B1.1 About the Application level programmers' model

This chapter contains the programmers' model information required for application development.

The information in this chapter is distinct from the system information required to service and support application execution under an operating system, or higher level of system software. However, some knowledge of the system information is needed to put the Application level programmers' model into context.

Depending on the implementation choices, the architecture supports multiple levels of execution privilege, indicated by different *Exception levels* that number upwards from EL0 to EL3. EL0 corresponds to the lowest privilege level and is often described as unprivileged. The Application level programmers' model is the programmers' model for software executing at EL0. For more information, see *Exception levels*.

System software determines the Exception level, and therefore the level of privilege, at which software runs. When an operating system supports execution at both EL1 and EL0, an application usually runs unprivileged at EL0. This:

- Permits the operating system to allocate system resources to an application in a unique or shared manner.
- Provides a degree of protection from other processes, and so helps protect the operating system from malfunctioning software.

This chapter indicates where some system level understanding is necessary, and where relevant it gives a reference to the system level description.

Execution at any Exception level above EL0 is often referred to as privileged execution.

For more information on the system level view of the architecture refer to Chapter D1 *The AArch64 System Level Programmers' Model*.

B1.2 Registers in AArch64 Execution state

The following registers are visible at EL0 using AArch64:

R0-R30 31 general-purpose registers, R0 to R30. Each can be accessed as:

- A 64-bit general-purpose register named X0 to X30.
- A 32-bit general-purpose register named W0 to W30.

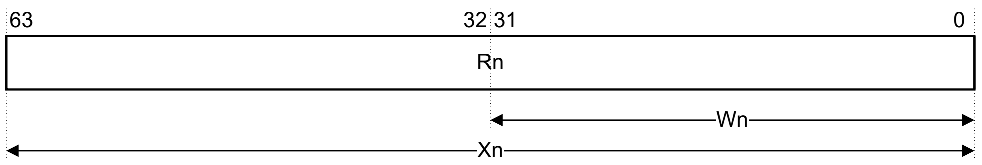

Figure B1-1 General-purpose register naming

The X30 general-purpose register is used as the procedure call link register.

SP A 64-bit dedicated Stack Pointer register. The least significant 32 bits of the stack pointer can be accessed using the register name WSP.

The use of SP as an operand in an instruction, indicates the use of the current stack pointer.

Note

Stack pointer alignment to a 16-byte boundary is configurable at EL1. For more information, see the *Procedure Call Standard for the Arm 64-bit Architecture*.

PC A 64-bit Program Counter holding the address of the current instruction.

Software cannot write directly to the PC. It can be updated only on a branch, exception entry or exception return.

Note

Attempting to execute an A64 instruction that is not word-aligned generates a PC alignment fault, see *PC alignment checking*.

V0-V31 32 SIMD&FP registers, V0 to V31. Each can be accessed as:

- A 128-bit register named Q0 to Q31.
- A 64-bit register named D0 to D31.
- A 32-bit register named S0 to S31.
- A 16-bit register named H0 to H31.
- An 8-bit register named B0 to B31.
- A 128-bit vector of elements. See *SIMD vectors in AArch64 state*.
- A 64-bit vector of elements. See *SIMD vectors in AArch64 state*.

Where the number of bits described by a register name does not occupy an entire SIMD&FP register, it refers to the least significant bits. See Figure B1-2.

For more information about data types and vector formats, see *Supported data types*.

B1.3 Process state, PSTATE

Process state, or PSTATE, is an abstraction of process state information. All of the instruction sets provide instructions that operate on elements of PSTATE.

For the system level view of PSTATE, see *Process state, PSTATE* in Chapter D1.

The following PSTATE information is accessible at EL0:

The Condition flags

Flag-setting instructions set these. They are:

N Negative Condition flag. If the result of the instruction is regarded as a two's complement signed integer, the PE sets this to:

- 1 if the result is negative.
- 0 if the result is positive or zero.

Z Zero Condition flag. Set to:

- 1 if the result of the instruction is zero.
- 0 otherwise.

A result of zero often indicates an equal result from a comparison.

C Carry Condition flag. Set to:

- 1 if the instruction results in a carry condition, for example an unsigned overflow that is the result of an addition.
- 0 otherwise.

V Overflow Condition flag. Set to:

- 1 if the instruction results in an overflow condition, for example a signed overflow that is the result of an addition.
- 0 otherwise.

Conditional instructions test the N, Z, C and V Condition flags, combining them with the Condition code for the instruction to determine whether the instruction must be executed. In this way, execution of the instruction is conditional on the result of a previous operation. For more information about conditional execution, see *Condition flags and related instructions*.

The exception masking bits

D Debug exception mask bit. When EL0 is enabled to modify the mask bits, this bit is visible and can be modified. However, this bit is architecturally ignored at EL0.

A SError interrupt mask bit.

I IRQ interrupt mask bit.

F FIQ interrupt mask bit.

For each bit, the values are:

0 Exception not masked.

1 Exception masked.

Access at EL0 using AArch64 state depends on SCTLR_EL1.UMA.

C1.2.3 Instruction Mnemonics

The A64 assembly language overloads instruction mnemonics and distinguishes between the different forms of an instruction based on the operand types. For example, the following ADD instructions all have different opcodes. However, the programmer must remember only one mnemonic, as the assembler automatically chooses the correct opcode based on the operands. The disassembler follows the same procedure in reverse.

Example C1-1 ADD instructions with different opcodes

```
ADD W0, W1, W2               // add 32-bit register
ADD X0, X1, X2               // add 64-bit register
ADD X0, X1, W2, SXTW         // add 64-bit extended register
ADD X0, X1, #42              // add 64-bit immediate
```

C1.2.4 Condition code

The A64 ISA has some instructions that set Condition flags or test Condition codes or both. For information about instructions that set the Condition flags or use the condition mnemonics, see *Condition flags and related instructions*.

Table C1-1 shows the available Condition codes.

Table C1-1 Condition codes

cond	Mnemonic	Meaning (integer)	Meaning (floating-point)[a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal or unordered	Z == 0
0010	CS or HS	Carry set	Greater than, equal, or unordered	C == 1
0011	CC or LO	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Ordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C ==1 && Z == 0
1001	LS	Unsigned lower or same	Less than or equal	!(C ==1 && Z ==0)
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 && N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z == 0 && N == V)
1110	AL	Always	Always	Any
1111	NV[b]	Always	Always	Any

a. Unordered means at least one NaN operand.

b. The Condition code NV exists only to provide a valid disassembly of the 0b1111 encoding, otherwise its behavior is identical to AL.

C1.2.5 SVE Condition code aliases

The *SVE* assembler syntax defines an alternative set of SVE condition code aliases for use with AArch64 conditional instructions, as follows:

Table C1-2 shows the available SVE Condition code aliases.

Table C1-2 SVE Condition codes

cond	Mnemonic	SVE alias	Meaning	Condition flags
`0000`	`EQ`	`NONE`	All *Active elements* were FALSE or there were no *Active elements*.	Z == 1
`0001`	`NE`	`ANY`	An *Active element* was TRUE.	Z == 0
`0010`	`CS` or `HS`	`NLAST`	The *Last active element* was FALSE or there were no *Active elements*.	C == 1
`0011`	`CC` or `LO`	`LAST`	The *Last active element* was TRUE.	C == 0
`0100`	`MI`	`FIRST`	The *First active element* was TRUE.	N == 1
`0101`	`PL`	`NRFST`	The *First active element* was FALSE or there were no *Active elements*.	N == 0
`0110`	`VS`	-	`CTERM` comparison failed, but end of partition reached.	V == 1
`0111`	`VC`	-	`CTERM` comparison succeeded, or end of partition not reached.	V == 0
`1000`	`HI`		An *Active element* was TRUE, but the *Last active element* was FALSE.	C ==1 && Z == 0
`1001`	`LS`	`PLAST`	The *Last active element* was TRUE, or all *Active elements* were FALSE, or there were no *Active elements*.	C ==0 || Z ==1
`1010`	`GE`	`TCONT`	`CTERM` termination condition not detected.	N == V
`1011`	`LT`	`TSTOP`	`CTERM` termination condition detected.	N != V

C1.2.6 Register names

See:

- *General-purpose register file and zero register and stack pointer*.
- *Advanced SIMD and floating-point register file*.
- *Advanced SIMD and floating-point scalar register names*.
- *SIMD vector register names*.
- *SIMD vector element names*.
- For SVE register names, see *Z0-Z31*. *P0-P15*, and *FFR, First Fault Register*.
- For SME ZA storage, see *ZA array vector access* and *ZA tile access*.
- For SME2 ZT0 storage, see *ZT0*.

C1.2.6.1 General-purpose register file and zero register and stack pointer

The 31 general-purpose registers in the general-purpose register file are named R0-R30 and encoded in the instruction register fields with values 0-30. In a general-purpose register field the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position.

When the registers are used in a specific instruction variant, they must be qualified to indicate the operand data size, 32 bits or 64 bits, and the data size of the instruction.

When the data size is 32 bits, the lower 32 bits of the register are used and the upper 32 bits are ignored on a read and cleared to zero on a write.

Table C1-3 shows the qualified names for registers, where *n* is a register number 0-30.

Table C1-3 Naming of general-purpose registers, the zero register, and the stack pointer

Name	Size	Encoding	Description
Wn	32 bits	0-30	General-purpose register 0-30
Xn	64 bits	0-30	General-purpose register 0-30
WZR	32 bits	31	Zero register
XZR	64 bits	31	Zero register
WSP	32 bits	31	Current stack pointer
SP	64 bits	31	Current stack pointer

This list gives more information about the instruction arguments shown in Table C1-3:

- The names Xn and Wn both refer to the same general-purpose register, Rn.
- There is no register named W31 or X31.
- The name SP represents the stack pointer for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as a read or write of the current stack pointer. When instructions do not interpret this operand encoding as the stack pointer, use of the name SP is an error.
- The name WSP represents the current stack pointer in a 32-bit context.
- The name XZR represents the zero register for 64-bit operands where an encoding of the value 31 in the corresponding register field is interpreted as returning zero when read or discarding the result when written. When instructions do not interpret this operand encoding as the zero register, use of the name XZR is an error.
- The name WZR represents the zero register in a 32-bit context.
- The architecture does not define a specific name for general-purpose register R30 to reflect its role as the link register on procedure calls. However, an A64 assembler must always use W30 and X30 for this purpose, and additional software names might be defined as part of the Procedure Call Standard, see *Procedure Call Standard for the Arm 64-bit Architecture*.

C1.2.6.2 Advanced SIMD and floating-point register file

The 32 registers in the Advanced SIMD and floating-point register file, V0-V31, hold floating-point operands for the scalar floating-point instructions, and both scalar and vector operands for the Advanced SIMD instructions. When they are used in a specific instruction form, the names must be further qualified to indicate the data shape, that is the data element size and the number of elements or lanes within the register. A similar requirement is placed on the general-purpose registers. See *General-purpose register file and zero register and stack pointer*.

Note

The data type is described by the instruction mnemonics that operate on the data. The data type is not described by the register name. The data type is the interpretation of bits within each register or vector element, whether these are integers, floating-point values, polynomials, or cryptographic hashes.

C1.3 Address generation

The A64 instruction set supports 64-bit virtual addresses (VAs). The valid VA range is determined by the following factors:

- The size of the implemented virtual address space.
- *Memory Management Unit* (MMU) configuration settings.

Limits on the VA size mean that the most significant bits of the virtual address do not hold valid address bits. These unused bits can hold:

- A tag, see *Address tagging*.
- If FEAT_PAuth is implemented, a Pointer authentication code (PAC), see *Pointer authentication*.

For more information on memory management and address translation, see Chapter D8 *The AArch64 Virtual Memory System Architecture*.

C1.3.1 Register indexed addressing

The A64 instruction set allows a 64-bit index register to be added to the 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling.

C1.3.2 PC-relative addressing

The A64 instruction set has support for position-independent code and data addressing:

- PC-relative literal loads have an offset range of ± 1MB.
- Process state flag and compare based conditional branches have a range of ± 1MB. Test bit conditional branches have a restricted range of ± 32KB.
- Unconditional branches, including branch and link, have a range of ± 128MB.

PC-relative load/store operations, and address generation with a range of ± 4GB can be performed using two instructions.

C1.3.3 Load/store addressing modes

Load/store addressing modes in the A64 instruction set require a 64-bit base address from a general-purpose register X0-X30 or the current stack pointer, `SP`, with an optional immediate or register offset. Table C1-7 shows the assembler syntax for the complete set of load/store addressing modes.

Table C1-7 A64 Load/store addressing modes

Addressing Mode	Offset		
	Immediate	Register	Extended Register
Base register only (no offset)	`[base{, #0}]`	-	-
Base plus offset	`[base{, #imm}]`	`[base, Xm{, LSL #imm}]`	`[base, Wm, (S|U)XT(X|W) {#imm}]`
Pre-indexed	`[base, #imm]!`	-	-
Post-indexed	`[base], #imm`	`[base], Xm`[a]	-
Literal (PC-relative)	`label`	-	-

a. The post-indexed by register offset mode can be used with the SIMD load/store structure instructions described in *Load/store Advanced SIMD*. Otherwise the post-indexed by register offset mode is not available.

Some types of load/store instruction support only a subset of the load/store addressing modes listed in Table C1-7. Details of the supported modes are as follows:

- Base plus offset addressing means that the address is the value in the 64-bit base register plus an offset.
- Pre-indexed addressing means that the address is the sum of the value in the 64-bit base register and an offset, and the address is then written back to the base register.
- Post-indexed addressing means that the address is the value in the 64-bit base register, and the sum of the address and the offset is then written back to the base register.
- Literal addressing means that the address is the value of the 64-bit program counter for this instruction plus a 19-bit signed word offset. This means that it is a 4 byte aligned address within ±1MB of the address of this instruction with no offset. Literal addressing can be used only for loads of at least 32 bits and for prefetch instructions. The PC cannot be referenced using any other addressing modes. The syntax for labels is specific to individual toolchains.
- An immediate offset can be unsigned or signed, and scaled or unscaled, depending on the type of load/store instruction. When the immediate offset is scaled it is encoded as a multiple of the transfer size, although the assembly language always uses a byte offset, and the assembler or disassembler performs the necessary conversion. The usable byte offsets therefore depend on the type of load/store instruction and the transfer size.

  Table C1-8 shows the offset and the type of load/store instruction.

  Table C1-8 Immediate offsets and the type of load/store instruction

Offset bits	Sign	Scaling	Write-Back	Load/store type
0	-	-	-	Exclusive/acquire/release
7	Signed	Scaled	Optional	Register pair
9	Signed	Unscaled	Optional	Single register
12	Unsigned	Scaled	No	Single register

- A register offset means that the offset is the 64 bits from a general-purpose register, Xm, optionally scaled by the transfer size, in bytes, if `LSL #imm` is present and where `imm` must be equal to log2(transfer_size). The `SXTX` extend/shift option is functionally equivalent to `LSL`, but the `LSL` option is preferred in source code.
- An extended register offset means that offset is the bottom 32 bits from a general-purpose register Wm, sign-extended or zero-extended to 64 bits, and then scaled by the transfer size if so indicated by `#imm`, where imm must be equal to log2(transfer_size). An assembler must accept Wm or Xm as an extended register offset, but Wm is preferred for disassembly.
- Generating an address lower than the value in the base register requires a negative signed immediate offset or a register offset holding a negative value.
- When stack alignment checking is enabled by system software and the base register is the SP, the current stack pointer must be initially quadword aligned, that is aligned to 16 bytes. Misalignment generates a Stack Alignment fault. The offset does not have to be a multiple of 16 bytes unless the specific load/store instruction requires this. SP cannot be used as a register offset.

C1.3.3.1 Address calculation

General-purpose arithmetic instructions can calculate the result of most addressing modes and write the address to a general-purpose register or, in most cases, to the current stack pointer.

Table C1-9 shows the arithmetic instructions that can compute addressing modes.

Table C1-9 Arithmetic instructions to compute addressing modes

Addressing Form	Offset		
	Immediate	Register	Extended Register
Base register (no offset)	`MOV Xd|SP, base`	-	-
Base plus offset	`ADD Xd|SP, base, #imm` or `SUB Xd|SP, base, #imm`	`ADD <Xd|SP>, base, Xm{,LSL#imm}`	`ADD <Xd|SP>, base, Wm,(S|U)XT(W|H|B|X) {#imm}`
Pre-indexed	-	-	-
Post-indexed	-	-	-
Literal (PC-relative)	`ADR Xd, label`	-	-

Note

- For the 64-bit base plus register offset form, the UXTX mnemonic is an alias for the LSL shift option, but LSL is preferred for disassembly. Similarly the SXTX extend/shift option is functionally equivalent to the LSL option, but the LSL option is preferred in source code.
- To calculate a base plus immediate offset the ADD instructions defined in *Arithmetic (immediate)* accept an unsigned 12-bit immediate offset, with an optional left shift by 12. This means that a single ADD instruction cannot support the full range of byte offsets available to a single register load/store with a scaled 12-bit immediate offset. For example, a quadword LDR effectively has a 16-bit byte offset. To calculate an address with a byte offset that requires more than 12 bits it is necessary to use two ADD instructions. The following example shows this:

```
ADD  Xd, base, #(imm & 0xFFF)
ADD  Xd, Xd, #(imm>>12), LSL #12
```

- To calculate a base plus extended register offset, the ADD instructions defined in *Arithmetic (extended register)* provide a superset of the addressing mode that also supports sign-extension or zero-extension of a byte or halfword value with any shift amount between 0 and 4, for example:

```
ADD   Xd, base, Wm, SXTW #3    // Xd = base + (SignExtend(Wm) LSL 3)
ADD   Xd, base, Wm, UXTH #4    // Xd = base + (ZeroExtend(Wm<15:0>) LSL 4)
```

- If the same extended register offset is used by more than one load/store instruction, then, depending on the implementation, it might be more efficient to calculate the extended and scaled intermediate result just once, and then reuse it as a simple register offset. The extend and scale calculation can be performed using the SBFIZ and UBFIZ bitfield instructions defined in *Bitfield move*, for example:

```
SBFIZ Xd, Xm, #3, #32    //Xd = "Wm, SXTW #3"
UBFIZ Xd, Xm, #4, #16    //Xd = "Wm, UXTH #4"
```

C3.1 Branches, Exception generating, and System instructions

This section describes the branch, exception generating, and System instructions. It contains the following subsections:

- *Conditional branch*.
- *Unconditional branch (immediate)*.
- *Unconditional branch (register)*.
- *Exception generation and return*.
- *System register instructions*.
- *System instructions*.
- *Hint instructions*.
- *Barriers and CLREX instructions*.
- *Pointer authentication instructions*.

For information about the encoding structure of the instructions in this instruction group, see *Branches, Exception Generating and System instructions*.

Note

Software must:

- Use only `BLR` or `BL` to perform a nested subroutine call when that subroutine is expected to return to the immediately following instruction, that is, the instruction with the address of the `BLR` or `BL` instruction incremented by four.
- Use only `RET` to perform a subroutine return, when that subroutine is expected to have been entered by a `BL` or `BLR` instruction.
- Use only `B`, `BR`, or the instructions listed in Table C3-1 to perform a control transfer that is not a subroutine call or subroutine return described in this *Note*.

C3.1.1 Conditional branch

Conditional branches change the flow of execution depending on the current state of the Condition flags or the value in a general-purpose register. See Table C1-1 for a list of the Condition codes that can be used for `cond`.

Table C3-1 shows the Conditional branch instructions.

Table C3-1 Conditional branch instructions

Mnemonic	Instruction	Branch offset range from the PC	See
`B.cond`	Branch conditionally	±1MB	*B.cond*
`BC.cond`	Branch Consistent conditionally	±1MB	*BC.cond*
`CBNZ`	Compare and branch if nonzero	±1MB	*CBNZ*
`CBZ`	Compare and branch if zero	±1MB	*CBZ*
`TBNZ`	Test bit and branch if nonzero	±32KB	*TBNZ*
`TBZ`	Test bit and branch if zero	±32KB	*TBZ*

C3.1.2 Unconditional branch (immediate)

Unconditional branch (immediate) instructions change the flow of execution unconditionally by adding an immediate offset with a range of ±128MB to the value of the program counter that fetched the instruction. The `BL` instruction also writes the address of the sequentially following instruction to general-purpose register, X30.

Table C3-2 shows the Unconditional branch instructions with an immediate branch offset.

Table C3-2 Unconditional branch instructions (immediate)

Mnemonic	Instruction	Immediate branch offset range from the PC	See
`B`	Branch unconditionally	±128MB	*B*
`BL`	Branch with link	±128MB	*BL*

C3.1.3 Unconditional branch (register)

Unconditional branch (register) instructions change the flow of execution unconditionally by setting the program counter to the value in a general-purpose register. The `BLR` instruction also writes the address of the sequentially following instruction to general-purpose register X30. The `RET` instruction behaves identically to `BR`, but provides an additional hint to the PE that this is a return from a subroutine. Table C3-3 shows Unconditional branch instructions that jump directly to an address held in a general-purpose register.

Table C3-3 Unconditional branch instructions (register)

Mnemonic	Instruction	See
`BLR`	Branch with link to register	*BLR*
`BR`	Branch to register	*BR*
`RET`	Return from subroutine	*RET*

C3.1.4 Exception generation and return

This section describes the following exceptions:

- *Exception generating*.
- *Exception return*.
- *Debug state*.

C3.1.4.1 Exception generating

Table C3-4 shows the Exception generating instructions.

Table C3-4 Exception generating instructions

Mnemonic	Instruction	See
`BRK`	Breakpoint Instruction	*BRK*
`HLT`	Halt Instruction	*HLT*

Table C3-4 Exception generating instructions (continued)

Mnemonic	Instruction	See
`HVC`	Generate exception targeting Exception level 2	*HVC*
`SMC`	Generate exception targeting Exception level 3	*SMC*
`SVC`	Generate exception targeting Exception level 1	*SVC*

C3.1.4.2 Exception return

Table C3-5 shows the Exception return instructions.

Table C3-5 Exception return instructions

Mnemonic	Instruction	See
`ERET`	Exception return using current ELR and SPSR	*ERET*

C3.1.4.3 Debug state

Table C3-6 shows the Debug state instructions.

Table C3-6 Debug state instructions

Mnemonic	Instruction	See
`DCPS1`	Debug switch to Exception level 1	*DCPS1*
`DCPS2`	Debug switch to Exception level 2	*DCPS2*
`DCPS3`	Debug switch to Exception level 3	*DCPS3*
`DRPS`	Debug restore PE state	*DRPS*

C3.1.5 System register instructions

For detailed information about the System register instructions, see Chapter C5 *The A64 System Instruction Class*. Table C3-7 shows the System register instructions.

If FEAT_SYSREG128 is implemented, the following instructions are added that allow the PE to move values between a 128-bit System register and two adjacent 64-bit general-purpose registers:

- MRRS.
- MSRR.

Table C3-7 System register instructions

Mnemonic	Instruction	See
`MRS`	Move System register to general-purpose register	*MRS*
`MSR`	Move general-purpose register to System register	*MSR (register)*
	Move immediate to PE state field	*MSR (immediate)*
`MRRS`	Move 128-bit System register to two adjacent 64-bit general-purpose registers	*MRRS*
`MSRR`	Move two adjacent 64-bit general-purpose registers to 128-bit System register	*MSRR*

C3.2 Loads and stores

This section describes the load/store instructions. It contains the following subsections:

- *Load/store register*.
- *Load/store register (unscaled offset)*.
- *Load/store pair*.
- *Load/store non-temporal pair*.
- *Load/store unprivileged*.
- *Load-Exclusive/Store-Exclusive*.
- *Load-Acquire/Store-Release*.
- *LoadLOAcquire/StoreLORelease*.
- *Load/store scalar SIMD and floating-point*.
- *Load/store Advanced SIMD*.
- *Prefetch memory*.
- *Atomic instructions*.
- *Memory Tagging instructions*.
- *Memory Copy and Memory Set instructions*.

The requirements for the alignment of data memory accesses are strict. For more information, see *Alignment of data accesses*.

The additional control bits SCTLR_ELx.SA and SCTLR_EL1.SA0 control whether the stack pointer must be quadword aligned when used as a base register. See *SP alignment checking*. Using a misaligned stack pointer generates an SP alignment fault exception.

For information about the encoding structure of the instructions in this instruction group, see *Loads and Stores*.

Note

In some cases, load/store instructions can lead to CONSTRAINED UNPREDICTABLE behavior. See *AArch64 CONSTRAINED UNPREDICTABLE behaviors*.

C3.2.1 Load/store register

The load/store register instructions support the following addressing modes:

- Base plus a scaled 12-bit unsigned immediate offset or base plus an unscaled 9-bit signed immediate offset.
- Base plus a 64-bit register offset, optionally scaled.
- Base plus a 32-bit extended register offset, optionally scaled.
- Pre-indexed by an unscaled 9-bit signed immediate offset.
- Post-indexed by an unscaled 9-bit signed immediate offset.
- PC-relative literal for loads of 32 bits or more.

See also *Load/store addressing modes*.

If a Load instruction specifies writeback and the register being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the load using the specified addressing mode and the base register becomes UNKNOWN. In addition, if an exception occurs during the execution of such an instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store instruction performs a writeback and the register that is stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs the store to the designated register using the specified addressing mode, but the value stored is UNKNOWN.

Table C3-17 shows the load/store register instructions.

Table C3-17 Load/store register instructions

Mnemonic	Instruction	See
`LDR`	Load register (register offset)	*LDR (register)*
	Load register (immediate offset)	*LDR (immediate)*
	Load register (PC-relative literal)	*LDR (literal)*
`LDRB`	Load byte (register offset)	*LDRB (register)*
	Load byte (immediate offset)	*LDRB (immediate)*
`LDRSB`	Load signed byte (register offset)	*LDRSB (register)*
	Load signed byte (immediate offset)	*LDRSB (immediate)*
`LDRH`	Load halfword (register offset)	*LDRH (register)*
	Load halfword (immediate offset)	*LDRH (immediate)*
`LDRSH`	Load signed halfword (register offset)	*LDRSH (register)*
	Load signed halfword (immediate offset)	*LDRSH (immediate)*
`LDRSW`	Load signed word (register offset)	*LDRSW (register)*
	Load signed word (immediate offset)	*LDRSW (immediate)*
	Load signed word (PC-relative literal)	*LDRSW (literal)*
`STR`	Store register (register offset)	*STR (register)*
	Store register (immediate offset)	*STR (immediate)*
`STRB`	Store byte (register offset)	*STRB (register)*
	Store byte (immediate offset)	*STRB (immediate)*
`STRH`	Store halfword (register offset)	*STRH (register)*
	Store halfword (immediate offset)	*STRH (immediate)*

C3.2.2 Load/store register (unscaled offset)

The load/store register instructions with an unscaled offset support only one addressing mode:

- Base plus an unscaled 9-bit signed immediate offset.

See *Load/store addressing modes*.

The load/store register (unscaled offset) instructions are required to disambiguate this instruction class from the load/store register instruction forms that support an addressing mode of base plus a scaled, unsigned 12-bit immediate offset, because that can represent some offset values in the same range.

The ambiguous immediate offsets are byte offsets that are both:

- In the range 0-255, inclusive.
- Naturally aligned to the access size.

Other byte offsets in the range -256 to 255 inclusive are unambiguous. An assembler program translating a load/store instruction, for example `LDR`, is required to encode an unambiguous offset using the unscaled 9-bit offset form, and to encode an ambiguous offset using the scaled 12-bit offset form. A programmer might force the generation of the unscaled 9-bit form by using one of the mnemonics in Table C3-18. Arm recommends that a disassembler outputs all unscaled 9-bit offset forms using one of these mnemonics, but unambiguous offsets can be output using a load/store single register mnemonic, for example, `LDR`.

Table C3-18 shows the load/store register instructions with an unscaled offset.

Table C3-18 Load/store register (unscaled offset) instructions

Mnemonic	Instruction	See
`LDUR`	Load register (unscaled offset)	*LDUR*
`LDURB`	Load byte (unscaled offset)	*LDURB*
`LDURSB`	Load signed byte (unscaled offset)	*LDURSB*
`LDURH`	Load halfword (unscaled offset)	*LDURH*
`LDURSH`	Load signed halfword (unscaled offset)	*LDURSH*
`LDURSW`	Load signed word (unscaled offset)	*LDURSW*
`STUR`	Store register (unscaled offset)	*STUR*
`STURB`	Store byte (unscaled offset)	*STURB*
`STURH`	Store halfword (unscaled offset)	*STURH*

C3.2.3 Load/store pair

The load/store pair instructions support the following addressing modes:

- Base plus a scaled 7-bit signed immediate offset.
- Pre-indexed by a scaled 7-bit signed immediate offset.
- Post-indexed by a scaled 7-bit signed immediate offset.

See also *Load/store addressing modes*.

If a Load Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.

- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

If a Load Pair instruction specifies writeback and one of the registers being loaded is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all of the loads using the specified addressing mode, and the base register becomes UNKNOWN. In addition, if an exception occurs during the instruction, the base address might be corrupted so that the instruction cannot be repeated.

If a Store Pair instruction performs a writeback and one of the registers being stored is also the base register, then behavior is CONSTRAINED UNPREDICTABLE and one of the following behaviors must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the stores of the registers indicated by the specified addressing mode, but the value stored for the base register is UNKNOWN.

Table C3-19 shows the load/store pair instructions.

Table C3-19 Load/store pair instructions

Mnemonic	Instruction	See
`LDP`	Load Pair	*LDP*
`LDPSW`	Load Pair signed words	*LDPSW*
`STP`	Store Pair	*STP*

C3.2.4 Load/store non-temporal pair

The load/store non-temporal pair instructions support only one addressing mode:

- Base plus a scaled 7-bit signed immediate offset.

See *Load/store addressing modes*.

The load/store non-temporal pair instructions provide a hint to the memory system that an access is non-temporal or streaming, and unlikely to be repeated in the near future. This means that data caching is not required. However, depending on the memory type, the instructions might permit memory reads to be preloaded and memory writes to be gathered to accelerate bulk memory transfers.

In addition, there is an exception to the usual memory ordering rules. If an address dependency exists between two memory reads, and a Load Non-temporal Pair instruction generated the second read, then in the absence of any other barrier mechanism to achieve order, the memory accesses can be observed in any order by the other observers within the shareability domain of the memory addresses being accessed.

If a Load Non-Temporal Pair instruction specifies the same register for the two registers that are being loaded, then behavior is CONSTRAINED UNPREDICTABLE and one of the following must occur:

- The instruction is treated as UNDEFINED.
- The instruction is treated as a NOP.
- The instruction performs all the loads using the specified addressing mode and the register that is loaded takes an UNKNOWN value.

C3.5 Data processing - immediate

This section describes the instruction groups for data processing with immediate operands. It contains the following subsections:

- *Arithmetic (immediate)*.
- *Integer minimum and maximum (immediate)*
- *Logical (immediate)*.
- *Move (wide immediate)*.
- *Move (immediate)*.
- *PC-relative address calculation*.
- *Bitfield move*.
- *Bitfield insert and extract*
- *Extract register*.
- *Shift (immediate)*.
- *Sign-extend and Zero-extend*.

For information about the encoding structure of the instructions in this instruction group, see *Data Processing -- Immediate*.

C3.5.1 Arithmetic (immediate)

The Arithmetic (immediate) instructions accept a 12-bit unsigned immediate value, optionally shifted left by 12 bits.

The Arithmetic (immediate) instructions that do not set Condition flags can read from and write to the current stack pointer. The flag setting instructions can read from the stack pointer, but they cannot write to it.

Table C3-65 shows the Arithmetic instructions with an immediate offset.

Table C3-65 Arithmetic instructions with an immediate

Mnemonic	Instruction	See
`ADD`	Add	*ADD (immediate)*
`ADDS`	Add and set flags	*ADDS (immediate)*
`SUB`	Subtract	*SUB (immediate)*
`SUBS`	Subtract and set flags	*SUBS (immediate)*
`CMP`	Compare	*CMP (immediate)*
`CMN`	Compare negative	*CMN (immediate)*

C3.5.2 Integer minimum and maximum (immediate)

The Integer maximum and minimum (immediate) instructions determine the maximum/minimum of the source register value and immediate.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-66 shows the Integer maximum and minimum (immediate) instructions.

Table C3-66 Integer maximum and minimum (immediate) instructions

Mnemonic	Instruction	See
`SMAX`	Signed Maximum (immediate)	*SMAX (immediate)*
`SMIN`	Signed Minimum (immediate)	*SMIN (immediate)*
`UMAX`	Unsigned Maximum (immediate)	*UMAX (immediate)*
`UMIN`	Unsigned Minimum (immediate)	*UMIN (immediate)*

C3.5.3 Logical (immediate)

The Logical (immediate) instructions accept a bitmask immediate value that is a 32-bit pattern or a 64-bit pattern viewed as a vector of identical elements of size e = 2, 4, 8, 16, 32 or, 64 bits. Each element contains the same sub-pattern, that is a single run of 1 to (e - 1) nonzero bits from bit 0 followed by zero bits, then rotated by 0 to (e - 1) bits. This mechanism can generate 5334 unique 64-bit patterns as 2667 pairs of pattern and their bitwise inverse.

Note

Values that consist of only zeros or only ones cannot be described in this way.

The Logical (immediate) instructions that do not set the Condition flags can write to the current stack pointer, for example to align the stack pointer in a function prologue.

Note

Apart from `ANDS` and its `TST` alias, Logical (immediate) instructions do not set the Condition flags. However, the final results of a bitwise operation can be tested by a `CBZ`, `CBNZ`, `TBZ`, or `TBNZ` conditional branch.

Table C3-67 shows the Logical immediate instructions.

Table C3-67 Logical immediate instructions

Mnemonic	Instruction	See
`AND`	Bitwise `AND`	*AND (immediate)*
`ANDS`	Bitwise `AND` and set flags	*ANDS (immediate)*
`EOR`	Bitwise exclusive `OR`	*EOR (immediate)*
`ORR`	Bitwise inclusive `OR`	*ORR (immediate)*
`TST`	Test bits	*TST (immediate)*

C3.5.4 Move (wide immediate)

The Move (wide immediate) instructions insert a 16-bit immediate, or inverted immediate, into a 16-bit aligned position in the destination register. The value of the other bits in the destination register depends on the variant used. The optional shift amount can be any multiple of 16 that is smaller than the register size.

Table C3-68 shows the Move (wide immediate) instructions.

Table C3-68 Move (wide immediate) instructions

Mnemonic	Instruction	See
`MOVZ`	Move wide with zero	*MOVZ*
`MOVN`	Move wide with NOT	*MOVN*
`MOVK`	Move wide with keep	*MOVK*

C3.5.5 Move (immediate)

The Move (immediate) instructions are aliases for a single `MOVZ`, `MOVN`, or `ORR` (immediate with zero register), instruction to load an immediate value into the destination register. An assembler must permit a signed or unsigned immediate, as long as its binary representation can be generated using one of these instructions, and an assembler error results if the immediate cannot be generated in this way. On disassembly, it is unspecified whether the immediate is output as a signed or an unsigned value.

If there is a choice between the `MOVZ`, `MOVN`, and `ORR` instruction to encode the immediate, then an assembler must prefer `MOVZ` to `MOVN`, and `MOVZ` or `MOVN` to `ORR`, to ensure reversability. A disassembler must output `ORR` (immediate with zero register) `MOVZ`, and `MOVN`, as a `MOV` mnemonic except that the underlying instruction must be used when:

- `ORR` has an immediate that can be generated by a `MOVZ` or `MOVN` instruction.
- A `MOVN` instruction has an immediate that can be encoded by `MOVZ`.
- `MOVZ #0` or `MOVN #0` have a shift amount other than `LSL #0`.

Table C3-69 shows the Move (immediate) instructions.

Table C3-69 Move (immediate) instructions

Mnemonic	Instruction	See
`MOV`	Move (inverted wide immediate)	*MOV (inverted wide immediate)*
	Move (wide immediate)	*MOV (wide immediate)*
	Move (bitmask immediate)	*MOV (bitmask immediate)*

C3.5.6 PC-relative address calculation

The `ADR` instruction adds a signed, 21-bit immediate to the value of the program counter that fetched this instruction, and then writes the result to a general-purpose register. This permits the calculation of any byte address within ±1MB of the current PC.

The `ADRP` instruction shifts a signed, 21-bit immediate left by 12 bits, adds it to the value of the program counter with the bottom 12 bits cleared to zero, and then writes the result to a general-purpose register. This permits the calculation of the address at a 4KB aligned memory region. In conjunction with an `ADD` (immediate) instruction, or a load/store instruction with a 12-bit immediate offset, this allows for the calculation of, or access to, any address within ±4GB of the current PC.

Note

The term *page* used in the `ADRP` description is short-hand for the 4KB memory region, and is not related to the virtual memory translation granule size.

Table C3-70 shows the instructions used for PC-relative address calculations are as follows:

Table C3-70 PC-relative address calculation instructions

Mnemonic	Instruction	See
`ADRP`	Compute address of 4KB page at a PC-relative offset	*ADRP*
`ADR`	Compute address of label at a PC-relative offset.	*ADR*

C3.5.7 Bitfield move

The Bitfield move instructions copy a field of constant width from bit 0 in the source register to a constant bit position in the destination register, or from a constant bit position in the source register to bit 0 in the destination register. The remaining bits in the destination register are set as follows:

- For `BFM`, the remaining bits are unchanged.
- For `UBFM` the lower bits, if any, and upper bits, if any, are set to zero.
- For `SBFM`, the lower bits, if any, are set to zero, and the upper bits, if any, are set to a copy of the most-significant bit in the copied field.

Table C3-71 shows the Bitfield move instructions.

Table C3-71 Bitfield move instructions

Mnemonic	Instruction	See
`BFM`	Bitfield move	*BFM*
`SBFM`	Signed bitfield move	*SBFM*
`UBFM`	Unsigned bitfield move (32-bit)	*UBFM*

C3.5.8 Bitfield insert and extract

The Bitfield insert and extract instructions are implemented as aliases of the Bitfield move instructions. Table C3-72 shows the Bitfield insert and extract aliases.

Table C3-72 Bitfield insert and extract instructions

Mnemonic	Instruction	See
`BFC`	Bitfield clear	*BFC*
`BFI`	Bitfield insert	*BFI*
`BFXIL`	Bitfield extract and insert low	*BFXIL*
`SBFIZ`	Signed bitfield insert in zero	*SBFIZ*
`SBFX`	Signed bitfield extract	*SBFX*
`UBFIZ`	Unsigned bitfield insert in zero	*UBFIZ*
`UBFX`	Unsigned bitfield extract	*UBFX*

C3.5.9 Extract register

Depending on the register width of the operands, the Extract register instruction copies a 32-bit or 64-bit field from a constant bit position within a double-width value formed by the concatenation of a pair of source registers to a destination register.

Table C3-73 shows the Extract (immediate) instructions.

Table C3-73 Extract register instructions

Mnemonic	Instruction	See
EXTR	Extract register from pair	*EXTR*

C3.5.10 Shift (immediate)

Shifts and rotates by a constant amount are implemented as aliases of the Bitfield move or Extract register instructions. The shift or rotate amount must be in the range 0 to one less than the register width of the instruction, inclusive.

Table C3-74 shows the aliases that can be used as immediate shift and rotate instructions.

Table C3-74 Aliases for immediate shift and rotate instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	*ASR (immediate)*
LSL	Logical shift left	*LSL (immediate)*
LSR	Logical shift right	*LSR (immediate)*
ROR	Rotate right	*ROR (immediate)*

C3.5.11 Sign-extend and Zero-extend

The Sign-extend and Zero-extend instructions are implemented as aliases of the Bitfield move instructions.

Table C3-75 shows the aliases that can be used as zero-extend and sign-extend instructions.

Table C3-75 Zero-extend and sign-extend instructions

Mnemonic	Instruction	See
SXTB	Sign-extend byte	*SXTB*
SXTH	Sign-extend halfword	*SXTH*
SXTW	Sign-extend word	*SXTW*
UXTB	Unsigned extend byte	*UXTB*
UXTH	Unsigned extend halfword	*UXTH*

C3.6 Data processing - register

This section describes the instruction groups for data processing with all register operands. It contains the following subsections:

- *Arithmetic (shifted register)*.
- *Arithmetic (extended register)*.
- *Arithmetic with carry*.
- *Integer maximum and minimum (register)*
- *Flag manipulation instructions*.
- *Logical (shifted register)*.
- *Move (register)*.
- *Absolute value*
- *Shift (register)*.
- *Multiply and divide*.
- *CRC32*.
- *Bit operation*.
- *Conditional select*.
- *Conditional comparison*.

For information about the encoding structure of the instructions in this instruction group, see *Data Processing -- Register*.

C3.6.1 Arithmetic (shifted register)

The Arithmetic (shifted register) instructions apply an optional shift operator to the second source register value before performing the arithmetic operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, and `LSR` accept an immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

The current stack pointer, SP or WSP, cannot be used with this class of instructions. See *Arithmetic (extended register)* for arithmetic instructions that can operate on the current stack pointer.

Table C3-76 shows the Arithmetic (shifted register) instructions.

Table C3-76 Arithmetic (shifted register) instructions

Mnemonic	Instruction	See
`ADD`	Add	*ADD (shifted register)*
`ADDS`	Add and set flags	*ADDS (shifted register)*
`SUB`	Subtract	*SUB (shifted register)*
`SUBS`	Subtract and set flags	*SUBS (shifted register)*
`CMN`	Compare negative	*CMN (shifted register)*

Table C3-76 Arithmetic (shifted register) instructions (continued)

Mnemonic	Instruction	See
CMP	Compare	*CMP (shifted register)*
NEG	Negate	*NEG (shifted register)*
NEGS	Negate and set flags	*NEGS*

C3.6.2 Arithmetic (extended register)

The extended register instructions provide an optional sign-extension or zero-extension of a portion of the second source register value, followed by an optional left shift by a constant amount of 1-4, inclusive.

The extended shift is described by the mandatory extend operator SXTB, SXTH, SXTW, UXTB, UXTH, or UXTW. This is followed by an optional left shift amount. If the shift amount is not specified, the default shift amount is zero. A disassembler must not output a shift amount of zero.

For 64-bit instruction forms, the additional operators UXTX and SXTX use all 64 bits of the second source register with an optional shift. In that case, Arm recommends UXTX as the operator. If and only if at least one register is SP, Arm recommends use of the LSL operator name, rather than UXTX, and when the shift amount is also zero then both the operator and the shift amount can be omitted. UXTW and SXTW both use all 32 bits of the second source register with an optional shift. In that case Arm recommends UXTW as the operator. If and only if at least one register is WSP, Arm recommends use of the LSL operator name, rather than UXTW, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

For 32-bit instruction forms, the operators UXTW and SXTW both use all 32 bits of the second source register with an optional shift. In that case, Arm recommends UXTW as the operator. If and only if at least one register is WSP, Arm recommends use of the LSL operator name, rather than UXTW, and when the shift amount is also zero then both the operator and the shift amount can be omitted.

The non-flag setting variants of the extended register instruction permit the use of the current stack pointer as either the destination register and the first source register. The flag setting variants only permit the stack pointer to be used as the first source register.

In the 64-bit form of these instructions, the final register operand is written as Wm for all except the UXTX/LSL and SXTX extend operators. For example:

```
CMP X4, W5, SXTW
ADD X1, X2, W3, UXTB #2
SUB SP, SP, X1              // SUB SP, SP, X1, UXTX #0
```

Table C3-77 shows the Arithmetic (extended register) instructions.

Table C3-77 Arithmetic (extended register) instructions

Mnemonic	Instruction	See
ADD	Add	*ADD (extended register)*
ADDS	Add and set flags	*ADDS (extended register)*
SUB	Subtract	*SUB (extended register)*
SUBS	Subtract and set flags	*SUBS (extended register)*
CMN	Compare negative	*CMN (extended register)*
CMP	Compare	*CMP (extended register)*

C3.6.3 Arithmetic with carry

The Arithmetic with carry instructions accept two source registers, with the carry flag as an additional input to the calculation. They do not support shifting of the second source register.

Table C3-78 shows the Arithmetic with carry instructions

Table C3-78 Arithmetic with carry instructions

Mnemonic	Instruction	See
`ADC`	Add with carry	*ADC*
`ADCS`	Add with carry and set flags	*ADCS*
`SBC`	Subtract with carry	*SBC*
`SBCS`	Subtract with carry and set flags	*SBCS*
`NGC`	Negate with carry	*NGC*
`NGCS`	Negate with carry and set flags	*NGCS*

C3.6.4 Integer maximum and minimum (register)

The Integer maximum and minimum (register) instructions determine the maximum/minimum of the two source register values.

These instructions are only present when FEAT_CSSC is implemented.

Table C3-79 shows the Integer maximum and minimum (register) instructions.

Table C3-79 Integer maximum and minimum (register) instructions

Mnemonic	Instruction	See
`SMAX`	Signed Maximum (register)	*SMAX (register)*
`SMIN`	Signed Minimum (register)	*SMIN (register)*
`UMAX`	Unsigned Maximum (register)	*UMAX (register)*
`UMIN`	Unsigned Minimum (register)	*UMIN (register)*

C3.6.5 Flag manipulation instructions

The Flag manipulation instructions set the value of the NZCV condition flags directly.

The instructions `SETF8` and `SETF16` accept one source register and set the NZV condition flags based on the value of the input register. The instruction `RMIF` accepts one source register and two immediate values, rotating the first source register using the first immediate value and setting the NZCV condition flags masked by the second immediate value.

The instructions `XAFLAG` and `AXFLAG` convert PSTATE condition flags between the FCMP instruction format and an alternative format. See Table C6-1 for more information.

Table C3-80 shows the Flag manipulation instructions.

Table C3-80 Flag manipulation instructions

Mnemonic	Instruction	See
`AXFLAG`	Convert from FCMP comparison format to the alternative format	*AXFLAG*
`CFINV`	Invert value of the PSTATE.C bit	*CFINV*
`RMIF`	Rotate, mask insert flags	*RMIF*
`SETF8`	Evaluation of 8-bit flags	*SETF8, SETF16*
`SETF16`	Evaluation of 16-bit flags	*SETF8, SETF16*
`XAFLAG`	Convert from alternative format to FCMP comparison format	*XAFLAG*

C3.6.6 Logical (shifted register)

The Logical (shifted register) instructions apply an optional shift operator to the second source register value before performing the main operation. The register width of the instruction controls whether the new bits are fed into the intermediate result on a right shift or rotate at bit[63] or bit[31].

The shift operators `LSL`, `ASR`, `LSR`, and `ROR` accept a constant immediate shift amount in the range 0 to one less than the register width of the instruction, inclusive.

Omitting the shift operator and amount implies `LSL #0`, which means that there is no shift. A disassembler must not output `LSL #0`. However, a disassembler must output all other shifts by zero.

Note

Apart from `ANDS,` `TST`, and `BICS`, the logical instructions do not set the Condition flags, but the final result of a bit operation can usually directly control a `CBZ`, `CBNZ`, `TBZ`, or `TBNZ` conditional branch.

Table C3-81 shows the Logical (shifted register) instructions.

Table C3-81 Logical (shifted register) instructions

Mnemonic	Instruction	See
`AND`	Bitwise AND	*AND (shifted register)*
`ANDS`	Bitwise AND and set flags	*ANDS (shifted register)*
`BIC`	Bitwise bit clear	*BIC (shifted register)*
`BICS`	Bitwise bit clear and set flags	*BICS (shifted register)*
`EON`	Bitwise exclusive-OR NOT	*EON (shifted register)*
`EOR`	Bitwise exclusive-OR	*EOR (shifted register)*
`ORR`	Bitwise inclusive OR	*ORR (shifted register)*
`MVN`	Bitwise NOT	*MVN*
`ORN`	Bitwise inclusive OR NOT	*ORN (shifted register)*
`TST`	Test bits	*TST (shifted register)*

C3.6.7 Move (register)

The Move (register) instructions are aliases for other data processing instructions. They copy a value from a general-purpose register to another general-purpose register or the current stack pointer, or from the current stack pointer to a general-purpose register.

Table C3-82 MOV register instructions

Mnemonic	Instruction	See
MOV	Move register	*MOV (register)*
	Move register to SP or move SP to register	*MOV (to/from SP)*

C3.6.8 Absolute value

The Absolute value instruction is only present when FEAT_CSSC is implemented.

Table C3-83 shows the Absolute value instruction.

Table C3-83 Absolute value instruction

Mnemonic	Instruction	See
ABS	Absolute value	*ABS*

C3.6.9 Shift (register)

In the Shift (register) instructions, the shift amount is the positive value in the second source register modulo the register size. The register width of the instruction controls whether the new bits are fed into the result on a right shift or rotate at bit[63] or bit[31].

Table C3-84 shows the Shift (register) instructions.

Table C3-84 Shift (register) instructions

Mnemonic	Instruction	See
ASRV	Arithmetic shift right variable	*ASRV*
LSLV	Logical shift left variable	*LSLV*
LSRV	Logical shift right variable	*LSRV*
RORV	Rotate right variable	*RORV*

However, the Shift (register) instructions have a preferred set of aliases that match the shift immediate aliases described in *Shift (immediate)*.

Table C3-85 shows the aliases for Shift (register) instructions.

Table C3-85 Aliases for Variable shift instructions

Mnemonic	Instruction	See
ASR	Arithmetic shift right	*ASR (register)*

Table C3-85 Aliases for Variable shift instructions (continued)

Mnemonic	Instruction	See
`LSL`	Logical shift left	*LSL (register)*
`LSR`	Logical shift right	*LSR (register)*
`ROR`	Rotate right	*ROR (register)*

C3.6.10 Multiply and divide

This section describes the instructions used for integer multiplication and division. It contains the following subsections:

- *Multiply*.
- *Divide*.

C3.6.10.1 Multiply

The Multiply instructions write to a single 32-bit or 64-bit destination register, and are built around the fundamental four operand multiply-add and multiply-subtract operation, together with 32-bit to 64-bit widening variants. A 64-bit to 128-bit widening multiple can be constructed with two instructions, using `SMULH` or `UMULH` to generate the upper 64 bits. Table C3-86 shows the Multiply instructions.

Table C3-86 Multiply integer instructions

Mnemonic	Instruction	See
`MADD`	Multiply-add	*MADD*
`MSUB`	Multiply-subtract	*MSUB*
`MNEG`	Multiply-negate	*MNEG*
`MUL`	Multiply	*MUL*
`SMADDL`	Signed multiply-add long	*SMADDL*
`SMSUBL`	Signed multiply-subtract long	*SMSUBL*
`SMNEGL`	Signed multiply-negate long	*SMNEGL*
`SMULL`	Signed multiply long	*SMULL*
`SMULH`	Signed multiply high	*SMULH*
`UMADDL`	Unsigned multiply-add long	*UMADDL*
`UMSUBL`	Unsigned multiply-subtract long	*UMSUBL*
`UMNEGL`	Unsigned multiply-negate long	*UMNEGL*
`UMULL`	Unsigned multiply long	*UMULL*
`UMULH`	Unsigned multiply high	*UMULH*

C3.6.10.2 Divide

The Divide instructions compute the quotient of a division, rounded towards zero. The remainder can then be computed as (numerator - (quotient × denominator)), using the `MSUB` instruction.

If a signed integer division (`INT_MIN / -1`) is performed where `INT_MIN` is the most negative integer value representable in the selected register size, then the result overflows the signed integer range. No indication of this overflow is produced and the result that is written to the destination register is `INT_MIN`.

A division by zero results in a zero being written to the destination register, without any indication that the division by zero occurred.

Table C3-87 shows the Divide instructions.

Table C3-87 Divide instructions

Mnemonic	Instruction	See
`SDIV`	Signed divide	*SDIV*
`UDIV`	Unsigned divide	*UDIV*

C3.6.11 CRC32

The `CRC32` instructions operate on the general-purpose register file to update a 32-bit CRC value from an input value comprising 1, 2, 4, or 8 bytes. There are two different classes of `CRC` instructions, `CRC32`, and `CRC32C`, that support two commonly used 32-bit polynomials, known as CRC-32 and CRC-32C.

To fit with common usage, the bit order of the values is reversed as part of the operation.

When bits[19:16] of ID_AA64ISAR0_EL1 are set to `0b0001`, the `CRC` instructions are implemented.

These instructions are optional in an Armv8.0 implementation.

All implementations of Armv8.1 architecture and later are required to implement the `CRC32` instructions.

Table C3-88 shows the `CRC` instructions.

Table C3-88 CRC32 instructions

Mnemonic	Instruction	See
`CRC32B`	CRC-32 sum from byte	*CRC32B, CRC32H, CRC32W, CRC32X*
`CRC32H`	CRC-32 sum from halfword	*CRC32B, CRC32H, CRC32W, CRC32X*
`CRC32W`	CRC-32 sum from word	*CRC32B, CRC32H, CRC32W, CRC32X*
`CRC32X`	CRC-32 sum from doubleword	*CRC32B, CRC32H, CRC32W, CRC32X*
`CRC32CB`	CRC-32C sum from byte	*CRC32CB, CRC32CH, CRC32CW, CRC32CX*
`CRC32CH`	CRC-32C sum from halfword	*CRC32CB, CRC32CH, CRC32CW, CRC32CX*
`CRC32CW`	CRC-32C sum from word	*CRC32CB, CRC32CH, CRC32CW, CRC32CX*
`CRC32CX`	CRC-32C sum from doubleword	*CRC32CB, CRC32CH, CRC32CW, CRC32CX*

C3.6.12 Bit operation

The `CNT` and `CTZ` instructions are only present when FEAT_CSSC is implemented.

Table C3-89 shows the Bit operation instructions.

Table C3-89 Bit operation instructions

Mnemonic	Instruction	See
`CLS`	Count leading sign bits	*CLS*
`CLZ`	Count leading zero bits	*CLZ*
`CNT`	Count bits	*CNT*
`CTZ`	Count trailing zero bits	*CTZ*
`RBIT`	Reverse bit order	*RBIT*
`REV`	Reverse bytes in register	*REV*
`REV16`	Reverse bytes in halfwords	*REV16*
`REV32`	Reverse bytes in words	*REV32*
`REV64`	Reverse bytes in register	*REV64*

C3.6.13 Conditional select

The Conditional select instructions select between the first or second source register, depending on the current state of the Condition flags. When the named condition is true, the first source register is selected and its value is copied without modification to the destination register. When the condition is false the second source register is selected and its value might be optionally inverted, negated, or incremented by one, before writing to the destination register.

Other useful conditional set and conditional unary operations are implemented as aliases of the four Conditional select instructions.

Table C3-90 shows the Conditional select instructions.

Table C3-90 Conditional select instructions

Mnemonic	Instruction	See
`CSEL`	Conditional select	*CSEL*
`CSINC`	Conditional select increment	*CSINC*
`CSINV`	Conditional select inversion	*CSINV*
`CSNEG`	Conditional select negation	*CSNEG*
`CSET`	Conditional set	*CSET*
`CSETM`	Conditional set mask	*CSETM*
`CINC`	Conditional increment	*CINC*
`CINV`	Conditional invert	*CINV*
`CNEG`	Conditional negate	*CNEG*

C3.6.14 Conditional comparison

The Conditional comparison instructions provide a conditional select for the NZCV Condition flags, setting the flags to the result of an arithmetic comparison of its two source register values if the named input condition is true, or to an immediate value if the input condition is false. There are register and immediate forms. The immediate form compares the source register to a small 5-bit unsigned value.

Table C3-91 shows the Conditional comparison instructions.

Table C3-91 Conditional comparison instructions

Mnemonic	Instruction	See
`CCMN`	Conditional compare negative (register)	*CCMN (register)*
`CCMN`	Conditional compare negative (immediate)	*CCMN (immediate)*
`CCMP`	Conditional compare (register)	*CCMP (register)*
`CCMP`	Conditional compare (immediate)	*CCMP (immediate)*