Arm® Architecture Reference Manual Supplement, Custom Datapath Extension for Armv8-M

arm

Release information

Date	Version	Changes
17/12/2019	A.a Non-confidential-Beta	• Beta release

Armv8-M Architecture Reference Manual

Product Status

The information in this document is for a Beta product, that is a product under development.

Web Address

`http://www.arm.com`

Contents

Arm® Architecture Reference Manual Supplement, Custom Datapath Extension for Armv8-M

Preface

Part A   Custom Datapath Extension for the Armv8-M Architecture Introduction and Overview

Part B   Custom Datapath Extension for the Armv8-M Architecture, Programmers' Model

Part C   Instruction Specification

Part D   Pseudocode Specification

Chapter D1   **Pseudocode Specification**

Glossary

Preface

This preface introduces the Armv8-M Custom Datapath Extension. It contains the following sections:

Introduction to CDE

Custom Datapath Extension for the Armv8-M Architecture, Programmers' Model

Alphabetical list of instructions

Pseudocode Specification

Additional reading

This section lists relevant publications from Arm and third parties.

See https://developer.arm.com, for access to Arm documentation.

Arm publications

- *Arm®v8-M Architecture Reference Manual* (ARM DDI 0553B.j)

Part A
Custom Datapath Extension for the Armv8-M Architecture
Introduction and Overview

Chapter A1
Introduction

This chapter introduces the Custom Datapath Extension for the Armv8-M architecture. It contains the following sections:

A1.1 Document layout and terminology

This section describes the structure and scope of this supplement. This section also describes the terminology that this supplement uses. It does not constitute part of the supplement, and must not be interpreted as implementation guidance.

A1.1.1 Structure of the document

This supplement describes the behavior of the Custom Datapath Extension as a set of individual rules.

Each rule is clearly identified by the letter R, followed by a random group of subscript letters that do not reflect any intended order or priority, for example R_{BSHJ}. In the following example, R_{BSHJ} is simply a random rule identifier that has no significance apart from uniquely identifying a rule in this supplement.

Identifier   Rule

R_{BSHJ} The following data accesses are single-copy atomic:

- All byte accesses.
- All halfword accesses to halfword-aligned locations.
- All word accesses to word-aligned locations.

Applies to an implementation of the architecture from Armv8.0-M onwards

Additional Information
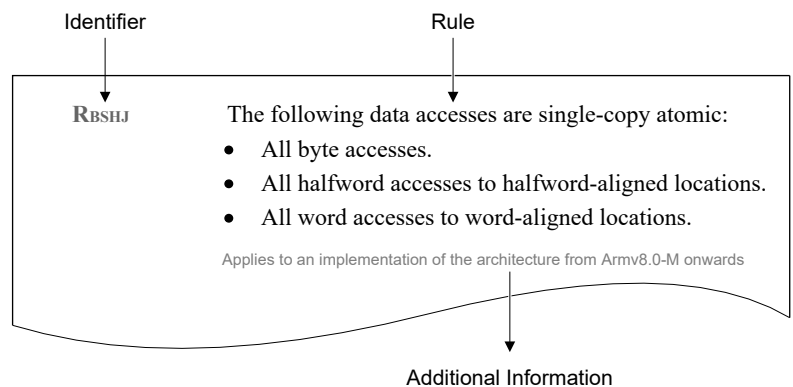
Rules must not be read in isolation, and where more than one rule relating to a particular feature exists, individual rules are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader.

An implementation that conforms to all the rules described in this supplement constitutes an Custom Datapath Extension for Armv8-M compliant implementation. An implementation whose behavior deviates from these rules is not compliant with the Custom Datapath Extension for Armv8-M architecture.

Some sections contain additional information and guidance that do not constitute rules. This information and guidance is provided purely as an aid to understanding the architecture. Information statements are clearly identified by the letter I, followed by a random group of subscript letters, for example I_{PRTD}.

Note

Arm strongly recommends that implementers read *all* chapters and sections of this document to ensure that an implementation is compliant.

An implementation that conforms to all the rules described in this specification but chooses to ignore any additional information and guidance is compliant with the Armv8-M architecture.

In the following parts of this supplement, architectural rules are not identified by a specific prefix and a random group of subscript letters:

- Part C Custom Datapath Extension for Armv8-M Instruction Set.

A1.1.2 Scope of the document

This supplement contains only rules and information that relate specifically to the Custom Datapath Extension for Armv8-M architecture. It does not include any information about other Arm architectures, nor does it describe the full details of the Armv8-M architecture.

A1.1.3 Intended audience

This supplement is written for users who want to design, implement, or program the Custom Datapath Extension for Armv8-M. This supplement is not a full description of the Armv8-M architecture.

The supplement provides a precise, accurate, and correct set of rules that must be followed in order for a Custom Datapath Extension for Armv8-M implementation to be architecturally compliant. It is an explicit reference supplement, and not a general introduction to, or user guide for, the Custom Datapath Extension for Armv8-M architecture.

A1.1.4 Terminology, phrases

This subsection identifies some standard words and phrases that are used in the Arm architecture documentation. These words and phrases have an Arm-specific definition, which is described in this section.

Architecturally visible

Something that is visible to the controlling agent. The controlling agent might be software.

Arm recommends

A particular usage that ensures consistency and usability. Following all the rules listed in this supplement leads to a predictable outcome that is compliant with the architecture, but might produce an unexpected output. Adhering to a recommendation ensures that the output is as expected.

Arm strongly recommends

Something that is essentially mandatory, but that is outside the scope of the architecture described in this supplement. Failing to adhere to a strong recommendation can break the system, although the PE itself remains compliant with the architecture that is described in this supplement.

Finite time

An action will occur at some point in the future. Finite time does not make any statement about the time involved. However, delaying an action longer than is absolutely necessary might have an adverse impact on performance.

Permitted

Allowed behavior.

Required

Mandatory behavior.

Support

The implementation has implemented a particular feature.

A1.1.5 Terminology, Armv8-M specific terms

For definitions of Custom Datapath Extension specific terms, see the Glossary.

A1.2 The Custom Datapath Extension for Armv8-M

The Custom Datapath Extension is an OPTIONAL feature available from Armv8-M architecture. An implementation that includes the Custom Datapath Extension must implement the features that are provided by the Main Extension (M), and might implement the following OPTIONAL features:

- The features that are provided by the Floating-point Extension (FP). Instructions that operate on the S or D register file require FP or MVE.
- The features that are provided by the Armv8.1 M-Profile Vector Extension (MVE). Instructions that operate on the Q register file require MVE.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

Where applicable, a line below each rule or information statement indicates the extensions that are required for the rule or information statement to apply, and any other notes.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

A line below each rule or information statement indicates the architecture version, the extensions that are required for the rule or information statement to apply, and any other notes. Some extensions depend on the implementation of other extensions, for example FP.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards.*

A1.3 Overview of the Custom Datapath Extension

I_{QNBG} The Custom Datapath Extension (CDE) for the Armv8-M architecture introduces three classes of two instructions in the co-processor instruction space:

- Three classes operate on the general-purpose register file, including the condition code flags `APSR_nzcv`.
- Three classes operate on the Floating-point or SIMD register file only.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**. *Note, (FP || MVE) required for Floating-point register file. MVE is only available in an Armv8.1-M implementation.*

I_{JWBF} A Custom Datapath instruction operating on the Floating-point or SIMD register files uses one of:

- 32-bit S registers.
- 64-bit D registers.
- 128-bit Q registers.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE && (FP || MVE)**. *Note, Q registers require MVE, MVE is only available in an Armv8.1-M implementation.*

I_{RBLJ} The three classes are defined by the following instruction patterns:

- `<operation code> <destination register>`.
- `<operation code> <destination register>,<source register>`.
- `<operation code> <destination register>,<source register 1>,<source register 2>`.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**.

I_{FZRZ} The destination register of an instruction might be optionally read, as well as written.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**.

I_{BXBG} The operation code can be split between a true operation code in the custom datapath and an immediate value used in the custom datapath. The architecture does not prescribe any split.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**.

I_{NFMB} Immediate consequences of the above are:

- No operations on the Floating-point or SIMD registers can set condition codes.
- There are no instructions that support the use of all of, or any combination of, S registers, D registers, Q registers and, the general-purpose register file.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**. *Note, Q registers require MVE, MVE is only available in an Armv8.1-M implementation.*

I_{TVRP} Operations on the general-purpose register file operate on 32-bit registers, or a dual-register consisting of a 64-bit value constructed from an even numbered general-purpose register and its immediately following odd numbered pair.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are -* **CDE**.

Part B
Custom Datapath Extension for the Armv8-M Architecture, Programmers' Model

Chapter B1
Programmers' Model

This chapter specifies the rules associated with the Custom Datapath Extension (CDE) for Armv8-M. It contains the following sections:

B1.1 Enabling CDE instructions

I_{CXBC} Custom Datapath instructions can be found within, and are associated with, the existing coprocessor encoding and numbering spaces.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{XLTS} Custom Datapath instructions fall into encoding spaces associated with a *coprocessor* number in the range 0 to 7 inclusive.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

I_{WWNQ} Enabling the coprocessor space in which the Custom Datapath Extension is implemented is the same as other IMPLEMENTATION DEFINED coprocessors. The function `IsCPEnabled()` describes this.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**. Note, S required for Secure state.*

R_{VSVW} If a coprocessor is associated with the Custom Datapath Extension, that coprocessor cannot execute the following instructions:

- `CDP, CDP2`.
- `LDC, LDC2 (immediate)`.
- `LDC, LDC2 (literal)`.
- `MCR, MCR2`.
- `MCRR, MCRR2`.
- `MRC, MRC2`.
- `MRRC, MRRC2`.
- `STC, STC2`.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{SKRX} Execution of a Custom Datapath instruction that accesses the Floating-point or SIMD register file causes *Lazy Floating-point stacking* as specified by the architecture.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && (**FP** || **MVE**). Note, MVE is only available in an Armv8.1-M implementation.*

I_{MVXW} Armv8-M Double-precision Floating-point Extension implements 16 "D" registers, D0 to D15. The instructions defined by the Custom Datapath Extension are capable of indexing registers D0 to D31.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && (**FP** || **MVE**). Note, MVE is only available in an Armv8.1-M implementation.*

I_{LPPY} Armv8.1-M MVE implements eight "Q" registers, Q0 to Q7. The instructions defined by the Custom Datapath Extension are capable of indexing registers Q0 to Q15.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && **MVE**. Note, MVE is only available in an Armv8.1-M implementation.*

R_{LBNN} Execution of a Custom Datapath instruction that attempts to access an unimplemented Floating-point or SIMD register, is CONSTRAINED UNPREDICTABLE and either of the following behaviors can occur:

- The instruction is UNDEFINED.
- The instruction is treated as a NOP.

> *Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && (**FP** || **MVE**). Note, MVE is only available in an Armv8.1-M implementation.*

B1.2 Execution of CDE instructions

R_{QGNK} The source and destination registers for any Custom Datapath instruction are restricted to those specified by the instruction pseudocode.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{GPLC} The operation of a Custom Datapath instruction cannot be stateful, and cannot operate directly on memory.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{XFCV} It is IMPLEMENTATION DEFINED which Custom Datapath instructions are implemented.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{SVBN} An unimplemented Custom Datapath instruction whose associated coprocessor is not disabled is UNDEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{SGPM} Which coprocessors adhere to the Custom Datapath Extension or the Arm architecture coprocessor instruction set is IMPLEMENTATION DEFINED.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

R_{DVDG} If the Performance Monitors Extension is implemented only the instruction counter, Cycle counter and, IMPLE-MENTATION DEFINED counters increment on execution of Custom Datapath Extension instructions. There are no architected PMU events for Custom Datapath Extension instructions.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && **PMU**. Note, PMU is only available in an Armv8.1-M implementation.*

R_{VPLL} When executing a CDE scalar dual instruction the CDE enabled coprocessor must process general-purpose register pairs according to the PE's current endianness.

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE**.*

I_{RRTS} All of the rules required for the M-Profile Vector Extension and the Low Overhead Loop and Branch Future Extension apply to all CDE beat-wise compatible instructions.

This includes the following, but is not limited to:

- Exception continuable behavior.
- Overlapping of beat-wise instructions.
- VPT predication.
- Tail predicated low overhead loops.

The CDE instructions are as follows:

- VCX1 (vector)
- VCX2 (vector)
- VCX3 (vector)

*Applies to an implementation of the architecture from **Armv8.0-M** onwards. The extension requirements are - **CDE** && **MVE** && **LOB**. Note, LOB && MVE are only available in an Armv8.1-M implementation.*

Part C
Instruction Specification

Chapter C1
Alphabetical list of instructions

Instructions relevant to this Extension are listed in this section. For the full list of Armv8-M instruction see **Armv8-M Architecture Reference Manual**.

C1.0.1 CDP, CDP2

Coprocessor Data Processing. Coprocessor Data Processing tells a coprocessor to perform an operation.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0	CRm

T1 variant

```
CDP{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  cp = UInt(coproc);
```

T2

Armv8-M Main Extension only

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	1 1 1 0	opc1	CRn	CRd	coproc	opc2	0	CRm

T2 variant

```
CDP2{<c>}{<q>} <coproc>, {#}<opc1>, <CRd>, <CRn>, <CRm> {, {#}<opc2>}
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  cp = UInt(coproc);
```

Assembler symbols for all encodings

<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<opc1>	Is a coprocessor-specific opcode, in the range 0 to 15, encoded in the "opc1" field.
<CRd>	Is the destination coprocessor register, encoded in the "CRd" field.
<CRn>	Is the coprocessor register that contains the first operand, encoded in the "CRn" field.
<CRm>	Is the coprocessor register that contains the second operand, encoded in the "CRm" field.
<opc2>	Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4      if !Coproc_Accepted(cp, ThisInstr()) then
5          GenerateCoprocessorException();
6      else
7          Coproc_InternalOperation(cp, ThisInstr());
```

C1.0.2 CX1

Custom Instruction Class 1. Custom instruction class 1 computes a value based on an immediate, and optionally the destination value, and writes the result to the destination register.

The source and destination registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0	0 0	op1	Rd	0 coproc	op2	0	op3

Accumulator variant

Applies when `A == 1`

```
CX1A <coproc>, <Rd>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX1 <coproc>, <Rd>, #<imm>
```

Decode for this encoding

```
1  if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2  if !HaveMainExt()                        then UNDEFINED;
3  cp  = UInt(coproc);
4  d   = UInt(Rd);
5  imm = op1:op2:op3;
6  acc = (A == '1');
7  ExecuteCPCheck(cp);
8  if d == 13                               then UNPREDICTABLE;
9  if !acc && InITBlock()                   then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction returns an UNKNOWN value in SP.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE behavior

If `!acc && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as `NOP`.

Assembler symbols for all encodings

<A>	Accumulate with existing register contents. This parameter must be one of the following values:

    "   Encoded as `A = 0`
    A   Encoded as `A = 1`

<coproc>	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
<Rd>	Is the general-purpose `R0 - R14` or `APSR_nzcv` destination register, encoded in the "Rd" field. For accumulator variants <Rd> also specifies the source register. `APSR_nzcv` is encoded by the "Rd" field value `0b1111`.
<imm>	Is the immediate encoded in `op1:op2:op3`.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if (!Coproc_Accepted(cp, ThisInstr())) then
4          GenerateCoprocessorException();
5      elsif acc then
6          RF[d] = CX_op1(ThisInstr(), RF[d], 32);
7      else
8          RF[d] = CX_op0(ThisInstr(), 32);
```

C1.0.3 CX1D

Custom Instruction Class 1. Custom instruction class 1 dual computes a value based on an immediate, and optionally the destination register pair value, and writes the result to a destination register pair.

The destination registers are a consecutive pair of general-purpose registers.

The significance of the words in each pair is consistent with the current data endianness.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0	0 0	op1	Rd	0	coproc	op2	1	op3

Accumulator variant

Applies when `A == 1`

```
CX1DA <coproc>, <Rd>, <Rd+1>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX1D <coproc>, <Rd>, <Rd+1>, #<imm>
```

Decode for this encoding

```
1  if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2  if !HaveMainExt()                       then UNDEFINED;
3  cp  = UInt(coproc);
4  d   = UInt(Rd);
5  d2  = d + 1;
6  imm = op1:op2:op3;
7  acc = (A == '1');
8  ExecuteCPCheck(cp);
9  // Register pairs containing SP or PC are UNPREDICTABLE.
10 if d > 10                               then UNPREDICTABLE;
11 if Rd[0] == '1'                         then UNPREDICTABLE;
12 if !acc && InITBlock()                  then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d is odd`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction returns and UNKNOWN value in Rd, Rd + 1 and Rd - 1.
- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 12`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction returns and UNKNOWN value in R12 and SP.
- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 12 || d == 13`, then one of the following behaviors must occur:

- It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE behavior

If `d == 14`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `!acc && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

`<A>`	Accumulate with existing register contents. This parameter must be one of the following values:
	`"` Encoded as `A = 0`
	`A` Encoded as `A = 1`
`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Rd>`	Is the general-purpose register `R0 - R11` specifying the first of destination register pair, encoded in the "Rd" field. For accumulator variants, `<Rd>` also specifies the source register.
`<imm>`	Is the immediate encoded in `op1:op2:op3`.

Operation for all encodings

```
1   if ConditionPassed() then
2       EncodingSpecificOperations();
3       if (!Coproc_Accepted(cp, ThisInstr())) then
4           GenerateCoprocessorException();
5       elsif acc then
6           RFD[d] = CX_op1(ThisInstr(), RFD[d], 64);
7       else
8           RFD[d] = CX_op0(ThisInstr(), 64);
```

C1.0.4 CX2

Custom Instruction Class 2. Custom instruction class 2 computes a value based on a source register, an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0 0 1	op1	Rn	Rd	0	coproc	op2	0	op3

Accumulator variant

Applies when `A == 1`

```
CX2A <coproc>, <Rd>, <Rn>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX2 <coproc>, <Rd>, <Rn>, #<imm>
```

Decode for this encoding

```
1  if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2  if !HaveMainExt()                        then UNDEFINED;
3  cp  = UInt(coproc);
4  d   = UInt(Rd);
5  n   = UInt(Rn);
6  imm = op1:op2:op3;
7  acc = (A == '1');
8  ExecuteCPCheck(cp);
9  if d == 13 || n == 13                    then UNPREDICTABLE;
10 if !acc && InITBlock()                    then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction returns an UNKNOWN value in SP.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE behavior

If `n == 13`, then one of the following behaviors must occur:

- The instruction executes as NOP.

- The instruction returns an UNKNOWN value.

- The instruction is UNDEFINED.

CONSTRAINED UNPREDICTABLE behavior

If `!acc && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

`<A>`	Accumulate with existing register contents. This parameter must be one of the following values:
	   `"` Encoded as `A = 0`
	   `A` Encoded as `A = 1`

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Rd>`	Is the general-purpose `R0 - R14` or `APSR_nzcv` destination register, encoded in the "Rd" field. For accumulator variants `<Rd>` also specifies the source register. `APSR_nzcv` is encoded by the "Rd" field value `0b1111`.
`<Rn>`	Is the general-purpose `R0 - R14` or `APSR_nzcv` source register, encoded in the "Rn" field. `APSR_nzcv` is encoded by the "Rn" field value `0b1111`.
`<imm>`	Is the immediate encoded in `op1:op2:op3`.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if (!Coproc_Accepted(cp, ThisInstr())) then
4          GenerateCoprocessorException();
5      elsif acc then
6          RF[d] = CX_op2(ThisInstr(), RF[d], RF[n], 32);
7      else
8          RF[d] = CX_op1(ThisInstr(), RF[n], 32);
```

C1.0.5 CX2D

Custom Instruction Class 2. Custom instruction class 2 dual computes a value based on a source register, an immediate, and optionally the destination register pair value, and writes the result to the destination register pair.

The destination registers are a consecutive pair of general-purpose registers.

The source registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

The significance of the words in each pair is consistent with the current data endianness.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0 0 1	op1	Rn	Rd	0 coproc	op2	1	op3

Accumulator variant

Applies when `A == 1`

```
CX2DA <coproc>, <Rd>, <Rd+1>, <Rn>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX2D <coproc>, <Rd>, <Rd+1>, <Rn>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   if !HaveMainExt()                       then UNDEFINED;
3   cp  = UInt(coproc);
4   d   = UInt(Rd);
5   n   = UInt(Rn);
6   imm = op1:op2:op3;
7   acc = (A == '1');
8   ExecuteCPCheck(cp);
9   // Register pairs containing SP or PC are UNPREDICTABLE.
10  if d > 10                               then UNPREDICTABLE;
11  if n == 13                              then UNPREDICTABLE;
12  if Rd[0] == '1'                         then UNPREDICTABLE;
13  if !acc && InITBlock()                  then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d is odd`, then one of the following behaviors must occur:

* The instruction is UNDEFINED.

* The instruction returns and UNKNOWN value in Rd, Rd + 1 and Rd - 1.

* The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 12`, then one of the following behaviors must occur:

* The instruction is UNDEFINED.

* The instruction returns and UNKNOWN value in R12 and SP.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE **behavior**

If d == 12 || d == 13, then one of the following behaviors must occur:

- It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE **behavior**

If d == 14, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE **behavior**

If n == 13, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The value in the destination register is UNKNOWN.
- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE **behavior**

If !acc && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP.

Assembler symbols for all encodings

<A>   Accumulate with existing register contents. This parameter must be one of the following values:

> "   Encoded as A = 0
> A   Encoded as A = 1

<coproc>   Is the name of the coprocessor, encoding in the "coproc" field. Valid names are p0 to p7 inclusive.

<Rd>   Is the general-purpose register R0 - R11 specifying the first of destination register pair, encoded in the "Rd" field. For accumulator variants, <Rd> also specifies the source register.

<Rn>   Is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "Rn" field. APSR_nzcv is encoded by the "Rn" field value 0b1110.

<imm>   Is the immediate encoded in op1:op2:op3.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if (!Coproc_Accepted(cp, ThisInstr())) then
4          GenerateCoprocessorException();
5      elsif acc then
6          RFD[d] = CX_op2(ThisInstr(), RFD[d], RF[n], 64);
7      else
8          RFD[d] = CX_op1(ThisInstr(), RF[n], 64);
```

C1.0.6 CX3

Custom Instruction Class 3. Custom instruction class 3 computes a value based two source registers, an immediate and optionally the destination value, and writes the result to the destination register.

The source and destination registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0 1	op1	Rn	Rm	0	coproc	op2 0	op3	Rd

Accumulator variant

Applies when `A == 1`

```
CX3A <coproc>, <Rd>, <Rn>, <Rm>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX3 <coproc>, <Rd>, <Rn>, <Rm>, #<imm>
```

Decode for this encoding

```
1  if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2  if !HaveMainExt()                        then UNDEFINED;
3  cp  = UInt(coproc);
4  d   = UInt(Rd);
5  n   = UInt(Rn);
6  m   = UInt(Rm);
7  imm = op1:op2:op3;
8  acc = (A == '1');
9  ExecuteCPCheck(cp);
10 if d == 13 || n == 13 || m == 13        then UNPREDICTABLE;
11 if !acc && InITBlock()                  then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction returns an UNKNOWN value in SP.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 13`, then one of the following behaviors must occur:

- It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE behavior

If `n == 13 or m == 13`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction returns and UNKNOWN value in SP.

- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If !acc && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

<A>	Accumulate with existing register contents. This parameter must be one of the following values:
	   " Encoded as A = 0
	   A Encoded as A = 1

<coproc>	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are p0 to p7 inclusive.
<Rd>	Is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "Rd" field. For accumulator variants <Rd> also specifies the source register. APSR_nzcv is encoded by the "Rd" field value 0b1111.
<Rn>	Is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "Rn" field. APSR_nzcv is encoded by the "Rn" field value 0b1111.
<Rm>	Is the general-purpose R0 - R14 or APSR_nzcv destination register, encoded in the "Rm" field. APSR_nzcv is encoded by the "Rm" field value 0b1111.
<imm>	Is the immediate encoded in op1:op2:op3.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if (!Coproc_Accepted(cp, ThisInstr())) then
4          GenerateCoprocessorException();
5      elsif acc then
6          RF[d] = CX_op3(ThisInstr(), RF[d], RF[n], RF[m], 32);
7      else
8          RF[d] = CX_op2(ThisInstr(), RF[n], RF[m], 32);
```

C1.0.7 CX3D

Custom Instruction Class 3. Custom instruction class 3 dual computes a value based on two source registers, an immediate, and optionally the destination register pair value, and writes the result to the destination register pair.

The source registers can be either general-purpose registers or the Condition flags, specified by use of APSR_nzcv.

The destination registers are a consecutive pair of general-purpose registers.

The significance of the words in each pair is consistent with the current data endianness.

T1

Armv8-M Custom Datapath Extension

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 A	1 1 1 0 1	op1	Rn	Rm	0 coproc	op2 1	op3	Rd

Accumulator variant

Applies when `A == 1`

```
CX3DA <coproc>, <Rd>, <Rd+1>, <Rn>, <Rm>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
CX3D <coproc>, <Rd>, <Rd+1>, <Rn>, <Rm>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   if !HaveMainExt()                        then UNDEFINED;
3   cp  = UInt(coproc);
4   d   = UInt(Rd);
5   n   = UInt(Rn);
6   m   = UInt(Rm);
7   imm = op1:op2:op3;
8   acc = (A == '1');
9   ExecuteCPCheck(cp);
10  // Register pairs containing SP or PC are UNPREDICTABLE.
11  if d > 10                                then UNPREDICTABLE;
12  if n == 13 || m == 13                    then UNPREDICTABLE;
13  if Rd[0] == '1'                          then UNPREDICTABLE;
14  if !acc && InITBlock()                   then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `d is odd`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction returns and UNKNOWN value in Rd, Rd + 1 and Rd - 1.
- The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If `d == 12`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction returns and UNKNOWN value in SP.

• The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If d == 12 || d == 13, then one of the following behaviors must occur:

• It is UNKNOWN whether a stack limit check is performed.

CONSTRAINED UNPREDICTABLE behavior

If d == 14, then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If n == 13 || m == 12, then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The value in the destination register is UNKNOWN.

• The instruction executes as NOP.

CONSTRAINED UNPREDICTABLE behavior

If !acc && InITBlock(), then one of the following behaviors must occur:

• The instruction is UNDEFINED.

• The instruction executes as if it passes the Condition code check.

• The instruction executes as NOP.

Assembler symbols for all encodings

<A>	Accumulate with existing register contents. This parameter must be one of the following values:
	   " Encoded as A = 0
	   A Encoded as A = 1

<coproc>	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are p0 to p7 inclusive.
<Rd>	Is the general-purpose register R0 - R11 specifying the first of destination register pair, encoded in the "Rd" field. For accumulator variants, <Rd> also specifies the source register.
<Rn>	Is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "Rn" field. APSR_nzcv is encoded by the "Rn" field value 0b1110.
<Rm>	Is the general-purpose R0 - R14 or APSR_nzcv source register, encoded in the "Rm" field. APSR_nzcv is encoded by the "Rm" field value 0b1110.
<imm>	Is the immediate encoded in op1:op2:op3.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      if (!Coproc_Accepted(cp, ThisInstr())) then
4          GenerateCoprocessorException();
5      elsif acc then
6          RFD[d] = CX_op3(ThisInstr(), RFD[d], RF[n], RF[m], 64);
7      else
8          RFD[d] = CX_op2(ThisInstr(), RFD[n], RF[m], 64);
```

C1.0.8 LDC, LDC2 (immediate)

Load Coprocessor (immediate). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. Some of the fields have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer. These fields are the D bit, the CRd field, and in the Unindexed addressing mode only, the imm8 field.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 0	P	U	D	W	1	Rn != 1111	CRd	coproc	imm8

Offset variant

Applies when `P == 1 && W == 0`.

`LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]`

Post-indexed variant

Applies when `P == 0 && W == 1`.

`LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>`

Pre-indexed variant

Applies when `P == 1 && W == 1`.

`LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!`

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

`LDC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>`

Decode for this encoding

```
1 if Rn == '1111' then SEE "LDC (literal)";
2 if coproc IN {'100x', '101x', '111x'}        then SEE "Floating-point and MVE";
3 if CoprocType(UInt(coproc)) == CP_CDEv1       then SEE "CDE instructions";
4 if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
5 if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
6 if !HaveMainExt()                             then UNDEFINED;
7 n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
8 index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

T2

Armv8-M Main Extension only

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	1 1 0	P	U	D	W	1	Rn != 1111	CRd	coproc	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

Decode for this encoding

```
1  if Rn == '1111'                              then SEE "LDC (literal)";
2  if coproc IN {'100x', '101x', '111x'}        then SEE "Floating-point and MVE";
3  if CoprocType(UInt(coproc)) == CP_CDEv1      then SEE "CDE instructions";
4  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
5  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
6  if !HaveMainExt()                            then UNDEFINED;
7  n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
8  index = (P == '1');  add = (U == '1');  wback = (W == '1');
```

Assembler symbols for all encodings

L	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<CRd>	Is the coprocessor register to be transferred, encoded in the "CRd" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see LDC, LDC2 (literal).
<option>	Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.

+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	  – when U = 0
	  + when U = 1

<imm>	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4
5      thisInstr = ThisInstr();
6      if !Coproc_Accepted(cp, thisInstr) then
```

```
 7              GenerateCoprocessorException();
 8          else
 9              offset_addr = if add then (R[n] + imm32) else (R[n] – imm32);
10              address = if index then offset_addr else R[n];
11
12              // Determine if the stack pointer limit check should be performed
13              if wback && n == 13 then
14                  violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
15              else
16                  violatesLimit = FALSE;
17
18              // Memory operation only performed if limit not violated
19              if !violatesLimit then
20                  repeat
21                      Coproc_SendLoadedWord(MemA[address,4], cp, thisInstr);
22                      address = address + 4;
23                  until Coproc_DoneLoading(cp, thisInstr);
24
25              // If the stack pointer is being updated a fault will be raised
26              // if the limit is violated
27              if wback then RSPCheck[n] = offset_addr;
```

C1.0.9 LDC, LDC2 (literal)

Load Coprocessor (literal). Load Coprocessor loads memory data from a sequence of consecutive memory addresses to a coprocessor. If no coprocessor can execute the instruction, a UsageFault exception is generated.

This is a generic coprocessor instruction. The D bit and the CRd field have no functionality defined by the architecture and are free for use by the coprocessor instruction set designer.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 0	P	U	D	W	1 1 1 1	CRd	coproc	imm8

T1 variant

Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}        then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1       then SEE "CDE instructions";
3  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
4  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt()                             then UNDEFINED;
6  index = (P == '1'); // Always TRUE in the T32 instruction set
7  add = (U == '1');  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
8  if W == '1' || P == '0'                       then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.

- The instruction executes without writeback of the base address.

- The instruction executes as LDC with writeback to the PC.

T2

Armv8-M Main Extension only

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	1 1 0	P	U	D	W	1 1 1 1	CRd	coproc	imm8

T2 variant

Applies when `!(P == 0 && U == 0 && W == 0)`.

```
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, <label>
LDC2{L}{<c>}{<q>} <coproc>, <CRd>, [PC, #{+/-}<imm>]
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}          then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1        then SEE "CDE instructions";
3  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MRRC, MRRC2";
4  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt()                              then UNDEFINED;
6  index = (P == '1'); // Always TRUE in the T32 instruction set
7  add = (U == '1');   cp = UInt(coproc);   imm32 = ZeroExtend(imm8:'00', 32);
8  if W == '1' || P == '0'                        then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `W == '1' || P == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as NOP.

- The instruction executes without writeback of the base address.

- The instruction executes as LDC with writeback to the PC.

Assembler symbols for all encodings

L	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c>`	See **Standard assembler syntax fields**.
`<q>`	See **Standard assembler syntax fields**.
`<coproc>`	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
`<CRd>`	Is the coprocessor register to be transferred, encoded in the "CRd" field.
`<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 Align(PC, 4) value of the instruction to this label. Permitted values of the offset are multiples of 4 in the range -1020 to 1020. If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1). If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).

+/−	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	  − when U = 0
	  + when U = 1

`<imm>`	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4
5      thisInstr = ThisInstr();
6      if !Coproc_Accepted(cp, thisInstr) then
7          GenerateCoprocessorException();
8      else
9          offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
10         address = if index then offset_addr else Align(PC,4);
11         repeat
12             Coproc_SendLoadedWord(MemA[address,4], cp, thisInstr);  address = address + 4;
13         until Coproc_DoneLoading(cp, thisInstr);
```

C1.0.10 MCR, MCR2

Move to Coprocessor from Register. Move to Coprocessor from Register passes the value of a general-purpose register to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

T1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  t = UInt(Rt);  cp = UInt(coproc);
5  if t == 15 || t == 13                    then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

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	1 1 1 0	opc1	0	CRn	Rt	coproc	opc2	1	CRm

T2 variant

```
MCR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  t = UInt(Rt);  cp = UInt(coproc);
5  if t == 15 || t == 13                    then UNPREDICTABLE;
```

Assembler symbols for all encodings

<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<CRn>	Is the first coprocessor register, encoded in the "CRn" field.
<CRm>	Is the second coprocessor register, encoded in the "CRm" field.
<opc2>	Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4      if !Coproc_Accepted(cp, ThisInstr()) then
5          GenerateCoprocessorException();
6      else
7          Coproc_SendOneWord(R[t], cp, ThisInstr());
```

C1.0.11 MCRR, MCRR2

Move to Coprocessor from two Registers. Move to Coprocessor from two Registers passes the values of two general-purpose registers to a coprocessor.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 0 0	0 1 0 0	Rt2	Rt	coproc	opc1	CRm

T1 variant

```
MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
5  if t == 15 || t2 == 15                    then UNPREDICTABLE;
6  if t == 13 || t2 == 13                    then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

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	1 1 0 0	0 1 0 0	Rt2	Rt	coproc	opc1	CRm

T2 variant

```
MCRR2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3  if !HaveMainExt()                        then UNDEFINED;
4  t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
5  if t == 15 || t2 == 15                    then UNPREDICTABLE;
6  if t == 13 || t2 == 13                    then UNPREDICTABLE;
```

Assembler symbols for all encodings

<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field.
<Rt>	Is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	Is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<CRm>	Is a coprocessor register, encoded in the "CRm" field.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4      if !Coproc_Accepted(cp, ThisInstr()) then
5          GenerateCoprocessorException();
6      else
7          Coproc_SendTwoWords(R[t2], R[t], cp, ThisInstr());
```

C1.0.12 MRC, MRC2

Move to Register from Coprocessor. Move to Register from Coprocessor causes a coprocessor to transfer a value to a general-purpose register or to the condition flags.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 1 0	opc1	1 CRn	Rt	coproc	opc2	1 CRm

T1 variant

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'}  then SEE "Floating-point and MVE";
2 if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3 if !HaveMainExt()                       then UNDEFINED;
4 t = UInt(Rt);  cp = UInt(coproc);
5 if t == 13                              then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

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	1 1 1 0	opc1	1 CRn	Rt	coproc	opc2	1 CRm

T2 variant

```
MRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'}  then SEE "Floating-point and MVE";
2 if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3 if !HaveMainExt()                       then UNDEFINED;
4 t = UInt(Rt);  cp = UInt(coproc);
5 if t == 13                              then UNPREDICTABLE;
```

Assembler symbols for all encodings

<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 7, encoded in the "opc1" field.
<Rt>	Is the general-purpose register to be transferred or APSR_nzcv (encoded as 0b1111), encoded in the "Rt" field. If APSR_nzcv is used, bits [31:28] of the transferred value are written to the APSR condition flags.
<CRn>	Is the first coprocessor register, encoded in the "CRn" field.
<CRm>	Is the second coprocessor register, encoded in the "CRm" field.
<opc2>	Is a coprocessor-specific opcode in the range 0 to 7, defaulting to 0 and encoded in the "opc2" field.

Operation for all encodings

```
 1   if ConditionPassed() then
 2       EncodingSpecificOperations();
 3       ExecuteCPCheck(cp);
 4       if !Coproc_Accepted(cp, ThisInstr()) then
 5           GenerateCoprocessorException();
 6       else
 7           value = Coproc_GetOneWord(cp, ThisInstr());
 8           if t != 15 then
 9               R[t] = value;
10           else
11               APSR.N = value[31];
12               APSR.Z = value[30];
13               APSR.C = value[29];
14               APSR.V = value[28];
15               // value[27:0] are not used.
```

C1.0.13 MRRC, MRRC2

Move to two Registers from Coprocessor. Move to two Registers from Coprocessor causes a coprocessor to transfer values to two general-purpose registers.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 0 0	0 1 0 1	Rt2	Rt	coproc	opc1	CRm

T1 variant

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2 if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3 if !HaveMainExt()                       then UNDEFINED;
4 t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
5 if t == 15 || t2 == 15 || t == t2       then UNPREDICTABLE;
6 if t == 13 || t2 == 13                  then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The value in the destination register is UNKNOWN.

T2

Armv8-M Main Extension only

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	1 1 0 0	0 1 0 1	Rt2	Rt	coproc	opc1	CRm

T2 variant

```
MRRC2{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
1 if coproc IN {'100x', '101x', '111x'}   then SEE "Floating-point and MVE";
2 if CoprocType(UInt(coproc)) == CP_CDEv1 then SEE "CDE instructions";
3 if !HaveMainExt()                       then UNDEFINED;
4 t = UInt(Rt);  t2 = UInt(Rt2);  cp = UInt(coproc);
5 if t == 15 || t2 == 15 || t == t2       then UNPREDICTABLE;
6 if t == 13 || t2 == 13                  then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `t == t2`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as `NOP`.

- The value in the destination register is UNKNOWN.

Assembler symbols for all encodings

<c>	See **Standard assembler syntax fields**.
<q>	See **Standard assembler syntax fields**.
<coproc>	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
<opc1>	Is a coprocessor-specific opcode in the range 0 to 15, encoded in the "opc1" field.
<Rt>	Is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	Is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<CRm>	Is a coprocessor register, encoded in the "CRm" field.

Operation for all encodings

```
1  if ConditionPassed() then
2      EncodingSpecificOperations();
3      ExecuteCPCheck(cp);
4      if !Coproc_Accepted(cp, ThisInstr()) then
5          GenerateCoprocessorException();
6      else
7          (R[t2], R[t]) = Coproc_GetTwoWords(cp, ThisInstr());
```

C1.0.14 STC, STC2

Store Coprocessor. Store Coprocessor stores data from a coprocessor to a sequence of consecutive memory addresses.

If no coprocessor can execute the instruction, a UsageFault exception is generated.

Arm reserves coprocessors CP8 to CP15, and this manual defines the valid instructions when coproc is in this range.

T1

Armv8-M Main Extension only

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 0	1 1 0	P	U	D	W 0	Rn	CRd	coproc	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1`.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1`.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0`.

```
STC{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}        then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1       then SEE "CDE instructions";
3  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
4  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt()                             then UNDEFINED;
6  n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if n == 15 then UNPREDICTABLE;
```

T2

Armv8-M Main Extension only

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	1 1 0	P	U	D	W 0	Rn	CRd	coproc	imm8

Offset variant

Applies when `P == 1 && W == 0`.

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>{, #{+/-}<imm>}]
```

Post-indexed variant

Applies when `P == 0 && W == 1.`

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], #{+/-}<imm>
```

Pre-indexed variant

Applies when `P == 1 && W == 1.`

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>, #{+/-}<imm>]!
```

Unindexed variant

Applies when `P == 0 && U == 1 && W == 0.`

```
STC2{L}{<c>}{<q>} <coproc>, <CRd>, [<Rn>], <option>
```

Decode for this encoding

```
1  if coproc IN {'100x', '101x', '111x'}          then SEE "Floating-point and MVE";
2  if CoprocType(UInt(coproc)) == CP_CDEv1        then SEE "CDE instructions";
3  if P == '0' && U == '0' && D == '1' && W == '0' then SEE "MCRR, MCRR2";
4  if P == '0' && U == '0' && D == '0' && W == '0' then UNDEFINED;
5  if !HaveMainExt()                              then UNDEFINED;
6  n = UInt(Rn);  cp = UInt(coproc);  imm32 = ZeroExtend(imm8:'00', 32);
7  index = (P == '1');  add = (U == '1');  wback = (W == '1');
8  if n == 15 then UNPREDICTABLE;
```

Assembler symbols for all encodings

`L`	If specified, selects the D == 1 form of the encoding. If omitted, selects the D == 0 form.
`<c>`	See **Standard assembler syntax fields**.
`<q>`	See **Standard assembler syntax fields**.
`<coproc>`	Is the name of the coprocessor, encoded in the "coproc" field. The generic coprocessor names are p0 to p15.
`<CRd>`	Is the coprocessor register to be transferred, encoded in the "CRd" field.
`<Rn>`	Is the general-purpose base register, encoded in the "Rn" field.
`<option>`	Is a coprocessor option, in the range 0 to 255 enclosed in { }, encoded in the "imm8" field.

`+/-`	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in the "U" field. It can have the following values:
	  − when `U = 0`
	  + when `U = 1`

`<imm>`	Is the immediate offset used for forming the address, a multiple of 4 in the range 0-1020, defaulting to 0 and encoded in the "imm8" field, as <imm>/4.

Operation for all encodings

```
1   if ConditionPassed() then
2       EncodingSpecificOperations();
3       ExecuteCPCheck(cp);
4       if !Coproc_Accepted(cp, ThisInstr()) then
5           GenerateCoprocessorException();
6       else
7           offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
8           address = if index then offset_addr else R[n];
9
10          // Determine if the stack pointer limit check should be performed
11          if wback && n == 13 then
```

```
12                    violatesLimit = ViolatesSPLim(LookUpSP(), offset_addr);
13          else
14                    violatesLimit = FALSE;
15
16          // Memory operation only performed if limit not violated
17          if !violatesLimit then
18              repeat
19                  MemA[address,4] = Coproc_GetWordToStore(cp, ThisInstr());
20                  address         = address + 4;
21              until Coproc_DoneStoring(cp, ThisInstr());
22
23          // If the stack pointer is being updated a fault will be raised
24          // if the limit is violated
25          if wback then RSPCheck[n] = offset_addr;
```

C1.0.15   VCX1 (vector)

Custom Extension Instruction Class 1 Vector. Custom extension register instruction class 1 vector computes a value based on an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point and SIMD register file, and require the current execution state to have access to these registers.

This instruction is subject to beat-wise execution.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE

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 A	1 1 0 op1	0	D 1 0	op2	Vd	0	coproc	op3 1	op4

Accumulator variant

Applies when `A == 1`

```
VCX1A<v> <coproc>, <Qd>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
VCX1<v> <coproc>, <Qd>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   CheckDecodeFaults(ExtType_Mve);
3   if Vd[0] == '1'                      then UNDEFINED;
4   cp  = UInt(coproc);
5   imm = op1:op2:op3:op4;
6   acc = (A == '1');
7   d   = UInt(D:Vd[3:1]);
8   ExecuteCPCheck(cp);
9   if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

A   Accumulate with existing register contents. This parameter must be one of the following values:

    " Encoded as A = 0

    A Encoded as A = 1

<v>   See **Standard assembler syntax fields**.

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Qd>`	Is the source and destination vector register `Q0` – `Q7`, encoded in the "D:Vd" fields as `<Qd>`*2.
`<imm>`	Is the immediate encoded in "op1:op2:op3:op4".

Operation for all encodings

```
1  EncodingSpecificOperations();
2  ExecuteFPCheck();
3  if (!Coproc_Accepted(cp, ThisInstr())) then GenerateCoprocessorException();
4
5  // Get current beat number and predication mask
6  (curBeat, elmtMask) = GetCurInstrBeat();
7
8  result = Zeros(32);
9  if acc then
10     // If the accumulator variant is used, then the 32-bit value from the vector
11     // source-destination register is used as an input to the custom operation
12     result = VCX_op1(ThisInstr(), Q[d, curBeat], 32, TRUE, curBeat, elmtMask);
13 else
14     result = VCX_op0(ThisInstr(), 32, TRUE, curBeat, elmtMask);
15
16 for e = 0 to 3
17     // If the vector lane is not predicated
18     if elmtMask[e] == '1' then
19         Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

C1.0.16 VCX1

Custom Extension Instruction Class 1. Custom extension register instruction class 1 computes a value based on an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point register file, and require the current execution state to have access to these registers.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE or Armv8-M Floating-point Extension.

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 A	1 1 0 sz	0 D 1 0	op1	Vd	0	coproc	op2	0	op3

Single-register accumulator variant

Applies when `A == 1 and sz == 0`

```
VCX1A <coproc>, <Sd>, #<imm>
```

Double-register accumulator variant

Applies when `A == 1 and sz == 1`

```
VCX1A <coproc>, <Dd>, #<imm>
```

Single-register non-accumulator variant

Applies when `A == 0 and sz == 0`

```
VCX1 <coproc>, <Sd>, #<imm>
```

Double-register non-accumulator variant

Applies when `A == 0 and sz == 1`

```
VCX1 <coproc>, <Dd>, #<imm>
```

Decode for this encoding

```
1  if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2  dp_operation = (sz == '1');
3  CheckDecodeFaults(ExtType_MveOrFp);
4  cp  = UInt(coproc);
5  imm = op1:op2:op3;
6  acc = (A == '1');
7  d   = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
8  ExecuteCPCheck(cp);
9  if InITBlock()        then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as if NOP.

Assembler symbols for all encodings

`<A>`	Accumulate with existing register contents. This parameter must be one of the following values:

      "   Encoded as `A = 0`
      A   Encoded as `A = 1`

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Dd>`	Is the 64-bit name of the floating-point source and destination register `D0 - D15` encoded in the "D:Vd" fields.
`<Sd>`	Is the 32-bit name of the floating-point source and destination register `S0 - S31` encoded in the "Vd:D" fields.
`<imm>`	Is the immediate encoded in "op1:op2:op3".

Operation for all encodings

```
1   if ConditionPassed() then
2       EncodingSpecificOperations();
3       ExecuteFPCheck();
4       if (!Coproc_Accepted(cp, ThisInstr())) then
5           GenerateCoprocessorException();
6       elsif dp_operation then
7           if acc then
8               D[d] = VCX_op1(ThisInstr(), D[d], 64);
9           else
10              D[d] = VCX_op0(ThisInstr(), 64);
11      else
12          if acc then
13              S[d] = VCX_op1(ThisInstr(), S[d], 32);
14          else
15              S[d] = VCX_op0(ThisInstr(), 32);
```

C1.0.17   VCX2 (vector)

Custom Extension Instruction Class 2 Vector. Custom extension register instruction class 2 vector computes a value based on a source register, an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point and SIMD register file, and require the current execution state to have access to these registers.

This instruction is subject to beat-wise execution.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE

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 A	1 1 0 op1	0	D 1 1	op2	Vd	0	coproc	op3 1	M op4	Vm

Accumulator variant

Applies when `A == 1`

```
VCX2A<v> <coproc>, <Qd>, <Qm>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
VCX2<v> <coproc>, <Qd>, <Qm>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   CheckDecodeFaults(ExtType_Mve);
3   if Vd[0] == '1' || Vm[0] == '1'          then UNDEFINED;
4   cp  = UInt(coproc);
5   imm = op1:op2:op3:op4;
6   acc = (A == '1');
7   d   = UInt(D:Vd[3:1]);
8   m   = UInt(M:Vm[3:1]);
9   ExecuteCPCheck(cp);
10  if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP.

Assembler symbols for all encodings

A   Accumulate with existing register contents. This parameter must be one of the following
                    values:
                    "   Encoded as A = 0
                    A   Encoded as A = 1
<v>   See **Standard assembler syntax fields**.

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Qd>`	Is the source and destination vector register `Q0` – `Q7`, encoded in the "D:Vd" fields as `<Qd>*2`.
`<Qm>`	Is the source vector register `Q0` – `Q7`, encoded in the "M:Vm" fields as `<Qm>*2`.
`<imm>`	Is the immediate encoded in "op1:op2:op3:op4".

Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3   if (!Coproc_Accepted(cp, ThisInstr())) then GenerateCoprocessorException();
4
5   // Get current beat number and predication mask
6   (curBeat, elmtMask) = GetCurInstrBeat();
7
8   result = Zeros(32);
9   if acc then
10      // If the accumulator variant is used, then the 32-bit value from the vector
11      // source-destination register is used as an input to the custom operation
12      result = VCX_op2(ThisInstr(), Q[d, curBeat], Q[m, curBeat], 32, TRUE,
13                       curBeat, elmtMask);
14  else
15      result = VCX_op1(ThisInstr(), Q[m, curBeat], 32, TRUE, curBeat, elmtMask);
16
17  for e = 0 to 3
18      // If the vector lane is not predicated
19      if elmtMask[e] == '1' then
20          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

C1.0.18 VCX2

Custom Extension Instruction Class 2. Custom extension register instruction class 2 computes a value based on a source register, an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point register file, and require the current execution state to have access to these registers.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE or Armv8-M Floating-point Extension.

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 A	1 1 0 sz	0	D 1 1	op1	Vd	0	coproc	op2 0 M	op3	Vm

Single-register accumulator variant

Applies when `A == 1` and `sz == 0`

```
VCX2A <coproc>, <Sd>, <Sm>, #<imm>
```

Double-register accumulator variant

Applies when `A == 1` and `sz == 1`

```
VCX2A <coproc>, <Dd>, <Dm>, #<imm>
```

Single-register non-accumulator variant

Applies when `A == 0` and `sz == 0`

```
VCX2 <coproc>, <Sd>, <Sm>, #<imm>
```

Double-register non-accumulator variant

Applies when `A == 0` and `sz == 1`

```
VCX2 <coproc>, <Dd>, <Dm>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   dp_operation = (sz == '1');
3   CheckDecodeFaults(ExtType_MveOrFp);
4   cp   = UInt(coproc);
5   imm = op1:op2:op3;
6   acc = (A == '1');
7   d    = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
8   m    = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
9   ExecuteCPCheck(cp);
10  if InITBlock()        then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

`<A>`	Accumulate with existing register contents. This parameter must be one of the following values:

  "   Encoded as `A = 0`
  A   Encoded as `A = 1`

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Dd>`	Is the 64-bit name of the floating-point source and destination register `D0 - D15` encoded in the "D:Vd" fields.
`<Dm>`	Is the 64-bit name of the floating-point source register `D0 - D15`, encoded in the "M:Vm" fields.
`<Sd>`	Is the 32-bit name of the floating-point source and destination register `S0 - S31` encoded in the "Vd:D" fields.
`<Sm>`	Is the 32-bit name of the floating-point source register `S0 - S31`, encoded in the "Vm:M" fields
`<imm>`	Is the immediate encoded in "op1:op2:op3".

Operation for all encodings

```
1   if ConditionPassed() then
2       EncodingSpecificOperations();
3       ExecuteFPCheck();
4       if (!Coproc_Accepted(cp, ThisInstr())) then
5           GenerateCoprocessorException();
6       elsif dp_operation then
7           if acc then
8               D[d] = VCX_op2(ThisInstr(), D[d], D[m], 64);
9           else
10              D[d] = VCX_op1(ThisInstr(), D[m], 64);
11      else
12          if acc then
13              S[d] = VCX_op2(ThisInstr(), S[d], S[m], 32);
14          else
15              S[d] = VCX_op1(ThisInstr(), S[m], 32);
```

C1.0.19 VCX3 (vector)

Custom Extension Instruction Class 3 Vector. Custom extension register instruction class 3 vector computes a value based on two source registers, an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point and SIMD register file, and require the current execution state to have access to these registers.

This instruction is subject to beat-wise execution.

This instruction is VPT compatible.

This instruction is not permitted in an IT block.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE

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 A	1 1 0 op1	1 D op2	Vn	Vd	0 coproc	N 1 M op3		Vm

Accumulator variant

Applies when `A == 1`

```
VCX3A<v> <coproc>, <Qd>, <Qn>, <Qm>, #<imm>
```

Non-accumulator variant

Applies when `A == 0`

```
VCX3<v> <coproc>, <Qd>, <Qn>, <Qm>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1        then SEE "Coprocessor instructions";
2   CheckDecodeFaults(ExtType_Mve);
3   if Vd[0] == '1' || Vn[0] == '1' || Vm[0] == '1' then UNDEFINED;
4   cp  = UInt(coproc);
5   imm = op1:op2:op3;
6   acc = (A == '1');
7   d   = UInt(D:Vd[3:1]);
8   n   = UInt(N:Vn[3:1]);
9   m   = UInt(M:Vm[3:1]);
10  ExecuteCPCheck(cp);
11  if InITBlock() then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

A   Accumulate with existing register contents. This parameter must be one of the following
                  values:
     "   Encoded as A = 0
     A   Encoded as A = 1
\<v\>   See **Standard assembler syntax fields**.

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Qm>`	Is the source vector register `Q0` – `Q7`, encoded in the "M:Vm" fields as `<Qm>`*2.
`<Qd>`	Is the source and destination vector register `Q0` – `Q7`, encoded in the "D:Vd" fields as `<Qd>`*2.
`<Qn>`	Is the source vector register `Q0` – `Q7`, encoded in the "N:Vn" fields as `<Qn>`*2.
`<imm>`	Is the immediate encoded in "op1:op2:op3".

Operation for all encodings

```
1   EncodingSpecificOperations();
2   ExecuteFPCheck();
3   if (!Coproc_Accepted(cp, ThisInstr())) then GenerateCoprocessorException();
4
5   // Get current beat number and predication mask
6   (curBeat, elmtMask) = GetCurInstrBeat();
7
8   result = Zeros(32);
9   if acc then
10      // If the accumulator variant is used, then the 32-bit value from the vector
11      // source-destination register is used as an input to the custom operation
12      result = VCX_op3(ThisInstr(), Q[d, curBeat], Q[n, curBeat], Q[m, curBeat], 32,
13                  TRUE, curBeat, elmtMask);
14  else
15      result = VCX_op2(ThisInstr(), Q[n, curBeat], Q[m, curBeat], 32,
16                  TRUE, curBeat, elmtMask);
17
18  for e = 0 to 3
19      // If the vector lane is not predicated
20      if elmtMask[e] == '1' then
21          Elem[Q[d, curBeat], e, 8] = Elem[result, e, 8];
```

C1.0.20 VCX3

Custom Extension Instruction Class 3. Custom extension register instruction class 3 computes a value based on two source registers, an immediate, and optionally the destination value, and writes the result to the destination register. The source and destination registers are within the Floating-point register file, and require the current execution state to have access to these registers.

T1

Armv8-M Custom Datapath Extension with Armv8.1-M MVE or Armv8-M Floating-point Extension.

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 A	1 1 0 sz	1 D op1	Vn	Vd	0 coproc	N 0 M op2	Vm

Single-register accumulator variant

Applies when `A == 1` and `sz == 0`

```
VCX3A <coproc>, <Sd>, <Sn>, <Sm>, #<imm>
```

Double-register accumulator variant

Applies when `A == 1` and `sz == 1`

```
VCX3A <coproc>, <Dd>, <Dn>, <Dm>, #<imm>
```

Single-register non-accumulator variant

Applies when `A == 0` and `sz == 0`

```
VCX3 <coproc>, <Sd>, <Sn>, <Sm>, #<imm>
```

Double-register non-accumulator variant

Applies when `A == 0` and `sz == 1`

```
VCX3 <coproc>, <Dd>, <Dn>, <Dm>, #<imm>
```

Decode for this encoding

```
1   if CoprocType(UInt(coproc)) != CP_CDEv1 then SEE "Coprocessor instructions";
2   dp_operation = (sz == '1');
3   CheckDecodeFaults(ExtType_MveOrFp);
4   cp  = UInt(coproc);
5   imm = op1:op2;
6   acc = (A == '1');
7   d   = if dp_operation then UInt(D:Vd) else UInt(Vd:D);
8   n   = if dp_operation then UInt(N:Vn) else UInt(Vn:N);
9   m   = if dp_operation then UInt(M:Vm) else UInt(Vm:M);
10  ExecuteCPCheck(cp);
11  if InITBlock()        then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.

- The instruction executes as if it passes the Condition code check.

- The instruction executes as NOP.

Assembler symbols for all encodings

`<A>`	Accumulate with existing register contents. This parameter must be one of the following values:

    "   Encoded as `A = 0`

    `A`   Encoded as `A = 1`

`<coproc>`	Is the name of the coprocessor, encoding in the "coproc" field. Valid names are `p0` to `p7` inclusive.
`<Dd>`	Is the 64-bit name of the floating-point source and destination register `D0 - D15` encoded in the "D:Vd" fields.
`<Dm>`	Is the 64-bit name of the floating-point source register `D0 - D15`, encoded in the "M:Vm" fields.
`<Dn>`	Is the 64-bit name of the floating-point source register `D0 - D15`, encoded in the "N:Vn" fields
`<Sd>`	Is the 32-bit name of the floating-point source and destination register `S0 - S31` encoded in the "Vd:D" fields.
`<Sm>`	Is the 32-bit name of the floating-point source register `S0 - S31`, encoded in the "Vm:M" fields
`<Sn>`	Is the 32-bit name of the floating-point source register `S0 - S31`, encoded in the "Vn:N" fields.
`<imm>`	Is the immediate encoded in "op1:op2".

Operation for all encodings

```
1   if ConditionPassed() then
2       EncodingSpecificOperations();
3       ExecuteFPCheck();
4       if (!Coproc_Accepted(cp, ThisInstr())) then
5           GenerateCoprocessorException();
6       elsif dp_operation then
7           if acc then
8               D[d] = VCX_op3(ThisInstr(), D[d], D[n], D[m], 64);
9           else
10              D[d] = VCX_op2(ThisInstr(), D[n], D[m], 64);
11      else
12          if acc then
13              S[d] = VCX_op3(ThisInstr(), S[d], S[n], S[m], 32);
14          else
15              S[d] = VCX_op2(ThisInstr(), S[n], S[m], 32);
```

Part D
Pseudocode Specification

Chapter D1
Pseudocode Specification

This chapter specifies the Armv8-M pseudocode. It contains the following section:

Alphabetical Pseudocode List

D1.1 Alphabetical Pseudocode List

D1.1.1 CdeImpDefValue

```
1  // CdeImpDefValue()
2  // ================
3  // IMPLEMENTATION DEFINED value functions for the Custom Datapath Extension
4
5  bits(size) CdeImpDefValue(bits(N) instr);
6  bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa);
7  bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa, bits(L) opb);
8  bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa, bits(L) opb, bits(K) opc);
9
10 bits(size) CdeImpDefValue(bits(N) instr, integer curBeat, bits(4) elmtMask);
11 bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa, integer curBeat, bits(4) elmtMask);
12 bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa, bits(L) opb, integer curBeat,
13                           bits(4) elmtMask);
14 bits(size) CdeImpDefValue(bits(N) instr, bits(M) opa, bits(L) opb, bits(K) opc,
15                           integer curBeat, bits(4) elmtMask);
```

D1.1.2 CoprocType

```
1  // CoprocType
2  // ==========
3
4  CPDef CoprocType(integer coproc)
5      assert coproc >= 0 && coproc <= 7;
6
7      // Returns the architecture defined enumeration of the instruction set
8      // supported by the given coprocessor space.
9
10     // The CDE extension defines two encoding patterns:
11     // - CP_GCP   : The architected coprocessor encodings for MRC, MCR, CDP etc.
12     // - CP_CDEv1 : Version 1 of the Custom Datapath Extension.
13
14     cdeEn = boolean IMPLEMENTATION_DEFINED "CDE enabled coprocessor";
15     return if cdeEn then CP_GCP else CP_CDEv1;
```

D1.1.3 CPDef

```
1  // CPDef
2  // =====
3  // The CDE extension defines two encoding patterns
4
5  enumeration CPDef { CP_GCP,  // The architected coprocessor encodings for MRC, MCR, CDP etc.
6                      CP_CDEv1 // Version 1 of the Custom Datapath Extension.
7                    };
```

D1.1.4 CX_op0

```
1  // CX_op0
2  // ======
3
4  bits(size) CX_op0(bits(N) instr, integer size)
5      assert size == N;
6      assert N IN {32, 64};
7
8      // Custom data path returning IMPLEMENTATION DEFINED value based on
9      // instruction opcode only.
10     return CdeImpDefValue(instr);
```

D1.1.5 CX_op1

```
1   // CX_op1
2   // ======
3
4   bits(size) CX_op1(bits(32) instr, bits(N) opa, integer size)
5       assert size == N;
6       assert N IN {32, 64};
7
8       // Custom data path returning IMPLEMENTATION DEFINED value based on
9       // instruction opcode and single 32-bit or 64-bit operand, opa, only.
10      return CdeImpDefValue(instr, opa);
```

D1.1.6 CX_op2

```
1   // CX_op2
2   // ======
3
4   bits(size) CX_op2(bits(32) instr, bits(N) opa, bits(32) opb, integer size)
5       assert size == N;
6       assert N IN {32, 64};
7
8       // Custom data path returning IMPLEMENTATION DEFINED value based on instruction
9       // opcode and two 32-bit or 64-bit operands, opa and opb, only.
10      return CdeImpDefValue(instr, opa, opb);
```

D1.1.7 CX_op3

```
1   // CX_op3
2   // ======
3
4   bits(size) CX_op3(bits(32) instr, bits(N) opa, bits(32) opb, bits(32) opc, integer size)
5       assert size == N;
6       assert N IN {32, 64};
7
8       // Custom data path returning IMPLEMENTATION DEFINED value based on instruction
9       // opcode and three 32-bit or 64-bit operands, opa, opb and opc, only.
10      return CdeImpDefValue(instr, opa, opb, opc);
```

D1.1.8 RF

```
1   // RF[] - non-assignment form
2   // ==========================
3
4   bits(32) RF[integer n]
5       assert n >= 0 && n <= 15;
6
7       // Returns the selected general-purpose register for indices less than 15,
8       // or the APSR Condition flags for the index 15.
9
10      if n < 15 then
11          result = R[n];
12      else
13          result = APSR[31:28] : Zeros(28);
14
15      return result;
16
17  // RF[] - assignment form
18  // ======================
19
20  RF[integer n] = bits(32) value
21      assert n >= 0 && n <= 15;
22
23      // Assigns a value to the selected general-purpose register for indices
24      // less than 15, or the APSR Condition flags for the index 15.
25
26      if n < 15 then
```

```
27          R[n] = value;
28      else
29          APSR.N = value[31];
30          APSR.Z = value[30];
31          APSR.C = value[29];
32          APSR.V = value[28];
```

D1.1.9 RFD

```
1   // RFD[] - non-assignment form
2   // ==========================
3
4   bits(64) RFD[integer n]
5       assert n >= 0 && n <= 14;
6       assert n[0] == '0';
7
8       // Returns the selected general-purpose register pair
9       // Register pairs containing SP or PC are UNPREDICTABLE
10      if n > 10 then UNPREDICTABLE;
11
12      result = R[n+1]:R[n];
13      return result;
14
15  // RFD[] - assignment form
16  // ======================
17
18  RFD[integer n] = bits(64) value
19      assert n >= 0 && n <= 14;
20      assert n[0] == '0';
21
22      // Assigns a value to the selected general-purpose register pair
23      // Register pairs containing SP or PC are UNPREDICTABLE
24      if n > 10 then UNPREDICTABLE;
25
26      R[n+1] = value[63:32];
27      R[n]   = value[31:0];
```

D1.1.10 VCX_op0

```
1   // VCX_op0
2   // =======
3
4   bits(size) VCX_op0(bits(N) instr, integer size)
5       return VCX_op0(instr, size, FALSE, integer UNKNOWN, bits(4) UNKNOWN);
6
7   bits(size) VCX_op0(bits(N) instr, integer size, boolean isBeatWise, integer curBeat,
8                      bits(4) elmtMask)
9       assert size == N;
10      assert N == 32;
11
12      // Custom data path returning IMPLEMENTATION DEFINED value based on
13      // instruction opcode only.
14      if isBeatWise then
15          return CdeImpDefValue(instr, curBeat, elmtMask);
16      else
17          return CdeImpDefValue(instr);
```

D1.1.11 VCX_op1

```
1   // VCX_op1
2   // =======
3
4   bits(size) VCX_op1(bits(32) instr, bits(N) opa, integer size)
5       return VCX_op1(instr, opa, size, FALSE, integer UNKNOWN, bits(4) UNKNOWN);
6
```

```
7   bits(size) VCX_op1(bits(32) instr, bits(N) opa, integer size, boolean isBeatWise,
8                        integer curBeat, bits(4) elmtMask)
9       assert size == N;
10      assert N IN {32, 64, 128};
11
12      // Custom data path returning IMPLEMENTATION DEFINED value based on instruction
13      // opcode and single 32-bit, 64-bit, or 128-bit operand, opa, only.
14      if isBeatWise then
15          return CdeImpDefValue(instr, opa, curBeat, elmtMask);
16      else
17          return CdeImpDefValue(instr, opa);
```

D1.1.12 VCX_op2

```
1   // VCX_op2
2   // ======
3
4   bits(size) VCX_op2(bits(32) instr, bits(N) opa, bits(N) opb, integer size)
5       return VCX_op2(instr, opa, opb, size, FALSE, integer UNKNOWN, bits(4) UNKNOWN);
6
7   bits(size) VCX_op2(bits(32) instr, bits(N) opa, bits(N) opb, integer size,
8                        boolean isBeatWise, integer curBeat, bits(4) elmtMask)
9       assert size == N;
10      assert N IN {32, 64, 128};
11
12      // Custom data path returning IMPLEMENTATION DEFINED value based on instruction
13      // opcode and two 32-bit or 64-bit operands, opa and opb, only.
14      if isBeatWise then
15          return CdeImpDefValue(instr, opa, opb, curBeat, elmtMask);
16      else
17          return CdeImpDefValue(instr, opa, opb);
```

D1.1.13 VCX_op3

```
1   // VCX_op3
2   // ======
3
4   bits(size) VCX_op3(bits(32) instr, bits(N) opa, bits(N) opb, bits(N) opc, integer size)
5       return VCX_op3(instr, opa, opb, opc, size, FALSE, integer UNKNOWN, bits(4) UNKNOWN);
6
7   bits(size) VCX_op3(bits(32) instr, bits(N) opa, bits(N) opb, bits(N) opc, integer size,
8                        boolean isBeatWise, integer curBeat, bits(4) elmtMask)
9       assert size == N;
10      assert N IN {32, 64, 128};
11
12      // Custom data path returning IMPLEMENTATION DEFINED value based on instruction
13      // opcode and three 32-bit, 64-bit, or 128-bit operands, opa, opb and opc, only.
14      if isBeatWise then
15          return CdeImpDefValue(instr, opa, opb, opc, curBeat, elmtMask);
16      else
17          return CdeImpDefValue(instr, opa, opb, opc);
```

Glossary

Application Program Status Register (APSR)

> The register containing those bits that deliver status information about the results of instructions, the N, Z, C, and V bits of the XPSR. In an implementation that includes the DSP extension, the APSR includes the GE bits that provide status information from DSP operations.

APSR

> *See* Application Program Status Register.

Architecturally executed

> An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired it has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a NOP because it fails its condition code check, is an architecturally executed instruction.

> In a PE that performs Speculative execution, an instruction is not architecturally executed if the PE discards the results of a Speculative execution.

> *See also* Condition code check, Simple sequential execution.

Architecturally Unknown

> An architecturally UNKNOWN value is a value that is not defined by the architecture but must meet the requirements of the definition of UNKNOWN. Implementations can define the value of the field, but are not required to do so.

> *See also* Implementation Defined.

Behaves as if

> Where this manual indicates that a PE *behaves as if* a certain condition applies, all descriptions of the operation of the PE must be re-evaluated taking account of that condition, together with any other conditions that affect operation.

Byte

> An 8-bit data item.

Callee-saved registers

> Are registers that a called procedure must preserve. To preserve a callee-saved register, the called procedure would normally either not use the register at all, or store the register to the stack during procedure entry and reload it from the stack during procedure exit.

Caller-saved registers

> Are registers that a called procedure is not required to preserve. If the calling procedure requires their values to be preserved, it must store and reload them itself.

Condition code check

> The process of determining whether a conditional instruction executes normally or is treated as a NOP. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally. For a T32 instruction in an IT block, the value of EPSR.IT determines whether the instruction is executed normally.

> *See also* Condition code field, Condition flags, Conditional execution.

Condition code field

A 4-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags

The N, Z, C, and V bits of APSR, or XPSR.

See also Condition code check.

Conditional execution

When a conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a NOP.

See also Condition code check.

CONSTRAINED UNPREDICTABLE

Where an instruction can result in UNPREDICTABLE behavior, the Armv8 architecture specifies a narrow range of permitted behaviors. This range is the range of CONSTRAINED UNPREDICTABLE behavior. All implementations that are compliant with the architecture must follow the CONSTRAINED UNPREDICTABLE behavior within the limits defined for each particular case, and this behavior might vary.

In body text, the term CONSTRAINED UNPREDICTABLE is shown in SMALLCAPS.

See also Unpredictable.

Doubleword

A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Exception

Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

Halfword

A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

If-Then block (IT block)

An IT block is a block of up to four instructions following an *If-Then* (IT) instruction. Each instruction in the block is conditional. The conditions for the instructions are either all the same, or some are the inverse of others.

Immediate and offset fields

Are unsigned unless otherwise stated.

Immediate value

A value that is encoded directly in the instruction and used as numeric data when the instruction is executed. Many T32 instructions can be used with an immediate argument.

IMP DEF

An abbreviation that is used in diagrams to indicate that one or more bits have IMPLEMENTATION DEFINED behavior.

IMPLEMENTATION DEFINED

Means that the behavior is not architecturally defined, but must be defined and documented by individual implementations.

In body text, the term IMPLEMENTATION DEFINED is shown in SMALLCAPS.

OPTIONAL

When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture. Arm expects such a feature to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALLCAPS.

Note: Do not confuse these Arm-specific uses of OPTIONAL with other uses of OPTIONAL, where it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior that is determined by an implementation choice.

PE

See Processing element.

Processing element (PE)

The abstract machine that is defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Quadword

A 128-bit data item. Quadwords are normally at least word-aligned in Arm systems.

Reserved

Unless otherwise stated:

- Instructions that are reserved or that access reserved registers have UNPREDICTABLE or CONSTRAINED UNPREDICTABLE behavior.

- Bit positions that are described as reserved are:

  - In an RW or WO register, RES0.
  - In an RO register, UNK.

See also CONSTRAINED UNPREDICTABLE, RES0, RES1, UNDEFINED, UNK, UNPREDICTABLE.

SIMD

Single-Instruction, Multiple-Data.

T32 instruction

One or two halfwords that specify an operation to be performed by a PE. T32 instructions must be halfword-aligned.

T32 instructions were previously called Thumb instructions.

UAL

See Unified Assembler Language.

Unallocated

Except where otherwise stated in this manual, an instruction encoding is unallocated if the architecture does not assign a specific function to the entire bit pattern of the instruction, but instead describes it as CONSTRAINED UNPREDICTABLE, UNDEFINED, UNPREDICTABLE, or as an unallocated hint instruction.

A bit in a register is unallocated if the architecture does not assign a function to that bit.

See also CONSTRAINED UNPREDICTABLE, UNPREDICTABLE, UNDEFINED.

UNDEFINED

Indicates an instruction that generates an Undefined Instruction exception.

In body text, the term UNDEFINED is shown in SMALLCAPS.

Unified Assembler Language

The assembler language that is introduced with Thumb-2 technology that is used in this manual.

UNK

An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

Hardware must implement the bit as read as 0, or all 0s for a multi-bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not return information that cannot be accessed at the current or a lower level of privilege using functionality that is not UNKNOWN, is not CONSTRAINED UNPREDICTABLE, and does not return UNKNOWN values.

An Unknown value must not be documented or promoted as having a defined value or effect.

In body text, the term UNKNOWN is shown in SMALLCAPS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED, UNK, UNPREDICTABLE.

UNPREDICTABLE

Means the behavior cannot be relied on. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege or security using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In body text, the term UNPREDICTABLE is shown in SMALLCAPS.

See also CONSTRAINED UNPREDICTABLE, UNDEFINED.

Word

A 32-bit data item. Words are normally word-aligned in Arm systems.