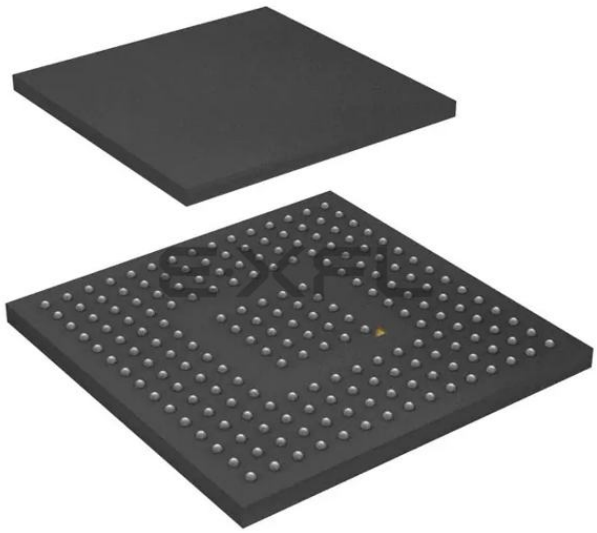

Welcome to E-XFL.COM

What is "Embedded - Microcontrollers"?

"Embedded - Microcontrollers" refer to small, integrated circuits designed to perform specific tasks within larger systems. These microcontrollers are essentially compact computers on a single chip, containing a processor core, memory, and programmable input/output peripherals. They are called "embedded" because they are embedded within electronic devices to control various functions, rather than serving as standalone computers. Microcontrollers are crucial in modern electronics, providing the intelligence and control needed for a wide range of applications.

Applications of "Embedded - Microcontrollers"

Details	
Product Status	Active
Core Processor	ARM® Cortex®-M7
Core Size	32-Bit Single-Core
Speed	216MHz
Connectivity	CANbus, EBI/EMI, I²C, IrDA, LINbus, MMC/SD, QSPI, SAI, SPI, UART/USART, USB
Peripherals	Brown-out Detect/Reset, DMA, I²S, POR, PWM, WDT
Number of I/O	138
Program Memory Size	512KB (512K x 8)
Program Memory Type	FLASH
EEPROM Size	-
RAM Size	256K x 8
Voltage - Supply (Vcc/Vdd)	1.7V ~ 3.6V
Data Converters	A/D 24x12b; D/A 2x12b
Oscillator Type	Internal
Operating Temperature	-40°C ~ 85°C (TA)
Mounting Type	Surface Mount
Package / Case	176-UFBGA
Supplier Device Package	176-UFBGA (10x10)
Purchase URL	https://www.e-xfl.com/product-detail/stmicroelectronics/stm32f723iek6

2 The Cortex-M7 processor

2.1 Programmers model

This section describes the Cortex®-M7 programmers model. In addition to the individual core register descriptions, it contains information about the processor modes and privilege levels for software execution and stacks.

2.1.1 Processor mode and privilege levels for software execution

The processor *modes* are:

Thread mode Executes application software. The processor enters Thread mode when it comes out of reset.

Handler mode Handles exceptions. The processor returns to Thread mode when it has finished all exception processing.

The *privilege levels* for software execution are:

Unprivileged The software:

- Has limited access to system registers using the MSR and MRS instructions, and cannot use the CPS instruction to mask interrupts.
- Cannot access the system timer, NVIC, or system control block.
- Might have restricted access to memory or peripherals.

Unprivileged software executes at the unprivileged level.

Privileged The software can use all the instructions and has access to all resources.

Privileged software executes at the privileged level.

In Thread mode, the CONTROL register controls whether software execution is privileged or unprivileged, see *CONTROL register on page 27*. In Handler mode, software execution is always privileged.

Only privileged software can write to the CONTROL register to change the privilege level for software execution in Thread mode. Unprivileged software can use the SVC instruction to make a *supervisor call* to transfer control to privileged software.

2.1.2 Stacks

The processor uses a full descending stack. This means the stack pointer holds the address of the last stacked item in memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location. The processor implements two stacks, the *main stack* and the *process stack*, with a pointer for each held in independent registers, see *Stack Pointer on page 21*.

In Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack, see *CONTROL register on page 27*. In Handler mode, the processor always uses the main stack. The options for processor operations are:

Table 5. IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode. 1 = Reserved. 2 = NMI. 3 = HardFault. 4 = MemManage. 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall. 12 = Reserved for debug 13 = Reserved 14 = PendSV. 15 = SysTick. 16 = IRQ0. . . 256 = IRQ239. see *Exception types on page 39* for more information.

Execution Program Status register

The EPSR contains the Thumb state bit, and the execution state bits for either the:

- *If-Then* (IT) instruction.
- *Interruptible-Continuable Instruction* (ICI) field for an interrupted load multiple or store multiple instruction.

See the register summary in *Table 6 on page 24* for the EPSR attributes. The bit assignments are

Table 6. EPSR bit assignments

Bits	Name	Function
[31:27]	-	Reserved.
[26:25], [15:10]	ICI	Interruptible-continuable instruction bits, see *Interruptible-continuable instructions on page 25*.
[26:25], [15:10]	IT	Indicates the execution state bits of the IT instruction, see *IT on page 148*.
[24]	T	Thumb state bit, see *Thumb state*.
[23:16]	-	Reserved.
[9:0]	-	Reserved.

Table 22. Cortex®-M7 instructions (continued)

Mnemonic	Operands	Brief description	Flags	Page
VCMPE.F<32|64>	<Sd|Dd>, <<Sm|Dm> | #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR	*3.11.3 on page 154*
VCVT	F<32|64>.<S|U>, <16|32>	Convert from floating-point to fixed point	-	*3.11.5 on page 156*
VCVT	<S|U>, <16|32>.F<32|64>	Convert from fixed point to floating-point		*3.11.5 on page 156*
VCVT.S32.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert from floating-point to integer	-	*3.11.4 on page 155*
VCVT<B|T>.F<32|64>.F16	<Sd|Dd>, Sm	Convert half-precision value to single-precision or double-precision	-	*3.11.6 on page 157*
VCVTA.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding to nearest ties away	-	*3.11.33 on page 173*
VCVTM.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding towards minus infinity	-	*3.11.33 on page 173*
VCVTN.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding to nearest even	-	*3.11.33 on page 173*
VCVTP.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert from floating-point to integer with directed rounding towards plus infinity	-	*3.11.33 on page 173*
VCVTR.S32.F<32|64>	<Sd|Dd>, <Sm|Dm>	Convert between floating-point and integer with rounding.	FPSCR	*3.11.4 on page 155*
VCVT<B|T>.F16.F<32|64>	Sd, <Sm|Dm>	Convert single-precision or double precision register to half-precision	-	*3.11.5 on page 156*
VDIV.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Divide	-	*3.11.7 on page 157*
VFMA.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Multiply Accumulate	-	*3.11.8 on page 158*
VFMS.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Multiply Subtract	-	*3.11.8 on page 158*
VFNMA.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Negate Multiply Accumulate	-	*3.11.9 on page 159*
VFNMS.F<32|64>	{<Sd|Dd>,} <Sn|Dn>, <Sm|Dm>	Floating-point Fused Negate Multiply Subtract	-	*3.11.9 on page 159*

Condition code suffixes

The instructions that can be conditional have an optional condition code, shown in syntax descriptions as {*cond*}. Conditional execution requires a preceding IT instruction. An instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition. *Table 25* shows the condition codes to use.

The conditional execution can be used with the IT instruction to reduce the number of branch instructions in the code.

Table 25 also shows the relationship between condition code suffixes and the N, Z, C, and V flags.

Table 25. Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Example 3-1: absolute value shows the use of a conditional instruction to find the absolute value of a number. R0 = abs(R1).

Example 3-1: absolute value

```
MOVS    R0, R1          ; R0 = R1, setting flags.
IT      MI              ; Skipping next instruction if value 0 or
                        ; positive.
RSBMI   R0, R0, #0      ; If negative, R0 = -R0.
```

Where:

op Is one of:
- LDR Load register.
- STR Store register.

type Is one of:
- B Unsigned byte, zero extend to 32 bits on loads.
- SB Signed byte, sign extend to 32 bits (LDR only).
- H Unsigned halfword, zero extend to 32 bits on loads.
- SH Signed halfword, sign extend to 32 bits (LDR only).
- – Omit, for word.

cond Is an optional condition code. See *Conditional execution on page 68*.

Rt Is the register to load or store.

Rn Is the register on which the memory address is based.

offset Is an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*.

Rt2 Is the additional register to load or store for two-word operations.

Operation

LDR instructions load one or two registers with a value from memory.

STR instructions store one or two register values to memory.

Load and store instructions with immediate offset can use the following addressing modes:

Offset addressing

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access. The register *Rn* is unaltered. The assembly language syntax for this mode is:

```
[Rn, #offset]
```

Pre-indexed addressing

The offset value is added to or subtracted from the address obtained from the register *Rn*. The result is used as the address for the memory access and written back into the register *Rn*. The assembly language syntax for this mode is:

```
[Rn, #offset]!
```

Post-indexed addressing

The address obtained from the register *Rn* is used as the address for the memory access. The offset value is added to or subtracted from the address, and written back into the register *Rn*. The assembly language syntax for this mode is:

```
[Rn], #offset
```

The value to load or store can be a byte, halfword, word, or two words. Bytes and halfwords can either be signed or unsigned. See *Address alignment on page 68*.

Table 27 shows the ranges of offset for immediate, pre-indexed and post-indexed forms

Table 27. Offset ranges

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	-255 to 4095	-255 to 255	-255 to 255
Two words	multiple of 4 in the range -1020 to 1020	multiple of 4 in the range -1020 to 1020	multiple of 4 in the range -1020 to 1020

Restrictions

For load instructions:

- *Rt* can be SP or PC for word loads only.
- *Rt* must be different from *Rt2* for two-word loads.
- *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms.

When *Rt* is PC in a word load instruction:

- Bit[0] of the loaded value must be 1 for correct execution.
- A branch occurs to the address created by changing bit[0] of the loaded value to 0.
- If the instruction is conditional, it must be the last instruction in the IT block.

For store instructions:

- *Rt* can be SP for word stores only.
- *Rt* must not be PC.
- *Rn* must not be PC.
- *Rn* must be different from *Rt* and *Rt2* in the pre-indexed or post-indexed forms.

Condition flags

These instructions do not change the flags.

Examples

```
LDR     R8, [R10]              ; Loads R8 from the address in R10.
LDRNE   R2, [R5, #960]!        ; Loads (conditionally) R2 from a word
                               ; 960 bytes above the address in R5,
                               ; and increments R5 by 960.
STR     R2, [R9,#const-struc]  ; const-struc is an expression
                               ; evaluating to a constant in the range
                               ; 0-4095.
STRH    R3, [R4], #4           ; Store R3 as halfword data into
                               ; address in R4, then increment R4 by
                               ; 4.
LDRD    R8, R9, [R3, #0x20]    ; Load R8 from a word 32 bytes above
                               ; the address in R3, and load R9 from a
                               ; word 36 bytes above the address in
                               ; R3.
STRD    R0, R1, [R8], #-16     ; Store R0 to address in R8, and store
                               ; R1 to a word 4 bytes above the
                               ; address in R8, and then decrement R8
                               ; by 16.
```

subtraction shows instructions that subtract a 96-bit integer contained in R9, R1, and R11 from another contained in R6, R2, and R8. The example stores the result in R6, R9, and R2.

Example 3-5: 96-bit subtraction

```
SUBS    R6, R6, R9      ; Subtract the least significant words.
SBCS    R9, R2, R1      ; Subtract the middle words with carry.
SBC     R2, R8, R11     ; Subtract the most significant words with
                        ; carry.
```

3.5.2 AND, ORR, EOR, BIC, and ORN

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Syntax

op{S}{*cond*} {*Rd*,} *Rn*, *Operand2*

Where:

op Is one of:

- AND Logical AND.
- ORR Logical OR, or bit set.
- EOR Logical Exclusive OR.
- BIC Logical AND NOT, or bit clear.
- ORN Logical OR NOT.

S Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution on page 68*.

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rn Is the register holding the first operand.

Operand2 Is a flexible second operand. See *Flexible second operand on page 64* for details of the options.

Operation

The AND, EOR, and ORR instructions perform bitwise AND, Exclusive OR, and OR operations on the values in *Rn* and *Operand2*.

The BIC instruction performs an AND operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

The ORN instruction performs an OR operation on the bits in *Rn* with the complements of the corresponding bits in the value of *Operand2*.

Restrictions

Do not use SP and do not use PC.

3.5.10 SHADD16 and SHADD8

Signed Halving Add 16 and Signed Halving Add 8.

Syntax

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

Where:

op Is one of:

- `SHADD16` Signed Halving Add 16
- `SHADD8` Signed Halving Add 8

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rn Is the first operand register.

Rm Is the second operand register.

Operation

Use these instructions to add 16-bit and 8-bit data and then to halve the result before writing the result to the destination register.

The SHADD16 instruction:

1. Adds each halfword from the first operand to the corresponding halfword of the second operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the halfword results in the destination register.

The SHADDB8 instruction:

1. Adds each byte of the first operand to the corresponding byte of the second operand.
2. Shuffles the result by one bit to the right, halving the data.
3. Writes the byte results in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

```
SHADD16 R1, R0      ; Adds halfwords in R0 to corresponding halfword of R1
                    ; and writes halved result to corresponding halfword in
                    ; R1.
SHADD8  R4, R0, R5 ; Adds bytes of R0 to corresponding byte in R5 and
                    ; writes halved result to corresponding byte in R4.
```

Where:

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rn Is the first operand register.

Rm Is the second operand register.

Operation

The USAD8 instruction:

1. Subtracts each byte of the second operand register from the corresponding byte of the first operand register.
2. Adds the absolute values of the differences together.
1. Writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not change the flags.

Examples

```
USAD8 R1, R4, R0      ; Subtracts each byte in R0 from corresponding byte
                      ; of R4 adds the differences and writes to R1.
USAD8  R0, R5         ; Subtracts bytes of R5 from corresponding byte in
                      ; R0, adds the differences and writes to R0.
```

3.5.23 USADA8

Unsigned Sum of Absolute Differences and Accumulate.

Syntax

`USADA8{`*cond*`}{`*Rd*`,} `*Rn*`, `*Rm*`, `*Ra*

Where:

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rn Is the first operand register.

Rm Is the second operand register.

Ra Is the register that contains the accumulation value.

Operation

The USADA8 instruction:

The MLS instruction multiplies the values from *Rn* and *Rm*, subtracts the product from the value from *Ra*, and places the least significant 32 bits of the result in *Rd*.

The results of these instructions do not depend on whether the operands are signed or unsigned.

Restrictions

In these instructions, do not use SP and do not use PC.

If the S suffix is used with the MUL instruction:

- *Rd*, *Rn*, and *Rm* must all be in the range R0 to R7.
- *Rd* must be the same as *Rm*.
- The *cond* suffix must not be used.

Condition flags

If S is specified, the MUL instruction:

- Updates the N and Z flags according to the result.
- Does not affect the C and V flags.

Examples

```
MUL     R10, R2, R5      ; Multiply, R10 = R2 x R5
MLA     R10, R2, R1, R5 ; Multiply with accumulate, R10 = (R2 x R1) +
                         ; R5
MULS    R0, R2, R2       ; Multiply with flag update, R0 = R2 x R2
MULLT   R2, R3, R2       ; Conditionally multiply, R2 = R3 x R2
MLS     R4, R5, R6, R7   ; Multiply with subtract, R4 = R7 - (R5 x R6)
```

3.6.2 UMULL, UMAAL, UMLAL

Unsigned Long Multiply, with optional Accumulate, using 32-bit operands and producing a 64-bit result.

Syntax

op{cond} RdLo, RdHi, Rn, Rm

Where:

op Is one of:

- `UMULL` Unsigned Long Multiply.
- `UMAAL` Unsigned Long Multiply with Accumulate Accumulate.
- `UMLAL` Unsigned Long Multiply, with Accumulate.

cond Is an optional condition code. See *Conditional execution on page 68*.

RdHi, RdLo Are the destination registers. For `UMAAL`, `UMLAL` and `UMLAL` they also hold the accumulating value.

Rn, Rm Are registers holding the first and second operands.

3.7.2 SSAT16 and USAT16

Signed Saturate and Unsigned Saturate to any bit position for two halfwords.

Syntax

`op{cond} Rd, #n, Rm`

Where:

op Is one of:
- `SSAT16` Saturates a signed halfword value to a signed range.
- `USAT16` Saturates a signed halfword value to an unsigned range.

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

n Specifies the bit position to saturate to:
- *n* ranges from 1 to 16 for `SSAT`.
- *n* ranges from 0 to 15 for `USAT`.

Rm Is the register containing the value to saturate.

Operation

The SSAT16 instruction:

1. Saturates two signed 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two signed 16-bit halfwords to the destination register.

The USAT16 instruction:

1. Saturates two unsigned 16-bit halfword values of the register with the value to saturate from selected by the bit position in *n*.
2. Writes the results as two unsigned halfwords in the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

If saturation occurs, these instructions set the Q flag to 1.

Examples

```
SSAT16    R7, #9, R2        ; Saturates the top and bottom highwords of R2
                            ; as 9-bit values, writes to corresponding
                            ; halfword of R7.
USAT16NE  R0, #13, R5       ; Conditionally saturates the top and bottom
                            ; halfwords of R5 as 13-bit values, writes to
                            ; corresponding halfword of R0.
```

Condition flags

These instructions do not affect the condition code flags.

Examples

```
QASX    R7, R4, R2      ; Adds top halfword of R4 to bottom halfword of R2,
                        ; saturates to 16 bits, writes to top halfword of
                        ; R7, Subtracts top highword of R2 from bottom
                        ; halfword of R4, saturates to 16 bits and writes
                        ; to bottom halfword of R7
QSAX    R0, R3, R5      ; Subtracts bottom halfword of R5 from top halfword
                        ; of R3, saturates to 16 bits, writes to top
                        ; halfword of R0
                        ; Adds bottom halfword of R3 to top halfword of R5,
                        ; saturates to 16 bits, writes to bottom halfword
                        ; of R0.
```

3.7.5 QDADD and QDSUB

Saturating Double and Add and Saturating Double and Subtract, signed.

Syntax

op{cond} {Rd}, Rm, Rn

Where:

op Is one of:
- QDADD Saturating Double and Add.
- QDSUB Saturating Double and Subtract.

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rm, Rn Are registers holding the first and second operands.

Operation

The QDADD instruction:

- Doubles the second operand value.
- Adds the result of the doubling to the signed saturated value in the first operand.
- Writes the result to the destination register.

The QDSUB instruction:

- Doubles the second operand value.
- Subtracts the doubled value from the signed saturated value in the first operand.
- Writes the result to the destination register.

Both the doubling and the addition or subtraction have their results saturated to the 32-bit signed integer range $-2^{31} \leq x \leq 2^{31} - 1$. If saturation occurs in either operation, it sets the Q flag in the APSR.

Restrictions

Do not use SP and do not use PC.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the condition code flags.

Examples

```
UQASX   R7, R4, R2  ; Adds top halfword of R4 with bottom halfword of R2,
                    ; saturates to 16 bits, writes to top halfword of R7
                    ; Subtracts top halfword of R2 from bottom halfword of
                    ; R4, saturates to 16 bits, writes to bottom halfword
                    ; of R7
UQSAX   R0, R3, R5  ; Subtracts bottom halfword of R5 from top halfword of
                    ; R3, saturates to 16 bits, writes to top halfword of
                    ; R0
                    ; Adds bottom halfword of R4 to top halfword of R5
                    ; saturates to 16 bits, writes to bottom halfword of
                    ; R0.
```

3.9.2 SBFX and UBFX

Signed Bit Field Extract and Unsigned Bit Field Extract.

Syntax

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

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

Where:

cond Is an optional condition code. See *Conditional execution on page 68*.

Rd Is the destination register.

Rn Is the source register.

lsb Is the position of the least significant bit of the bit field. *lsb* must be in the range 0 to 31.

width Is the width of the bit field and must be in the range 1 to 32–*lsb*.

Operation

SBFX extracts a bit field from one register, sign extends it to 32 bits, and writes the result to the destination register.

UBFX extracts a bit field from one register, zero extends it to 32 bits, and writes the result to the destination register.

Restrictions

Do not use SP and do not use PC.

Condition flags

These instructions do not affect the flags.

Examples

```
SBFX  R0, R1, #20, #4  ; Extract bit 20 to bit 23 (4 bits) from R1 and
                       ; sign extend to 32 bits and then write the
                       ; result to R0.
UBFX  R8, R11, #9, #10 ; Extract bit 9 to bit 18 (10 bits) from R11 and
                       ; zero extend to 32 bits and then write the
                       ; result to R8.
```

Other restrictions when using an IT block are:

- A branch or any instruction that modifies the PC must either be outside an IT block or must be the last instruction inside the IT block. These are:
  - ADD PC, PC, Rm.
  - MOV PC, Rm.
  - B, BL, BX, BLX.
  - Any LDM, LDR, or POP instruction that writes to the PC.
  - TBB and TBH.
- Do not branch to any instruction inside an IT block, except when returning from an exception handler
- All conditional instructions except B*cond* must be inside an IT block. B*cond* can be either outside or inside an IT block but has a larger branch range if it is inside one
- Each instruction inside the IT block must specify a condition code suffix that is either the same or logical inverse as for the other instructions in the block.

The assembler might place extra restrictions on the use of IT blocks, such as prohibiting the use of assembler directives within them.

Condition flags

This instruction does not change the flags.

Example

```
ITTE   NE           ; Next 3 instructions are conditional
ANDNE  R0, R0, R1   ; ANDNE does not update condition flags
ADDSNE R2, R2, #1   ; ADDSNE updates condition flags
MOVEQ  R2, R3       ; Conditional move


CMP    R0, #9       ; Convert R0 hex value (0 to 15) into ASCII
                    ; ('0'-'9', 'A'-'F')
ITE    GT           ; Next 2 instructions are conditional
ADDGT  R1, R0, #55  ; Convert 0xA -> 'A'
ADDLE  R1, R0, #48  ; Convert 0x0 -> '0'


IT     GT           ; IT block with only one conditional instruction
ADDGT  R1, R1, #1   ; Increment R1 conditionally


ITTEE  EQ           ; Next 4 instructions are conditional
MOVEQ  R0, R1       ; Conditional move
ADDEQ  R2, R2, #10  ; Conditional add
ANDNE  R3, R3, #1   ; Conditional AND
BNE.W  dloop        ; Branch instruction can only be used in the last
                    ; instruction of an IT block


IT     NE           ; Next instruction is conditional
ADD    R0, R0, R1   ; Syntax error: no condition code used in IT block
```

Operation

These instructions:

1. Read the source register.
2. Round to the nearest integer value in floating-point format using the rounding mode specified by the FPSCR.
3. Write the result to the destination register.
4. For the VRINTZX instruction only. Generate a floating-point exception if the result is not exact.

Restrictions

There are no restrictions.

Condition flags

These instructions do not change the flags.

3.11.35 VRINTA, VRINTN, VRINTP, VRINTM, VRINTZ

Round a floating-point value to an integer in floating-point format using directed rounding.

Encoding

`VRINT<rmode>.F<32|64> <Sd|Dd>, <Sm|Dm>`

Where:

<Sd|Dd> Is the destination single-precision or double-precision floating-point value.

<Sn|Dn>, <Sm|Dm>
Are the operand single-precision or double-precision floating-point values.

<rmode> Is one of:

- `A` Round to nearest ties away.
- `M` Round to Nearest Even.
- `N` Round towards Plus Infinity.
- `P` Round towards Minus Infinity.
- `Z` Round towards Zero.

Operation

These instructions:

1. Read the source register.
2. Round to the nearest integer value with a directed rounding mode specified by the instruction.
3. Write the result to the destination register.

Restrictions

These instructions cannot be conditional. These instructions cannot generate an inexact exception even if the result is not exact.

Table 43. ISPR bit assignments

Bits	Name	Function
[31:0]	SETPEND	Interrupt set-pending bits. Write: 0: No effect. 1: Changes interrupt state to pending. Read: 0: Interrupt is not pending. 1: Interrupt is pending.

Writing 1 to the ISPR bit corresponding to:

- An interrupt that is pending has no effect.
- A disabled interrupt sets the state of that interrupt to pending.

4.2.5 Interrupt clear-pending registers

The NVIC_ICPR0-NCVIC_ICPR7 registers remove the pending state from interrupts, and show which interrupts are pending. See the register summary in *Table 39 on page 184* for the register attributes.

The bit assignments are:

Figure 20. ICPR bit assignments

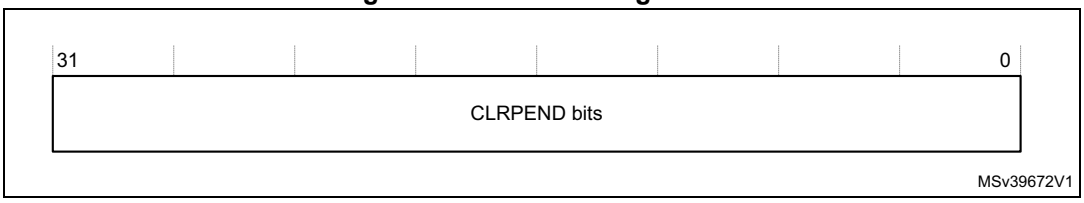

Table 44. ICPR bit assignments

Bits	Name	Function
[31:0]	CLRPEND	Interrupt clear-pending bits. Write: 0: No effect. 1: Removes pending state an interrupt. Read: 0: Interrupt is not pending. 1: Interrupt is pending.

Writing 1 to an ICPR bit does not affect the active state of the corresponding interrupt.

4.6.1 MPU Type register

The MPU_TYPE register indicates whether the MPU is present, and if so, how many regions it supports. If the MPU is not present the MPU_TYPE register is RAZ. See the register summary in *Table 83* for its attributes. The bit assignments are:

Figure 48. TYPE bit assignments

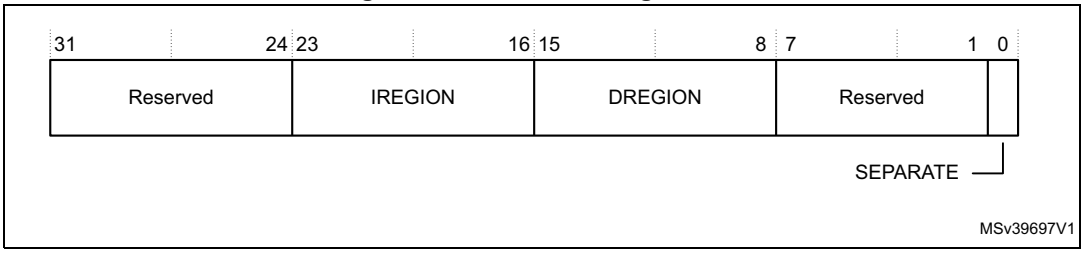

Table 84. TYPE bit assignments

Bits	Name	Function
[31:24]	-	Reserved.
[23:16]	IREGION	Indicates the number of supported MPU instruction regions. Always contains 0x00. The MPU memory map is unified and is described by the DREGION field.
[15:8]	DREGION	Indicates the number of supported MPU data regions: 0x08: 8 MPU regions. 0x0F: 16 MPU regions.
[7:1]	-	Reserved.
[0]	SEPARATE	Indicates support for unified or separate instruction and date memory maps: 0: Unified.

4.6.2 MPU Control register

The MPU_CTRL register:

- Enables the MPU.
- Enables the default memory map background region.
- Enables use of the MPU when in the hard fault, *Non Maskable Interrupt* (NMI), and FAULTMASK escalated handlers.

See the register summary in *Table 83 on page 222* for the MPU_CTRL attributes. The bit assignments are:

Figure 49. MPU_CTRL bit assignments

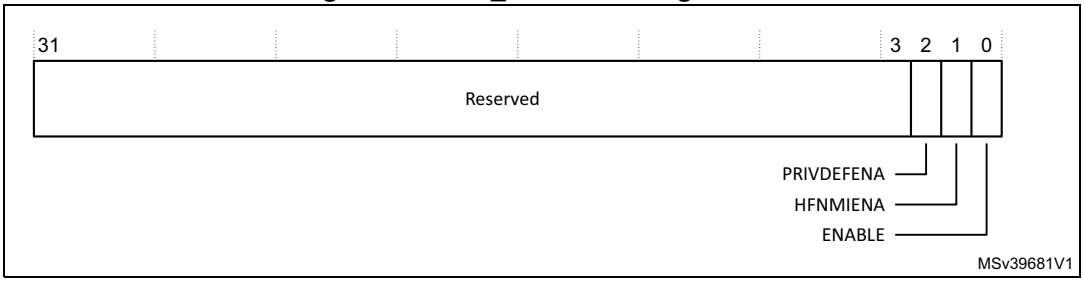

Table 98. FPDSCR bit assignments (continued)

Bits	Name	Function
[25]	DN	Default value for FPSCR.DN
[24]	FZ	Default value for FPSCR.FZ
[23:22]	RMode	Default value for FPSCR.RMode
[21:0]	-	Reserved

4.7.6 Enabling the FPU

The FPU is disabled from reset. The user must enable it before using any floating-point instructions. *Example 4-1: Enabling the FPU* shows an example code sequence for enabling the FPU in privileged mode. The processor must be in privileged mode to read from and write to the CPACR.

Example 4-1: Enabling the FPU

```
CPACR   EQU  0xE000ED88
LDR     R0,  =CPACR          ; Read CPACR
LDR     r1, [R0]             ; Set bits 20-23 to enable CP10 and CP11
                             ; coprocessors
ORR     R1, R1, #(0xF << 20)
STR     R1, [R0]             ; Write back the modified value to the CPACR
DSB
ISB                          ; Reset pipeline now the FPU is enabled.
```

4.8 Cache maintenance operations

The cache maintenance operations are only accessible by privileged loads and stores. Unprivileged accesses to these registers always generate a BusFault.

Table 99. Cache maintenance space register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000EF50	ICIALLU	WO	Privileged	Unknown	Instruction cache invalidate all to the *Point of Unification* (PoU)(1)
0xE000EF54	-	-	-	-	Reserved
0xE000EF58	ICIMVAU	WO	Privileged	Unknown	Instruction cache invalidate by address to the PoU(1)
0xE000EF5C	DCIMVAC	WO	Privileged	Unknown	Data cache invalidate by address to the *Point of Coherency* (PoC)(2)
0xE000EF60	DCISW	WO	Privileged	Unknown	Data cache invalidate by set/way
0xE000EF64	DCCMVAU	WO	Privileged	Unknown	Data cache by address to the PoU(1)
0xE000EF68	DCCMVAC	WO	Privileged	Unknown	Data cache clean by address to the PoC(2)
0xE000EF6C	DCCSW	WO	Privileged	Unknown	Data cache clean by set/way
0xE000EF70	DCCIMVAC	WO	Privileged	Unknown	Data cache clean and invalidate by address to the PoC(2)