35 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
LSLS <Rd>, <Rm>, #<imm5>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm		Rd			

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd); m = UInt(Rm); setFlags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

Assembler syntax

LSLS{<q>} <Rd>, <Rm>, #<imm5>

where:

- S The instruction updates the flags.
- <q> See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the first operand.
- <imm5> The shift amount, in the range 0 to 31. See *Shifts applied to a register* on page A6-101.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRTYPE_LSL, shift_n, APSR.C);
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.36 LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
LSLS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0	Rm		Rdn			

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setFlags = !InITBlock();
```

Assembler syntax

LSLS{<q>} <Rd>, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- <q> See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rn> The register that contains the first operand.
- <Rm> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRTYPE_LSL, shift_n, APSR.C);
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.37 LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
LSRS <Rd>, <Rm>, #<imm5>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	imm5					Rm		Rd			

```
d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('01', imm5);
```

Assembler syntax

```
LSRS{<q>} <Rd>, <Rm>, #<imm5>
```

where:

- S The instruction updates the flags..
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the first operand.
- <imm5> The shift amount, in the range 1 to 32. See *Shifts applied to a register* on page A6-101.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.38 LSR (register)

Logical Shift Right (register) shifts a register value right by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
LSRS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	1	Rm		Rdn			

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
```

Assembler syntax

```
LSRS{<q>} <Rd>, <Rn>, <Rm>
```

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rn> The register that contains the first operand.
- <Rm> The register whose bottom byte contains the amount to shift by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRType_LSR, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.



A6.7.39 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
MOV S <Rd>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd				imm8						

d = UInt(Rd); setFlags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;

Assembler syntax

MOV S{<q>} <Rd>, #<const>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <const> The immediate value to be placed in <Rd>. The range of permitted values is 0-255 for encoding T1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.40 MOV (register)

Move (register) copies a value from a register to the destination register. Encoding T2 updates the condition flags based on the value.

**Encoding T1** ARMv6-M, ARMv7-M, if <Rd> and <Rm> both from R0-R7.  
MOV <Rd>, <Rm> Otherwise all versions of the Thumb instruction set.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

d = UInt(D:Rd); m = UInt(Rm); setFlags = FALSE;

**Encoding T2** All versions of the Thumb instruction set.  
MOV S <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

d = UInt(Rd); m = UInt(Rm); setFlags = TRUE;

Assembler syntax

MOV{S}{<q>} <Rd>, <Rm>

where:

- {S} If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. This register can be the SP or PC, provided S is not specified. If <Rd> is the PC, the instruction causes a branch to the address moved to the PC.
- <Rm> The source register. This register can be the SP or PC. The instruction must not specify S if <Rm> is the SP or PC.

————— Note —————

- ARM deprecates the use of the following MOV (register) instructions:
- ones in which <Rd> is the SP or PC and <Rm> is also the SP or PC
  - ones in which S is specified and <Rm> is the SP, or <Rm> is the PC.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
result = R[m];
if d == 15 then
    ALUWritePC(result); // setflags is always FALSE here
else
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A6.7.41 MOV (shifted register)

Move (shifted register) is a pseudo-instruction for ASR, LSL, LSR, and ROR.

See the following sections for details:

- *ASR (immediate)* on page A6-117
- *ASR (register)* on page A6-118
- *LSL (immediate)* on page A6-150
- *LSL (register)* on page A6-151
- *LSR (immediate)* on page A6-152
- *LSR (register)* on page A6-153
- *ROR (register)* on page A6-171.

Assembler syntax

Table A6-2 shows the equivalences between MOV (shifted register) and other instructions.

Table A6-2 MOV (shift, register shift) equivalences)

MOV instruction	Canonical form
MOV S <Rd>, <Rm>, ASR #<n>	ASRS <Rd>, <Rm>, #<n>
MOV S <Rd>, <Rm>, LSL #<n>	LSLS <Rd>, <Rm>, #<n>
MOV S <Rd>, <Rm>, LSR #<n>	LSRS <Rd>, <Rm>, #<n>
MOV S <Rd>, <Rm>, ASR <Rs>	ASRS <Rd>, <Rm>, <Rs>
MOV S <Rd>, <Rm>, LSL <Rs>	LSLS <Rd>, <Rm>, <Rs>
MOV S <Rd>, <Rm>, LSR <Rs>	LSRS <Rd>, <Rm>, <Rs>
MOV S <Rd>, <Rm>, ROR <Rs>	RORS <Rd>, <Rm>, <Rs>

The canonical form of the instruction is produced on disassembly.

Exceptions

None.

A6.7.42 MRS

Move to Register from Special register moves the value from the selected special-purpose register into a general-purpose ARM register.

**Encoding T1** ARMv6-M Enhanced functionality in ARMv7-M  
MRS <Rd>, <spec\_reg>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	1	1	(0)	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				SYSm							

MRS is a system instruction except when accessing the APSR or CONTROL register. See *MRS* on page B4-308 for the complete instruction definition, including the application-level uses.

A6.7.43 MSR (register)

Move to Special Register from ARM Register moves the value of a general-purpose ARM register to the specified special-purpose register.

**Encoding T1** ARMv6-M Enhanced functionality in ARMv7-M  
MSR <spec\_reg>, <Rn>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	0	0	(0)	Rn				1	0	(0)	0	(1)	(0)	(0)	(0)	SYSm							

MSR (register) is a system instruction except when accessing the APSR. See *MSR (register)* on page B4-310 for the complete instruction definition, including the application-level uses.

A6.7.44 MUL

Multiply multiplies two register values. The least significant 32 bits of the result are written to the destination register. These 32 bits do not depend on whether signed or unsigned calculations are performed.

The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
MULS <Rdm>, <Rn>, <Rdm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	1	Rn			Rdm		

d = UInt(Rdm); n = UInt(Rn); m = UInt(Rdm); setFlags = !InITBlock();

Assembler syntax

MULS{<q>} {<Rd>}, {<Rn>, <Rm>

where:

- S
- The instruction updates the flags.
- {<q>}
- See *Standard assembler syntax fields* on page A6-98.
- <Rd>
- The destination register.

Note

For ARMv6-M, <Rd> can only be omitted when d == n == m. See *Assembler syntax prototype line conventions* on page A6-96 for the rule on optional arguments.

- <Rn>
- The register that contains the first operand.
- <Rm>
- The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        // APSR.C unchanged
        // APSR.V unchanged
```

Exceptions

None.

A6.7.45 MVN (register)

Bitwise NOT (register) writes the bitwise inverse of a register value to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.

MVNS <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	1	1	Rm			Rd		

```
d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

MVNS{<q>} <Rd>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that is used as the source register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.46 NEG

Negate is a pre-UAL synonym for RSB (immediate) with an immediate value of 0. See *RSB (immediate)* on page A6-172 for details.

Assembler syntax

NEG{<q>} {<Rd>}, <Rm>

This is equivalent to:

RSBS{<q>} {<Rd>}, <Rm>, #0

Exceptions

None.

A6.7.47 NOP

No Operation does nothing. This instruction can be used for code alignment purposes.

This is a NOP-compatible hint, the architected NOP. See *Hint Instructions* on page A6-104 for more information.

See *Pre-UAL pseudo-instruction NOP* on page AppxD-384 for details of NOP before the introduction of UAL.

Note

The timing effects of including a NOP instruction in code are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. NOP instructions are therefore not suitable for timing loops.

Encoding T1 ARMv6-M, ARMv7-M

NOP

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0

// No additional decoding required

Assembler syntax

NOP{<q>}

where:

{<q>} See *Standard assembler syntax fields* on page A6-98.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
// Do nothing
```

Exceptions

None.

A6.7.48 ORR (register)

Logical OR (register) performs a bitwise, inclusive, OR of a register value and an optionally-shifted register value, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
ORRS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	1	0	0	Rm			Rdn		

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

ORRS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.49 POP

Pop Multiple Registers loads a subset, or possibly all, of the general-purpose registers R0-R7 and the PC from the stack.

If the registers loaded include the PC, the word loaded for the PC is treated as a branch address or an exception return value and a branch occurs. Bit [0] complies with the ARM architecture interworking rules for branches to Thumb state execution and must be 1. If bit [0] is 0, a HardFault exception occurs.

**Encoding T1** All versions of the Thumb instruction set.  
POP <registers>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	0	P	register_list							

registers = P:'0000000':register\_list; UnalignedAllowed = FALSE;  
if BitCount(registers) < 1 then UNPREDICTABLE;  
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;

Assembler syntax

POP{<q>} <registers>

where:

{<q>} See *Standard assembler syntax fields* on page A6-98.

<registers>

Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be loaded. The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. If the PC is specified in the register list, the instruction causes a branch to the address (data) loaded into the PC.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP;

    for i = 0 to 7
        if registers<i> == '1' then
            R[i] = MemA[address,4]; address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemA[address,4]);

    SP = SP + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.50 PUSH

Push Multiple Registers stores a subset, or possibly all, of the general-purpose registers R0-R7 and the LR to the stack.

**Encoding T1** All versions of the Thumb instruction set.  
PUSH <registers>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	M	register_list							

```
registers = '0':M:'000000':register_list; UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

PUSH{<q>} <registers>

where:

{<q>} See *Standard assembler syntax fields* on page A6-98.

<registers>  
Is a list of one or more registers, separated by commas and surrounded by { and }. It specifies the set of registers to be stored. The registers are stored in sequence, the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = SP - 4*BitCount(registers);

    for i = 0 to 14
        if registers<i> == '1' then
            MemA[address,4] = R[i];
            address = address + 4;

    SP = SP - 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.51 REV

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

**Encoding T1** ARMv6-M, ARMv7-M  
REV <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	0	Rm				Rd	

d = UInt(Rd); m = UInt(Rm);

Assembler syntax

REV{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<7:0>;
    result<23:16> = R[m]<15:8>;
    result<15:8>  = R[m]<23:16>;
    result<7:0>   = R[m]<31:24>;
    R[d] = result;
```

Exceptions

None.

A6.7.52 REV16

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

**Encoding T1** ARMv6-M, ARMv7-M  
REV16 <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	0	1	Rm				Rd	

d = UInt(Rd); m = UInt(Rm);

Assembler syntax

REV16{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:24> = R[m]<23:16>;
    result<23:16> = R[m]<31:24>;
    result<15:8>  = R[m]<7:0>;
    result<7:0>   = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.



A6.7.53 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.

**Encoding T1** ARMv6-M, ARMv7-M  
REVSH <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	0	1	0	1	1	Rm			Rd		

d = UInt(Rd); m = UInt(Rm);

Assembler syntax

REVSH{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    bits(32) result;
    result<31:8> = SignExtend(R[m]<7:0>, 24);
    result<7:0> = R[m]<15:8>;
    R[d] = result;
```

Exceptions

None.

A6.7.54 ROR (register)

Rotate Right (register) provides the value of the contents of a register rotated by a variable number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left. The variable number of bits is read from the bottom byte of a register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
RORS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	1	Rm			Rdn		

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setFlags = !InITBlock();

Assembler syntax

RORS{<q>} <Rd>, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rn> The register that contains the first operand.
- <Rm> The register whose bottom byte contains the amount to rotate by.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shift_n = UInt(R[m]<7:0>);
    (result, carry) = Shift_C(R[n], SRTYPE_ROR, shift_n, APSR.C);
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.

A6.7.55 RSB (immediate)

Reverse Subtract (immediate) subtracts a register value from an immediate value, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
RSBS <Rd>, <Rn>, #0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	1	Rn			Rd		

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = Zeros(32); // immediate = #0

Assembler syntax

RSBS{<q>} {<Rd>}, <Rn>, #<const>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <const> The immediate value to be added to the value obtained from <Rn>. ARMv6-M only supports a value of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(NOT(R[n]), imm32, '1');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.56 SBC (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT(Carry flag) from a register value, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
SBCS <Rdn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	1	0	Rm			Rdn		

d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

SBCS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.57 SEV

Send Event is a hint instruction. It causes an event to be signaled to all CPUs within a multiprocessor system.  
This is a NOP-compatible hint, see *Hint Instructions* on page A6-104.

**Encoding T1** ARMv6-M, ARMv7-M  
SEV

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	1	0	0	0	0	0	0	0

// No additional decoding required

Assembler syntax

SEV{<q>}

where:

{<q>} See *Standard assembler syntax fields* on page A6-98.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_SendEvent();
```

Exceptions

None.

A6.7.58 STM, STMIA, STMEA

The Store Multiple Increment After and the Store Multiple Empty Ascending instructions store multiple registers to consecutive memory locations using an address from a base register. The sequential memory locations start at this address, and the address above the last of those locations is written back to the base register.

**Encoding T1** All versions of the Thumb instruction set.  
STM <Rn>!,<registers>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	Rn				register_list						

```
n = UInt(Rn); registers = '00000000':register_list; wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

Assembler syntax

STM{IA|EA}{<q>} <Rn>!, <registers>

where:

{<q>} See *Standard assembler syntax fields* on page A6-98.

<Rn> The base register.

! Causes the instruction to write a modified value back to <Rn>.

<registers>  
Is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The lowest-numbered register is stored to the lowest memory address, through to the highest-numbered register to the highest memory address.  
If the base register is included and not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. Use of <Rn> in the register list is deprecated.

STMEA and STMIA are pseudo-instructions for STM, STMEA referring to its use for pushing data onto Empty Ascending stacks.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];

    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemA[address,4] = bits(32) UNKNOWN;    // encoding T1 only
            else
                MemA[address,4] = R[i];
                address = address + 4;

    if wback then R[n] = R[n] + 4*BitCount(registers);
```

Exceptions

HardFault.

A6.7.59 STR (immediate)

Store Register (immediate) calculates an address from a base register value and an immediate offset, and stores a word from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.  
STR <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	0	0	imm5					Rn		Rt			

```
t = UInt(Rt);  n = UInt(Rn);  imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

**Encoding T2** All versions of the Thumb instruction set.  
STR <Rt>, [SP, #<imm8>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	0	Rt		imm8								

```
t = UInt(Rt);  n = 13;  imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE;  add = TRUE;  wback = FALSE;
```

Assembler syntax

STR{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to the value of <Rn> to form the address. Permitted values are multiples of 4 in the range 0-124 for encoding T1 and multiples of 4 in the range 0-1020 for encoding T2. <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.60 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.

STR <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0		Rm		Rn				Rt	

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;   wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

STR{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The register that contains the base value.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,4] = R[t];
```

Exceptions

HardFault.

A6.7.61 STRB (immediate)

Store Register Byte (immediate) calculates an address from a base register value and an immediate offset, and stores a byte from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.  
STRB <Rt>, [<Rn>, #<imm5>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	1	0	imm5			Rn			Rt				

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5, 32);  
index = TRUE; add = TRUE; wback = FALSE;

Assembler syntax

STRB{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to the value of <Rn> to form the address. The range of permitted values is 0-31 for encoding T1. <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.62 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.  
STRB <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rm			Rn			Rt		

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

STRB{<q>} <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The base register.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,1] = R[t]<7:0>;
```

Exceptions

HardFault.

A6.7.63 STRH (immediate)

Store Register Halfword (immediate) calculates an address from a base register value and an immediate offset, and stores a halfword from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.  
STRH <Rt>, [<Rn>{, #<imm5>}]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	imm5					Rn		Rt			

t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);  
index = TRUE; add = TRUE; wback = FALSE;

Assembler syntax

STRH{<q>} <Rt>, [<Rn> {, #+/-<imm>}]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The base register.
- +/- Is + or omitted to indicate that the immediate offset is added to the base register value.  
add == TRUE
- <imm> The immediate offset added to or subtracted from the value of <Rn> to form the address. Permitted values are multiples of 2 in the range 0-62 for encoding T1. <imm> can be omitted, meaning an offset of 0.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;

    if wback then R[n] = offset_addr;
```

Exceptions

HardFault.

A6.7.64 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a register to memory. See *Memory accesses* on page A6-103 for information about memory accesses.

**Encoding T1** All versions of the Thumb instruction set.  
STRH <Rt>, [<Rn>, <Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	1	Rm		Rn		Rt				

t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);  
index = TRUE; add = TRUE; wback = FALSE;  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

STRH{<q>} <Rt>, [<Rn>, <Rm>]

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rt> The source register.
- <Rn> The base register.
- <Rm> Contains the offset that is added to the value of <Rn> to form the address.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, APSR.C);
    address = R[n] + offset;
    MemU[address,2] = R[t]<15:0>;
```

Exceptions

HardFault.

A6.7.65 SUB (immediate)

This instruction subtracts an immediate value from a register value, and writes the result to the destination register. The condition flags are updated based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
SUBS <Rd>, <Rn>, #<imm3>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	imm3			Rn		Rd			

```
d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);
```

**Encoding T2** All versions of the Thumb instruction set.  
SUBS <Rdn>, #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	Rdn			imm8							

```
d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);
```

Assembler syntax

SUBS{<q>} {<Rd>}, <Rn>, #<const>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <const> The immediate value to be subtracted from the value obtained from <Rn>. The range of permitted values is 0-7 for encoding T1 and 0-255 for encoding T2.  
Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.



Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), '1');
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.66 SUB (register)

This instruction subtracts an optionally-shifted register value from a register value, and writes the result to the destination register. It updates the condition flags based on the result.

**Encoding T1** All versions of the Thumb instruction set.  
SUBS <Rd>, <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1		Rm		Rn					Rd

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setFlags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

Assembler syntax

SUBS{<q>} {<Rd>}, <Rn>, <Rm>

where:

- S The instruction updates the flags.
- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is the same as <Rn>.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setFlags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.7.67 SUB (SP minus immediate)

This instruction subtracts an immediate value from the SP value, and writes the result to the destination register.

**Encoding T1** All versions of the Thumb instruction set.  
SUB SP,SP,#<imm7>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0	1	imm7						

d = 13; setflags = FALSE; imm32 = ZeroExtend(imm7: '00', 32);

Assembler syntax

SUB{<q>} {<Rd>}, SP, #<const>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register. If <Rd> is omitted, this register is SP.
- <const> The immediate value to be added to the value obtained from SP. Permitted values are multiples of 4 in the range 0-508 for encoding T1.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(SP, NOT(imm32), '1');
    R[d] = result;

// no flag setting form of the instruction supported
```

Exceptions

None.

A6.7.68 SVC

The Supervisor Call instruction generates a call to a system supervisor, see *Exceptions* on page B1-207 for more information. When the exception is escalated, a HardFault exception is caused.

Use it as a call to an operating system to provide a service.

————— **Note** —————

In older versions of the ARM architecture, SVC was called SWI, Software Interrupt.

**Encoding T1** All versions of the Thumb instruction set M profile specific behavior  
SVC #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	1	imm8							

```
imm32 = ZeroExtend(imm8, 32);
// imm32 is for assembly/disassembly, and is ignored by hardware. SVC handlers in some
// systems interpret imm8 in software, for example to determine the required service.
```

Assembler syntax

SVC{<q>} {#}<imm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <imm> Specifies an 8-bit immediate constant.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CallSupervisor();
```

Exceptions

SVCall, HardFault.

A6.7.69 SXTB

Signed Extend Byte extracts an 8-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6-M, ARMv7-M  
SXTB <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	1	Rm				Rd	

d = UInt(Rd); m = UInt(Rm); rotation = 0;

Assembler syntax

SXTB{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the operand.

Operation

if ConditionPassed() then  
  EncodingSpecificOperations();  
  rotated = ROR(R[m], rotation);  
  R[d] = SignExtend(rotated<7:0>, 32);

Exceptions

None.

A6.7.70 SXTH

Signed Extend Halfword extracts a 16-bit value from a register, sign extends it to 32 bits, and writes the result to the destination register.

Encoding T1 ARMv6-M, ARMv7-M  
SXTH <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	0	0	Rm				Rd	

d = UInt(Rd); m = UInt(Rm); rotation = 0;

Assembler syntax

SXTH{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the operand.

Operation

if ConditionPassed() then  
  EncodingSpecificOperations();  
  rotated = ROR(R[m], rotation);  
  R[d] = SignExtend(rotated<15:0>, 32);

Exceptions

None.

A6.7.71 TST (register)

Test (register) performs a logical AND operation on two register values. It updates the condition flags based on the result, and discards the result.

**Encoding T1** All versions of the Thumb instruction set.  
TST <Rn>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	1	0	0	0	Rm			Rn		

n = UInt(Rn); m = UInt(Rm);  
(shift\_t, shift\_n) = (SRTYPE\_LSL, 0);

Assembler syntax

TST{<q>} <Rn>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rn> The register that contains the first operand.
- <Rm> The register that is used as the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, APSR.C);
    result = R[n] AND shifted;
    APSR.N = result<31>;
    APSR.Z = IsZeroBit(result);
    APSR.C = carry;
    // APSR.V unchanged
```

Exceptions

None.

A6.7.72 UDF

Permanently Undefined generates an Undefined Instruction exception.

————— **Note** —————

The encodings for UDF are defined as permanently undefined in the versions of the architecture specified in this section. Issue C of this manual first defines an assembler mnemonic for these encodings.

**Encoding T1** All versions of the Thumb instruction set.  
UDF #<imm8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	0	imm8							

imm32 = ZeroExtend(imm8, 32);  
// imm32 is for assembly and disassembly only, and is ignored by hardware.

**Encoding T2** ARMv6-M, ARMv7-M  
UDF.W #<imm16>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	1	1	1	1	1	imm4				1	0	1	0	imm12											

imm32 = ZeroExtend(imm4:imm12, 32);  
// imm32 is for assembly and disassembly only, and is ignored by hardware.

Assembler syntax

UDF{<q>} {#}<imm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <imm> Specifies an immediate constant, that is 8-bit in encoding T1, and 16-bit in encoding T2. The processor ignores the value of this constant.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    UNDEFINED;
```

Exceptions

Undefined Instruction.

A6.7.73 UXTB

Unsigned Extend Byte extracts an 8-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.

**Encoding T1** ARMv6-M, ARMv7-M  
UXTB <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	1	Rm				Rd	

```
d = UInt(Rd); m = UInt(Rm); rotation = 0;
```

Assembler syntax

UXTB{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Exceptions

None.

A6.7.74 UXTH

Unsigned Extend Halfword extracts a 16-bit value from a register, zero extends it to 32 bits, and writes the result to the destination register.

**Encoding T1** ARMv6-M, ARMv7-M  
UXTH <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	0	1	0	1	0	Rm		Rd			

d = UInt(Rd); m = UInt(Rm); rotation = 0;

Assembler syntax

UXTH{<q>} <Rd>, <Rm>

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.
- <Rd> The destination register.
- <Rm> The register that contains the second operand.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Exceptions

None.

A6.7.75 WFE

Wait For Event is a hint instruction that permits the processor to enter a low-power state until one of a number of events occurs, including events signaled by the SEV instruction on any processor in a multiprocessor system. For more information, see *Wait For Event and Send Event* on page B1-241.

For general hint behavior, see *Hint Instructions* on page A6-104.

**Encoding T1** ARMv6-M, ARMv7-M  
WFE

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0

// No additional decoding required

Assembler syntax

WFE{<q>}

where:

- {<q>} See *Standard assembler syntax fields* on page A6-98.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if EventRegistered() then
        ClearEventRegister();
    else
        WaitForEvent();
```

Exceptions

None.

A6.7.76 WFI

Wait For Interrupt is a hint instruction that suspends execution until one of a number of events occurs. For more information, see *Wait For Interrupt* on page B1-243.

For general hint behavior, see *Hint Instructions* on page A6-104.

**Encoding T1**                  ARMv6-M, ARMv7-M  
WFI

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	1	1	0	0	0	0	0

// No additional decoding required

Assembler syntax

WFI{<q>}

where:

{<q>}                  See *Standard assembler syntax fields* on page A6-98.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    WaitForInterrupt();
```

Exceptions

None.

Notes

**PRIMASK**      If PRIMASK.PM is set to 1, an asynchronous exception that has a higher group priority than any active exception results in a WFI instruction exit. If the group priority of the exception is less than or equal to the execution group priority, the exception is ignored.

A6.7.77 YIELD

YIELD is a hint instruction. It enables software with a multithreading capability to indicate to the hardware that it is performing a task, for example a spinlock, that could be swapped out to improve overall system performance. Hardware can use this hint to suspend and resume multiple code threads if it supports the capability.

For general hint behavior, see *Hint Instructions* on page A6-104.

**Encoding T1**                  ARMv6-M, ARMv7-M  
YIELD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	0	0	0	1	0	0	0	0	0

// No additional decoding required

Assembler syntax

YIELD{<q>}

where:

{<q>}                  See *Standard assembler syntax fields* on page A6-98.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    Hint_Yield();
```

Exceptions

None.