Cortex®-A57 Software Optimisation Guide

Date of Issue: January 19, 2015

Cortex®-A57
Software Optimisation Guide

Release Information

The following changes have been made to this Software Optimisation Guide.

Change History

Date	Issue	Confidentiality	Change
19 January 2015		Non-Confidential	First release

Confidentiality Status

This document is Non-Confidential. This document has no restriction on distribution.

Product Status

The information in this document is final, that is for a developed product .

Web Address

http://www.arm.com

Contents

1 ABOUT THIS DOCUMENT

1.1 References

This document refers to the following documents.

Title	Location
ARM Cortex-A57 MPCore Processor Technical Reference Manual	Infocenter.arm.com

1.2 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ALU	Arithmetic/Logical Unit
ASIMD	Advanced SIMD
μop	Micro-operation
VFP	Vector Floating Point

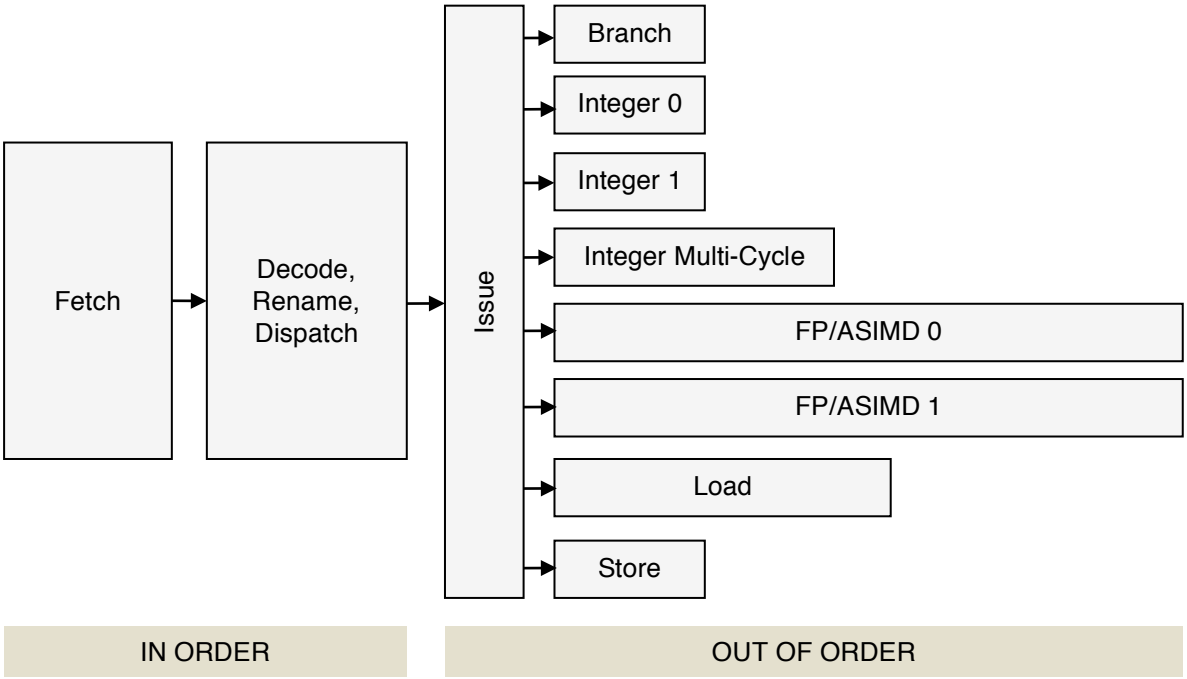
1.3 Document scope

This document provides high-level information about the Cortex®-A57 processor pipeline, instruction performance characteristics, and special performance considerations. This information is intended to aid those who are optimizing software and compilers for the Cortex®-A57 processor. For a more complete description of the Cortex®-A57 processor refer to the *ARM Cortex®-A57 MPCore Processor Technical Reference Manual*, available at infocenter.arm.com.

2 INTRODUCTION

2.1 Pipeline overview

The following diagram describes the high-level Cortex®-A57 instruction processing pipeline. Instructions are first fetched, then decoded into internal micro-operations (*µ*ops). From there, the *µ*ops proceed through register renaming and dispatch stages. After they are dispatched, *µ*ops wait for their operands and issue out-of-order to one of eight execution pipelines. Each execution pipeline can accept and complete one *µ*op per cycle.



The execution pipelines support different types of operations, as shown below.

Pipeline (mnemonic)	Supported functionality
Branch (B)	Branch *µ*ops
Integer 0/1 (I)	Integer ALU *µ*ops
Multi-cycle (M)	Integer shift-ALU, multiply, divide, CRC and sum-of-absolute-differences *µ*ops
Load (L)	Load and register transfer *µ*ops
Store (S)	Store and special memory *µ*ops
FP/ASIMD-0 (F0)	ASIMD ALU, ASIMD misc, FP misc, FP add, FP multiply, ASIMD integer multiply, FP divide *µ*ops, crypto *µ*ops

FP/ASIMD-1 (F1)	ASIMD ALU, ASIMD misc, FP misc, FP add, FP multiply, ASIMD shift μops

3 INSTRUCTION CHARACTERISTICS

3.1 Instruction tables

This chapter describes high-level performance characteristics for most ARMv8 A32, T32 and A64 instructions. It includes a series of tables to summarize the effective execution latency and throughput, pipelines utilized, and special behaviors associated with each group of instructions. Utilized pipelines correspond to the execution pipelines described in chapter 2.

In the following tables:

- **Exec Latency** is defined as the minimum latency seen by an operation dependent on an instruction in the described group.
- **Execution Throughput** is defined as the maximum throughput (in instructions / cycle) of the specified instruction group that can be achieved in the entirety of the Cortex®-A57 microarchitecture.

3.2 Branch instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Branch, immed	B	1	1	B	
Branch, register	BX	1	1	B	
Branch and link, immed	BL, BLX	1	1	I0/I1, B	
Branch and link, register != LR	BLX	2	1	I0/I1, B	
Branch and link, register = LR	BLX	3	1	I0/I1, B	
Compare and branch	CBZ, CBNZ	1	1	B	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Branch, immed	B	1	1	B	
Branch, register	BR, RET	1	1	B	
Branch and link, immed	BL	1	1	I0/I1, B	
Branch and link, register != LR	BLR	2	1	I0/I1, B	
Branch and link, register = LR	BLR	3	1	I0/I1, B	
Compare and branch	CBZ, CBNZ, TBZ, TBNZ	1	1	B	

3.3 Arithmetic and logical instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ALU, basic	ADD{S}, ADC{S}, ADR, AND{S}, BIC{S}, CMN, CMP, EOR{S}, ORN{S}, ORR{S}, RSB{S}, RSC{S}, SUB{S}, SBC{S}, TEQ, TST	1	2	I0/I1	
ALU, shift by immed	(same as above)	2	1	M	
ALU, shift by register, unconditional	(same as above)	2	1	M	
ALU, shift by register, conditional	(same as above)	2	1	I0/I1	
ALU, branch forms		+2	1	+B	1

Notes:

1. Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch *μ*op is required. This adds two cycles to the latency.

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ALU, basic	ADD{S}, ADC{S}, AND{S}, BIC{S}, EON, EOR, ORN, ORR, SUB{S}, SBC{S}	1	2	I0/I1	
ALU, extend and/or shift	ADD{S}, AND{S}, BIC{S}, EON, EOR, ORN, ORR, SUB{S}	2	1	M	
Conditional compare	CCMN, CCMP	1	2	I0/I1	
Conditional select	CSEL, CSINC, CSINV, CSNEG	1	2	I0/I1	

3.4 Move and shift instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Move, basic	MOV{S}, MOVW, MVN{S}	1	2	I0/I1	1
Move, shift by immed, no setflags	ASR, LSL, LSR, ROR, RRX, MVN	1	2	I0/I1	
Move, shift by immed, setflags	ASRS, LSLS, LSRS, RORS, RRXS, MVNS	2	1	M	
Move, shift by register, no setflags, unconditional	ASR, LSL, LSR, ROR, RRX, MVN	1	2	I0/I1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Move, shift by register, no setflags, conditional	ASR, LSL, LSR, ROR, RRX, MVN	2	1	I0/I1	
Move, shift by register, setflags, unconditional	ASRS, LSLS, LSRS, RORS, RRXS, MVNS	2	1	M	
Move, shift by register, setflags, conditional	ASRS, LSLS, LSRS, RORS, RRXS, MVNS	2	1	I0/I1	
Move, top	MOVT	2/1	1 on r0px, 2 on r1px	M/I	2
(Move, branch forms)		+2	1	+B	3

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Address generation	ADR, ADRP	1	2	I0/I1	4
Move immed	MOVN, MOVK, MOVZ	1	2	I0/I1	1
Variable shift	ASRV, LSLV, LSRV, RORV	1	2	I0/I1	

Note:

1. In the Cortex®-A57 processor r1p0 and later revisions, sequential MOVW/MOVT (AArch32) instruction pairs and certain MOVZ/MOVK, MOVK/MOVK (AArch64) instruction pairs can be executed with one-cycle execute latency and four-instruction/cycle execution throughput in I0/I1. See Section 4.13 for more details on the instruction pairs that can be merged.
2. MOVT is implemented as a single-cycle *μ*op in the Cortex®-A57 processor r1p0 and later. A latency given as “N/M” implies a latency of N cycles in r0pX revisions and M cycles in r1p0 and later revisions. A similar notation applies to utilized pipelines.
3. Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch *μ*op is required. This adds two cycles to the latency.
4. In the Cortex®-A57 processor r1p0 and later revisions, sequential ADRP/ADD instruction pairs can be executed with one-cycle execute latency and four instruction/cycle execution throughput in I0/I1. See Section 4.14 for more information.

3.5 Divide and multiply instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Divide	SDIV, UDIV	4 - 20	1/20 - 1/4	M	1
Multiply	MUL, SMULBB, SMULBT, SMULTB, SMULTT, SMULWB, SMULWT, SMMUL{R}, SMUAD{X}, SMUSD{X}	3	1	M	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Multiply accumulate	MLA, MLS, SMLABB, SMLABT, SMLATB, SMLATT, SMLAWB, SMLAWT, SMLAD{X}, SMLSD{X}, SMMLA{R}, SMMLS{R}	3 (1)	1	M	2
Multiply accumulate long	SMLAL, SMLALBB, SMLALBT, SMLALTB, SMLALTT, SMLALD{X}, SMLSLD{X}, UMAAL, UMLAL	4 (2)	1/2	M	2, 3
Multiply long	SMULL, UMULL	4	1/2	M	3
(Multiply, setflags forms)		+1	(Same as above)	+I0/I1	4

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized pipelines	Notes
Divide, W-form	SDIV, UDIV	4-20	1/20 – 1/4	M	1
Divide, X-form	SDIV, UDIV	4-36	1/36 - 1/4	M	1
Multiply accumulate, W-form	MADD, MSUB	3 (1)	1	M	2
Multiply accumulate, X-form	MADD, MSUB	5 (3)	1/3	M	2,5
Multiply accumulate long	SMADDL, SMSUBL, UMADDL, UMSUBL	3 (1)	1	M	2
Multiply high	SMULH, UMULH	6 [3]	1/4	M	6

Note:

1. Integer divides are performed using an iterative algorithm and block any subsequent divide operations until complete. Early termination is possible, depending upon the data values.
2. Multiply-accumulate pipelines support late-forwarding of accumulate operands from similar μops, allowing a typical sequence of multiply-accumulate μops to issue one every N cycles (accumulate latency N is shown in parentheses).
3. Long-form multiplies (which produce two result registers) stall the multiplier pipeline for one extra cycle.
4. Multiplies that set the condition flags require an additional integer μop.
5. X-form multiply accumulates stall the multiplier pipeline for two extra cycles.

6. Multiply high operations stall the multiplier pipeline for N extra cycles before any other type M *µ*op can be issued to that pipeline, with N shown in parentheses.

3.6 Saturating and parallel arithmetic instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized pipelines	Notes
Parallel arith, unconditional	SADD16, SADD8, SSUB16, SSUB8, UADD16, UADD8, USUB16, USUB8	2	1	M	
Parallel arith, conditional	SADD16, SADD8, SSUB16, SSUB8, UADD16, UADD8, USUB16, USUB8	2 (4)	1/2	M, I0/I1	1
Parallel arith with exchange, unconditional	SASX, SSAX, UASX, USAX	3	1	I0/I1, M	
Parallel arith with exchange, conditional	SASX, SSAX, UASX, USAX	3 (5)	1/2	I0/I1, M	1
Parallel halving arith	SHADD16, SHADD8, SHSUB16, SHSUB8, UHADD16, UHADD8, UHSUB16, UHSUB8	2	1	M	
Parallel halving arith with exchange	SHASX, SHSAX, UHASX, UHSAX	3	1	I0/I1, M	
Parallel saturating arith	QADD16, QADD8, QSUB16, QSUB8, UQADD16, UQADD8, UQSUB16, UQSUB8	2	1	M	
Parallel saturating arith with exchange	QASX, QSAX, UQASX, UQSAX	3	1	I0/I1, M	
Saturate	SSAT, SSAT16, USAT, USAT16	2	1	M	
Saturating arith	QADD, QSUB	2	1	M	
Saturating doubling arith	QDADD, QDSUB	3	1	I0/I1, M	

Note:

1. Conditional GE-setting instructions require three extra *µ*ops and two additional cycles to conditionally update the GE field (GE latency is shown in parentheses).

3.7 Miscellaneous data-processing instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Bit field extract	SBFX, UBFX	1	2	I0/I1	
Bit field insert/clear	BFI, BFC	2	1	M	
Count leading zeros	CLZ	1	2	I0/I1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Pack halfword	PKH	2	1	M	
Reverse bits/bytes	RBIT, REV, REV16, REVSH	1	2	I0/I1	
Select bytes, unconditional	SEL	1	2	I0/I1	
Select bytes, conditional	SEL	2	1	I0/I1	
Sign/zero extend, normal	SXTB, SXTH, UXTB, UXTH	1	2	I0/I1	
Sign/zero extend, parallel	SXTB16, UXTB16	2	1	M	
Sign/zero extend and add, normal	SXTAB, SXTAH,UXTAB, UXTAH	2	1	M	
Sign/zero extend and add, parallel	SXTAB16, UXTAB16	4	1/2	M	
Sum of absolute differences	USAD8, USADA8	3	1	M	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Bitfield extract, one reg	EXTR	1	2	I0/I1	
Bitfield extract, two regs	EXTR	3	1	I0/I1, M	
Bitfield move, basic	SBFM, UBFM	1	2	I0/I1	
Bitfield move, insert	BFM	2	1	M	
Count leading	CLS, CLZ	1	2	I0/I1	
Reverse bits/bytes	RBIT, REV, REV16, REV32	1	2	I0/I1	

NOP instructions are resolved at dispatch and have a maximum execution throughput of one00 per cycle. Only one NOP can be dispatched per cycle.

3.8 Load instructions

The latencies shown in the following table assume the memory access hits in the Level 1 Data Cache.

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load, immed offset	LDR{T}, LDRB{T}, LDRD, LDRH{T}, LDRSB{T}, LDRSH{T}	4	1	L	
Load, register offset, plus	LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH	4	1	L	
Load, register offset, minus	LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH	5	1	I0/I1, L	
Load, scaled register offset, plus LSL2	LDR, LDRB	4	1	L	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load, scaled register offset, other	LDR, LDRB, LDRH, LDRSB, LDRSH	5	1	I0/I1, L	
Load, immed pre-indexed	LDR, LDRB, LDRD, LDRH, LDRSB, LDRSH	4 (1)	1	L, I0/I1	1
Load, register pre-indexed	LDR, LDRB, LDRH, LDRSB, LDRSH	4 (2)	1	I0/I1, L	1
Load, register pre-indexed	LDRD	5 (2)	1	I0/I1, L	1
Load, scaled register pre-indexed, plus LSL2	LDR, LDRB	4 (2)	1	I0/I1, L	1
Load, scaled register pre-indexed, other	LDR, LDRB	5 (2)	1	I0/I1, L	1
Load, immed post-indexed	LDR{T}, LDRB{T}, LDRD, LDRH{T}, LDRSB{T}, LDRSH{T}	4 (1)	1	L, I0/I1	1
Load, register post-indexed	LDR{T}, LDRB{T}, LDRD, LDRH{T}, LDRSB{T}, LDRSH{T}	4 (2)	1	I0/I1, L	1
Load, scaled register post-indexed	LDR, LDRB	4 (2)	1	I0/I1, L	1
Load, scaled register post-indexed	LDRT, LDRBT	4 (3)	1	I0/I1, L, M	1
Preload, immed offset	PLD, PLDW	4	1	L	
Preload, register offset, plus	PLD, PLDW	4	1	L	
Preload, register offset, minus	PLD, PLDW	5	1	I0/I1, L	
Preload, scaled register offset, plus LSL2	PLD, PLDW	4	1	L	
Preload, scaled register offset, other	PLD, PLDW	5	1	I0/I1, L	
Load multiple, no writeback, base reg not in list	LDMIA, LDMIB, LDMDA, LDMDB	3 + N	1/N	L	2
Load multiple, no writeback, base reg in list	LDMIA, LDMIB, LDMDA, LDMDB	4 + N	1/N	I0/I1, L	2
Load multiple, writeback	LDMIA, LDMIB, LDMDA, LDMDB, POP	3 + N	1/N	L, I0/I1	1, 2
Load, branch forms with addressing mode as register post-indexed (scaled or unscaled) or scaled, register pre-indexed, plus, LSL2	LDR	4(2)	1	L, M	1

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load, branch forms with addressing mode register pre-indexed	LDR	4(1)	1	L, I0/I1	1
(Load, branch forms)		+2		+B	3

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load register, literal	LDR, LDRSW, PRFM	4	1	L	
Load register, unscaled immed	LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, PRFUM	4	1	L	
Load register, immed post-index	LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW	4 (1)	1	L, I0/I1	1
Load register, immed pre-index	LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW	4 (1)	1	L, I0/I1	1
Load register, immed unprivileged	LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW	4	1	L	
Load register, unsigned immed	LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM	4	1	L	
Load register, register offset, basic	LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM	4	1	L	
Load register, register offset, scale by 4/8	LDR, LDRSW, PRFM	4	1	L	
Load register, register offset, scale by 2	LDRH, LDRSH	5	1	I0/I1, L	
Load register, register offset, extend	LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, PRFM	4	1	L	
Load register, register offset, extend, scale by 4/8	LDR, LDRSW, PRFM	4	1	L	
Load register, register offset, extend, scale by 2	LDRH, LDRSH	5	1	I0/I1, L	
Load pair, immed offset, normal	LDP, LDNP	4	1	L	
Load pair, immed offset, signed words, base != SP	LDPSW	5	1/2	I0/I1, L	
Load pair, immed offset, signed words, base = SP	LDPSW	5	1/2	L	
Load pair, immed post-index, normal	LDP	4 (1)	1	L, I0/I1	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load pair, immed post-index, signed words	LDPSW	5 (1)	1/2	L, I0/I1	1
Load pair, immed pre-index, normal	LDP	4 (1)	1	L, I0/I1	1
Load pair, immed pre-index, signed words	LDPSW	5 (1)	1/2	L, I0/I1	1

Note:

1. Base register updates are typically completed in parallel with the load operation and with shorter latency (update latency is shown in parentheses).
2. For load multiple instructions, N=floor((num_regs+1)/2).
3. Branch forms are possible when the instruction destination register is the PC. For those cases, an additional branch *µ*op is required. This adds two cycles to the latency.

3.9 Store instructions

The following table describes performance characteristics for standard store instructions. Stores *µ*ops can issue after their address operands become available and do not need to wait for data operands. After they are executed, stores are buffered and committed in the background.

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store, immed offset	STR{T}, STRB{T}, STRD, STRH{T}	1	1	S	
Store, register offset, plus	STR, STRB, STRD, STRH	1	1	S	
Store, register offset, minus	STR, STRB, STRD, STRH	3	1	I0/I1, S	
Store, scaled register offset, plus LSL2	STR, STRB	1	1	S	
Store, scaled register offset, other	STR, STRB	3	1	I0/I1, S	
Store, immed pre-indexed	STR, STRB, STRD, STRH	1 (1)	1	S, I0/I1	1
Store, register pre-indexed, plus	STR, STRB, STRD, STRH	1 (1)	1	S, I0/I1	1
Store, register pre-indexed, minus	STR, STRB, STRD, STRH	3 (2)	1	I0/I1, S	1
Store, scaled register pre-indexed, plus LSL2	STR, STRB	1 (2)	1	S, M	1
Store, scaled register pre-indexed, other	STR, STRB	3 (2)	1	I0/I1, S	1
Store, immed post-indexed	STR{T}, STRB{T}, STRD, STRH{T}	1 (1)	1	S, I0/I1	1

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store, register post-indexed	STRH{T}, STRD	1 (1)	1	S, I0/I1	1
Store, register post-indexed	STR{T}, STRB{T}	1 (2)	1	S, M	1
Store, scaled register post-indexed	STR{T}, STRB{T}	1 (2)	1	S, M	1
Store multiple, no writeback	STMIA, STMIB, STMDA, STMDB	N	1/N	S	1, 2
Store multiple, writeback	STMIA, STMIB, STMDA, STMDB, PUSH	N	1/N	S, I0/I1	1, 2

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store register, unscaled immed	STUR, STURB, STURH	1	1	S	
Store register, immed post-index	STR, STRB, STRH	1 (1)	1	S, I0/I1	1
Store register, immed pre-index	STR, STRB, STRH	1 (1)	1	S, I0/I1	1
Store register, immed unprivileged	STTR, STTRB, STTRH	1	1	S	
Store register, unsigned immed	STR, STRB, STRH	1	1	S	
Store register, register offset, basic	STR, STRB, STRH	1	1	S	
Store register, register offset, scaled by 4/8	STR	1	1	S	
Store register, register offset, scaled by 2	STRH	3	1	I0/I1, S	
Store register, register offset, extend	STR, STRB, STRH	1	1	S	
Store register, register offset, extend, scale by 4/8	STR	1	1	S	
Store register, register offset, extend, scale by 1	STRH	3	1	I0/I1, S	
Store pair, immed offset, W-form	STP, STNP	1	1	S	
Store pair, immed offset, X-form	STP, STNP	2	1/2	S	
Store pair, immed post-index, W-form	STP	1 (1)	1	S, I0/I1	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store pair, immed post-index, X-form	STP	2 (1)	1/2	S, I0/I1	1
Store pair, immed pre-index, W-form	STP	1 (1)	1	S, I0/I1	1
Store pair, immed pre-index, X-form	STP	2 (1)	1/2	S, I0/I1	1

Note:

1. Base register updates are typically completed in parallel with the store operation and with shorter latency (update latency is shown in parentheses).
2. For store multiple instructions, N=floor((num_regs+1)/2).

3.10 FP data processing instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP absolute value	VABS	3	2	F0/F1	
FP arith	VADD, VSUB	5	2	F0/F1	
FP compare, unconditional	VCMP, VCMPE	3	1	F1	
FP compare, conditional	VCMP, VCMPE	6	1/6	F0/F1, F1	
FP convert	VCVT{R}, VCVTB, VCVTT, VCVTA, VCVTM, VCVTN, VCVTP	5	2	F0/F1	
FP round to integral	VRINTA, VRINTM, VRINTN, VRINTP, VRINTR, VRINTX, VRINTZ	5	2	F0/F1	
FP divide, S-form	VDIV	7-17	2/15-2/5	F0	1
FP divide, D-form	VDIV	7-32	1/30-1/5	F0	1
FP max/min	VMAXNM, VMINNM	5	2	F0/F1	
FP multiply, FZ	VMUL, VNMUL	5	2	F0/F1	2
FP multiply, no FZ	VMUL, VNMUL	6	2	F0/F1	2
FP multiply accumulate, FZ	VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VNMLA, VNMLS	9 (4)	2	F0/F1	3
FP multiply accumulate, no FZ	VFMA, VFMS, VFNMA, VFNMS, VMLA, VMLS, VNMLA, VNMLS	10 (4)	2	F0/F1	3
FP negate	VNEG	3	2	F0/F1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP select	VSELEQ, VSELGE, VSELGT, VSELVS	3	2	F0/F1	
FP square root, S-form	VSQRT	7-17	2/15-2/5	F0	1
FP square root, D-form	VSQRT	7-32	1/30-1/5	F0	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP absolute value	FABS	3	2	F0/F1	
FP arithmetic	FADD, FSUB	5	2	F0/F1	
FP compare	FCCMP{E}, FCMP{E}	3	1	F1	
FP divide, S-form	FDIV	7-17	2/15-2/5	F0	1
FP divide, D-form	FDIV	7-32	1/30-1/5	F0	1
FP min/max	FMIN, FMINNM, FMAX, FMAXNM	5	2	F0/F1	
FP multiply, FZ	FMUL, FNMUL	5	2	F0/F1	2
FP multiply, no FZ	FMUL, FNMUL	6	2	F0/F1	2
FP multiply accumulate, FZ	FMADD, FMSUB, FNMADD, FNMSUB	9 (4)	2	F0/F1	3
FP multiply accumulate, no FZ	FMADD, FMSUB, FNMADD, FNMSUB	10 (4)	2	F0/F1	3
FP negate	FNEG	3	2	F0/F1	
FP round to integral	FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ	5	2	F0/F1	
FP select	FCSEL	3	2	F0/F1	
FP square root, S-form	FSQRT	7-17	2/15-2/5	F0	1
FP square root, D-form	FSQRT	7-32	1/30-1/5	F0	1

Note:

1. FP divide and square root operations are performed using an iterative algorithm and block subsequent similar operations to the same pipeline until complete.
2. FP multiply-accumulate pipelines support late forwarding of the result from FP multiply *μ*ops to the accumulate operands of an FP multiply-accumulate *μ*op. The latter can potentially be issued one cycle after the FP multiply *μ*op has been issued.
3. FP multiply-accumulate pipelines support late-forwarding of accumulate operands from similar *μ*ops, allowing a typical sequence of multiply-accumulate *μ*ops to issue one every N cycles (accumulate latency N is shown in parentheses).

3.11 FP miscellaneous instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP move, immed	VMOV	3	2	F0/F1	
FP move, register	VMOV	3	2	F0/F1	
FP transfer, vfp to core reg	VMOV	5	1	L	
FP transfer, core reg to upper or lower half of vfp D-reg	VMOV	8	1	L, F0/F1	
FP transfer, core reg to vfp	VMOV	5	1	L	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP convert, from vec to vec reg	FCVT, FCVTXN	5	1	F0/F1	
FP convert, from gen to vec reg	SCVTF, UCVTF	10	1	L, F0/F1	
FP convert, from vec to gen reg	FCVTAS, FCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU	10	1	L, F0/F1	
FP move, immed	FMOV	3	2	F0/F1	
FP move, register	FMOV	3	2	F0/F1	
FP transfer, from gen to vec reg	FMOV	5	1	L	
FP transfer, from vec to gen reg	FMOV	5	1	L	

3.12 FP load instructions

The latencies shown assume the memory access hits in the Level 1 Data Cache. Compared to standard loads, an extra cycle is required to forward results to FP/ASIMD pipelines.

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP load, register	VLDR	5	1	L	
FP load multiple, unconditional	VLDMIA, VLDMDB, VPOP	4 + N	1/N	L	1
FP load multiple, conditional	VLDMIA, VLDMDB, VPOP	4 + N	1/N	L	2

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
(FP load, writeback forms)		(1)	Same as before	+I0/I1	3

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Load vector reg, literal	LDR	5	1	L	
Load vector reg, unscaled immed	LDUR	5	1	L	
Load vector reg, immed post-index	LDR	5 (1)	1	L, I0/I1	3
Load vector reg, immed pre-index	LDR	5 (1)	1	L, I0/I1	3
Load vector reg, unsigned immed	LDR	5	1	L	
Load vector reg, register offset, basic	LDR	5	1	L	
Load vector reg, register offset, scale, S/D-form	LDR	5	1	L	
Load vector reg, register offset, scale, H/Q-form	LDR	6	1	I0/I1, L	
Load vector reg, register offset, extend	LDR	5	1	L	
Load vector reg, register offset, extend, scale, S/D-form	LDR	5	1	L	
Load vector reg, register offset, extend, scale, H/Q-form	LDR	6	1	I0/I1, L	
Load vector pair, immed offset, S/D-form	LDP, LDNP	5	1	L	
Load vector pair, immed offset, Q-form	LDP, LDNP	6	1/2	L	
Load vector pair, immed post-index, S/D-form	LDP	5 (1)	1	L, I0/I1	3
Load vector pair, immed post-index, Q-form	LDP	6 (1)	1/2	L, I0/I1	3
Load vector pair, immed pre-index, S/D-form	LDP	5 (1)	1	L, I0/I1	3
Load vector pair, immed pre-index, Q-form	LDP	6 (1)	1/2	L, I0/I1	3

Note:

1. For FP load multiple instructions, N=floor((num_regs+1)/2) for unconditional forms only.
2. For conditional FP load multiple instructions, N = num_regs for conditional forms only.
3. Writeback forms of load instructions require an extra μop to update the base address. This update is typically performed in parallel with, or prior to, the load μop (update latency is shown in parentheses).

3.13 FP store instructions

Store *μ*ops can issue after their address operands become available and do not need to wait for data operands. After they are executed, stores are buffered and committed in the background.

Instruction Group	Aarch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
FP store, immed offset	VSTR	1	1	S	
FP store multiple, S-form	VSTMIA, VSTMDB, VPUSH	N	1/N	S	1
FP store multiple, D-form	VSTMIA, VSTMDB, VPUSH	N	1/N	S	1
(FP store, writeback forms)		(1)	Same as before	+I0/I1	2

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store vector reg, unscaled immed, B/H/S/D-form	STUR	1	1	S	
Store vector reg, unscaled immed, Q-form	STUR	2	1/2	S	
Store vector reg, immed post-index, B/H/S/D-form	STR	1 (1)	1	S, I0/I1	2
Store vector reg, immed post-index, Q-form	STR	2 (1)	1/2	S, I0/I1	2
Store vector reg, immed pre-index, B/H/S/D-form	STR	1 (1)	1	S, I0/I1	2
Store vector reg, immed pre-index, Q-form	STR	2 (1)	1/2	I0/I1, S	2
Store vector reg, unsigned immed, B/H/S/D-form	STR	1	1	S	
Store vector reg, unsigned immed, Q-form	STR	2	1/2	I0/I1, S	
Store vector reg, register offset, basic, B/H/S/D-form	STR	1	1	S	
Store vector reg, register offset, basic, Q-form	STR	2	1/2	I0/I1, S	
Store vector reg, register offset, scale, H-form	STR	3	1	I0/I1, S	
Store vector reg, register offset, scale, S/D-form	STR	1	1	S	
Store vector reg, register offset, scale, Q-form	STR	4	1/2	I0/I1, S	
Store vector reg, register offset, extend, B/H/S/D-form	STR	1	1	S	
Store vector reg, register offset, extend, Q-form	STR	4	1/2	M, S	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Store vector reg, register offset, extend, scale, H-form	STR	3	1	I0/I1, S	
Store vector reg, register offset, extend, scale, S/D-form	STR	1	1	S	
Store vector reg, register offset, extend, scale, Q-form	STR	4	1/2	I0/I1, S	
Store vector pair, immed offset, S-form	STP	1	1	S	
Store vector pair, immed offset, D-form	STP	2	1/2	S S	
Store vector pair, immed offset, Q-form	STP	4	1/4	I0/I1, S	
Store vector pair, immed post-index, S-form	STP	1 (1)	1	S, I0/I1	2
Store vector pair, immed post-index, D-form	STP	2 (1)	1/2	S, I0/I1	2
Store vector pair, immed post-index, Q-form	STP	4 (1)	1/4	S, I0/I1	2
Store vector pair, immed pre-index, S-form	STP	1 (1)	1	S, I0/I1	2
Store vector pair, immed pre-index, D-form	STP	2 (1)	1/2	S, I0/I1	2
Store vector pair, immed pre-index, Q-form	STP	4 (1)	1/4	I0/I1, S	2

Note:

1. For single-precision store-multiple instructions, N=floor((num_regs+1)/2). For double-precision stores, N=(num_regs).
2. Writeback forms of store instructions require an extra *μ*op to update the base address. This update is typically performed in parallel with, or prior to, the store *μ*op (address update latency is shown in parentheses).

3.14 ASIMD integer instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD absolute diff, D-form	VABD	3	2	F0/F1	
ASIMD absolute diff, Q-form	VABD	3	1	F0/ F1	
ASIMD absolute diff accum, D-form	VABA	4 (1)	1	F1	2
ASIMD absolute diff accum, Q-form	VABA	5 (2)	1/2	F1	2
ASIMD absolute diff accum long	VABAL	4 (1)	1	F1	2
ASIMD absolute diff long	VABDL	3	2	F0/F1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD arith, basic	VADD, VADDL, VADDW, VNEG, VPADD, VPADDL, VSUB, VSUBL, VSUBW	3	2	F0/F1	
ASIMD arith, complex	VABS, VADDHN, VHADD, VHSUB, VQABS, VQADD, VQNEG, VQSUB, VRADDHN, VRHADD, VRSUBHN, VSUBHN	3	2	F0/F1	
ASIMD compare	VCEQ, VCGE, VCGT, VCLE, VTST	3	2	F0/F1	
ASIMD logical	VAND, VBIC, VMVN, VORR, VORN, VEOR	3	2	F0/F1	
ASIMD max/min	VMAX, VMIN, VPMAX, VPMIN	3	2	F0/F1	
ASIMD multiply, D-form	VMUL, VQDMULH, VQRDMULH	5/4	1	F0	4
ASIMD multiply, Q-form	VMUL, VQDMULH, VQRDMULH	6/5	1/2	F0	4
ASIMD multiply accumulate, D-form	VMLA, VMLS	5/4 (1)	1	F0	1, 4
ASIMD multiply accumulate, Q-form	VMLA, VMLS	6/5 (2)	1/2	F0	1, 4
ASIMD multiply accumulate long	VMLAL, VMLSL	5/4 (1)	1	F0	1, 4
ASIMD multiply accumulate saturating long	VQDMLAL, VQDMLSL	5/4 (2)	1	F0	1, 4
ASIMD multiply long	VMULL.S, VMULL.I, VMULL.P8, VQDMULL	5/4	1	F0	4
ASIMD pairwise add and accumulate	VPADAL	4 (1)	1	F1	2
ASIMD shift accumulate	VSRA, VRSRA	4 (1)	1	F1	2
ASIMD shift by immed, basic	VMOVL, VSHL, VSHLL, VSHR, VSHRN	3	1	F1	
ASIMD shift by immed, complex	VQRSHRN, VQRSHRUN, VQSHL{U}, VQSHRN, VQSHRUN, VRSHR, VRSHRN	4	1	F1	
ASIMD shift by immed and insert, basic, D-form	VSLI, VSRI	3	1	F1	
ASIMD shift by immed and insert, basic, Q-form	VSLI, VSRI	4	1/2	F1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD shift by register, basic, D-form	VSHL	3	1	F1	
ASIMD shift by register, basic, Q-form	VSHL	4	1/2	F1	
ASIMD shift by register, complex, D-form	VQRSHL, VQSHL, VRSHL	4	1	F1	
ASIMD shift by register, complex, Q-form	VQRSHL, VQSHL, VRSHL	5	1/2	F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD absolute diff, D-form	SABD, UABD	3	2	F0/F1	
ASIMD absolute diff, Q-form	SABD, UABD	3	1	F0/F1	
ASIMD absolute diff accum, D-form	SABA, UABA	4 (1)	1	F1	2
ASIMD absolute diff accum, Q-form	SABA, UABA	5 (2)	1/2	F1	2
ASIMD absolute diff accum long	SABAL(2), UABAL(2)	4 (1)	1	F1	2
ASIMD absolute diff long	SABDL, UABDL	3	2	F0/F1	
ASIMD arith, basic	ABS, ADD, ADDP, NEG, SADDL(2), SADDLP, SADDW(2), SHADD, SHSUB, SSUBL(2), SSUBW(2), SUB, UADDL(2), UADDLP, UADDW(2), UHADD, UHSUB, USUBW(2)	3	2	F0/F1	
ASIMD arith, complex	ADDHN(2), RADDHN(2), RSUBHN(2), SQABS, SQADD, SQNEG, SQSUB, SRHADD, SUBHN(2), SUQADD, UQADD, UQSUB, URHADD, USQADD	3	2	F0/F1	
ASIMD arith, reduce, 4H/4S	ADDV, SADDLV, UADDLV	4	1	F1	
ASIMD arith, reduce, 8B/8H	ADDV, SADDLV, UADDLV	7	1	F1, F0/F1	
ASIMD arith, reduce, 16B	ADDV, SADDLV, UADDLV	8	1/2	F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD compare	CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE, CMLT, CMTST	3	2	F0/F1	
ASIMD logical	AND, BIC, EOR, MOV, MVN, ORN, ORR	3	2	F0/F1	
ASIMD max/min, basic	SMAX, SMAXP, SMIN, SMINP, UMAX, UMAXP, UMIN, UMINP	3	2	F0/F1	
ASIMD max/min, reduce, 4H/4S	SMAXV, SMINV, UMAXV, UMINV	4	1	F1	
ASIMD max/min, reduce, 8B/8H	SMAXV, SMINV, UMAXV, UMINV	7	1	F1, F0/F1	
ASIMD max/min, reduce, 16B	SMAXV, SMINV, UMAXV, UMINV	8	1/2	F1	
ASIMD multiply, D-form	MUL, PMUL, SQDMULH, SQRDMULH	5/4	1	F0	4
ASIMD multiply, Q-form	MUL, PMUL, SQDMULH, SQRDMULH	6/5	1/2	F0	4
ASIMD multiply accumulate, D-form	MLA, MLS	5/4 (1)	1	F0	4
ASIMD multiply accumulate, Q-form	MLA, MLS	6/5 (2)	1/2	F0	4
ASIMD multiply accumulate long	SMLAL(2), SMLSL(2), UMLAL(2), UMLSL(2)	5/4 (1)	1	F0	1, 4
ASIMD multiply accumulate saturating long	SQDMLAL(2), SQDMLSL(2)	5/4 (2)	1	F0	1, 4
ASIMD multiply long	SMULL(2), UMULL(2), SQDMULL(2)	5/4	1	F0	4
ASIMD polynomial (8x8) multiply long	PMULL.8B, PMULL2.16B	5/4	1	F0	3, 4
ASIMD pairwise add and accumulate	SADALP, UADALP	4 (1)	1	F1	2
ASIMD shift accumulate	SRA, SRSRA, USRA, URSRA	4 (1)	1	F1	2
ASIMD shift by immed, basic	SHL, SHLL(2), SHRN(2), SLI, SRI, SSHLL(2), SSHR, SXTL(2), USHLL(2), USHR, UXTL(2)	3	1	F1	
ASIMD shift by immed and insert, basic, D-form	SLI, SRI	3	1	F1	
ASIMD shift by immed and insert, basic, Q-form	SLI, SRI	4	1/2	F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD shift by immed, complex	RSHRN(2), SRSHR, SQSHL{U}, SQRSHRN(2), SQRSHRUN(2), SQSHRN(2), SQSHRUN(2), URSHR, UQSHL, UQRSHRN(2), UQSHRN(2)	4	1	F1	
ASIMD shift by register, basic, D-form	SSHL, USHL	3	1	F1	
ASIMD shift by register, basic, Q-form	SSHL, USHL	4	1/2	F1	
ASIMD shift by register, complex, D-form	SRSHL, SQRSHL, SQSHL, URSHL, UQRSHL, UQSHL	4	1	F1	
ASIMD shift by register, complex, Q-form	SRSHL, SQRSHL, SQSHL, URSHL, UQRSHL, UQSHL	5	1/2	F1	

Note:

1. Multiply-accumulate pipelines support late-forwarding of accumulate operands from similar *μ*ops, allowing a typical sequence of integer multiply-accumulate *μ*ops to issue one every cycle or one every other cycle (accumulate latency is shown in parentheses).
2. Other accumulate pipelines also support late-forwarding of accumulate operands from similar *μ*ops, allowing a typical sequence of such *μ*ops to issue one every cycle (accumulate latency is shown in parentheses).
3. This category includes instructions of the form “PMULL Vd.8H, Vn.8B, Vm.8B” and “PMULL2 Vd.8H, Vn.16B, Vm.16B”.
4. The Cortex®-A57 processor r1p0 and later reduce the latency of ASIMD multiply and multiply-with-accumulate instructions relative to r0pX. Latencies listed as ‘N/M’ imply a latency of N on r0pX and M on r1p0.

3.15 ASIMD floating-point instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP absolute value	VABS	3	2	F0/F1	
ASIMD FP arith, D-form	VABD, VADD, VPADD, VSUB	5	2	F0/F1	
ASIMD FP arith, Q-form	VABD, VADD, VSUB	5	1	F0/F1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP compare, D-form	VACGE, VACGT, VACLE, VACLT, VCEQ, VCGE, VCGT, VCLE	5	2	F0/F1	
ASIMD FP compare, Q-form	VACGE, VACGT, VACLE, VACLT, VCEQ, VCGE, VCGT, VCLE	5	1	F0/F1	
ASIMD FP convert, integer, D-form	VCVT, VCVTA, VCVTM, VCVTN, VCVTP	5	2	F0/F1	
ASIMD FP convert, integer, Q-form	VCVT, VCVTA, VCVTM, VCVTN, VCVTP	5	1	F0/F1	
ASIMD FP convert, fixed, D-form	VCVT	5	2	F0/F1	
ASIMD FP convert, fixed, Q-form	VCVT	5	1	F0/F1	
ASIMD FP convert, half-precision	VCVT	8	2/3	F0/F1	
ASIMD FP max/min, D-form	VMAX, VMIN, VPMAX, VPMIN, VMAXNM, VMINNM	5	2	F0/F1	
ASIMD FP max/min, Q-form	VMAX, VMIN, VMAXNM, VMINNM	5	1	F0/F1	
ASIMD FP multiply, D-form	VMUL	5	2	F0/F1	2
ASIMD FP multiply, Q-form	VMUL	5	1	F0/F1	2
ASIMD FP multiply accumulate, D-form	VMLA, VMLS, VFMA, VFMS	9 (4)	2	F0/F1	1
ASIMD FP multiply accumulate, Q-form	VMLA, VMLS, VFMA, VFMS	9 (4)	1	F0/F1	1
ASIMD FP negate	VNEG	3	2	F0/F1	
ASIMD FP round to integral, D-form	VRINTA, VRINTM, VRINTN, VRINTP, VRINTX, VRINTZ	5	2	F0/F1	
ASIMD FP round to integral, Q-form	VRINTA, VRINTM, VRINTN, VRINTP, VRINTX, VRINTZ	5	1	F0/F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP absolute value	FABS	3	2	F0/F1	
ASIMD FP arith, normal, D-form	FABD, FADD, FSUB	5	2	F0/F1	
ASIMD FP arith, normal, Q-form	FABD, FADD, FSUB	5	1	F0/F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP arith, pairwise, D-form	FADDP	5	2	F0/F1	
ASIMD FP arith, pairwise, Q-form	FADDP	8	2/3	F0/F1	
ASIMD FP compare, D-form	FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT	5	2	F0/F1	
ASIMD FP compare, Q-form	FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT	5	1	F0/F1	
ASIMD FP convert, long	FCVTL(2)	8	2/3	F0/F1	
ASIMD FP convert, narrow	FCVTN(2), FCVTXN(2)	8	2/3	F0/F1	
ASIMD FP convert, other, D-form	FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF	5	2	F0/F1	
ASIMD FP convert, other, Q-form	FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF	5	1	F0/F1	
ASIMD FP divide, D-form, F32	FDIV	7-17	1/15-1/5	F0	3
ASIMD FP divide, Q-form, F32	FDIV	14-34	1/30-1/10	F0	3
ASIMD FP divide, Q-form, F64	FDIV	14-64	1/60-1/10	F0	3
ASIMD FP max/min, normal, D-form	FMAX, FMAXNM, FMIN, FMINNM	5	2	F0/F1	
ASIMD FP max/min, normal, Q-form	FMAX, FMAXNM, FMIN, FMINNM	5	1	F0/F1	
ASIMD FP max/min, pairwise, D-form	FMAXP, FMAXNMP, FMINP, FMINNMP	5	2	F0/F1	
ASIMD FP max/min, pairwise, Q-form	FMAXP, FMAXNMP, FMINP, FMINNMP	9	2/3	F0/F1	
ASIMD FP max/min, reduce	FMAXV, FMAXNMV, FMINV, FMINNMV	10	1	F0/F1	
ASIMD FP multiply, D-form, FZ	FMUL, FMULX	5	2	F0/F1	2
ASIMD FP multiply, D-form, no FZ	FMUL, FMULX	6	2	F0/F1	2
ASIMD FP multiply, Q-form, FZ	FMUL, FMULX	5	1	F0/F1	2

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD FP multiply, Q-form, no FZ	FMUL, FMULX	6	1	F0/F1	2
ASIMD FP multiply accumulate, D-form, FZ	FMLA, FMLS	9 (4)	2	F0/F1	1
ASIMD FP multiply accumulate, D-form, no FZ	FMLA, FMLS	9 (4)	2	F0/F1	1
ASIMD FP multiply accumulate, Q-form, FZ	FMLA, FMLS	10 (4)	1	F0/F1	1
ASIMD FP multiply accumulate, Q-form, no FZ	FMLA, FMLS	10 (4)	1	F0/F1	1
ASIMD FP negate	FNEG	3	2	F0/F1	
ASIMD FP round, D-form	FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ	5	2	F0/F1	
ASIMD FP round, Q-form	FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ	5	1	F0/F1	

Note:

1. ASIMD multiply-accumulate pipelines support late-forwarding of accumulate operands from similar *µ*ops, allowing a typical sequence of floating-point multiply-accumulate *µ*ops to issue one every four cycles (accumulate latency is shown in parentheses).
2. ASIMD multiply-accumulate pipelines support late forwarding of the result from ASIMD FP multiply *µ*ops to the accumulate operands of an ASIMD FP multiply-accumulate *µ*op. The latter can potentially be issued one cycle after the ASIMD FP multiply *µ*op has been issued.
3. FP divide operations are performed using an iterative algorithm and block subsequent similar operations to the same pipeline until complete.

3.16 ASIMD miscellaneous instructions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD bitwise insert, D-form	VBIF, VBIT, VBSL	3	2	F0/F1	
ASIMD bitwise insert, Q-form	VBIF, VBIT, VBSL	3	1	F0/F1	
ASIMD count, D-form	VCLS, VCLZ, VCNT	3	2	F0/F1	
ASIMD count, Q-form	VCLS, VCLZ, VCNT	3	1	F0/F1	
ASIMD duplicate, core reg, D-form	VDUP	8	1	L, F0/F1	

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD duplicate, core reg, Q-form	VDUP	8	1	L, F0/F1	
ASIMD duplicate, scalar	VDUP	3	2	F0/F1	
ASIMD extract	VEXT	3	2	F0/F1	
ASIMD move, immed	VMOV	3	2	F0/F1	
ASIMD move, register	VMOV	3	2	F0/F1	
ASIMD move, narrowing	VMOVN	3	2	F0/F1	
ASIMD move, saturating	VQMOVN, VQMOVUN	4	1	F1	
ASIMD reciprocal estimate, D-form	VRECPE, VRSQRTE	5	2	F0/F1	
ASIMD reciprocal estimate, Q-form	VRECPE, VRSQRTE	5	1	F0/F1	
ASIMD reciprocal step, D-form, FZ	VRECPS, VRSQRTS	9	2	F0/F1	
ASIMD reciprocal step, D-form, no FZ	VRECPS, VRSQRTS	10	2	F0/F1	
ASIMD reciprocal step, Q-form, FZ	VRECPS, VRSQRTS	9	1	F0/F1	
ASIMD reciprocal step, Q-form, no FZ	VRECPS, VRSQRTS	10	1	F0/F1	
ASIMD reverse	VREV16, VREV32, VREV64	3	2	F0/F1	
ASIMD swap, D-form	VSWP	3	2	F0/F1	
ASIMD swap, Q-form	VSWP	3	1	F0/F1	
ASIMD table lookup, 1 reg	VTBL, VTBX	3	2	F0/F1	
ASIMD table lookup, 2 reg	VTBL, VTBX	3	2	F0/F1	
ASIMD table lookup, 3 reg	VTBL, VTBX	6	2	F0/F1	
ASIMD table lookup, 4 reg	VTBL, VTBX	6	2	F0/F1	
ASIMD transfer, scalar to core reg	VMOV	6	1	L, I0/I1	
ASIMD transfer, core reg to scalar	VMOV	8	1	L, F0/F1	
ASIMD transpose, D-form	VTRN	3	2	F0/F1	
ASIMD transpose, Q-form	VTRN	3	1	F0/F1	
ASIMD unzip/zip, D-form	VUZP, VZIP	3	2	F0/F1	
ASIMD unzip/zip, Q-form	VUZP, VZIP	6	2/3	F0/F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD bit reverse	RBIT	3	2	F0/F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD bitwise insert, D-form	BIF, BIT, BSL	3	2	F0/F1	
ASIMD bitwise insert, Q-form	BIF, BIT, BSL	3	1	F0/F1	
ASIMD count, D-form	CLS, CLZ, CNT	3	2	F0/F1	
ASIMD count, Q-form	CLS, CLZ, CNT	3	1	F0/F1	
ASIMD duplicate, gen reg	DUP	8	1	L, F0/F1	
ASIMD duplicate, element	DUP	3	2	F0/F1	
ASIMD extract	EXT	3	2	F0/F1	
ASIMD extract narrow	XTN	3	2	F0/F1	
ASIMD extract narrow, saturating	SQXTN(2), SQXTUN(2), UQXTN(2)	4	1	F1	
ASIMD insert, element to element	INS	3	2	F0/F1	
ASIMD move, integer immed	MOVI	3	2	F0/F1	
ASIMD move, FP immed	FMOV	3	2	F0/F1	
ASIMD reciprocal estimate, D-form	FRECPE, FRECPX, FRSQRTE, URECPE, URSQRTE	5	2	F0/F1	
ASIMD reciprocal estimate, Q-form	FRECPE, FRECPX, FRSQRTE, URECPE, URSQRTE	5	1	F0/F1	
ASIMD reciprocal step, D-form, FZ	FRECPS, FRSQRTS	9	2	F0/F1	
ASIMD reciprocal step, D-form, no FZ	FRECPS, FRSQRTS	10	2	F0/F1	
ASIMD reciprocal step, Q-form, FZ	FRECPS, FRSQRTS	9	1	F0/F1	
ASIMD reciprocal step, Q-form, no FZ	FRECPS, FRSQRTS	10	1	F0/F1	
ASIMD reverse	REV16, REV32, REV64	3	2	F0/F1	
ASIMD table lookup, D-form	TBL, TBX	3xN		F0/F1	1
ASIMD table lookup, Q-form	TBL, TBX	3xN + 3		F0/F1	1
ASIMD transfer, element to gen reg, word or dword	UMOV	5	1	L	
ASIMD transfer, element to gen reg	SMOV, UMOV	6	1	L, I0/I1	
ASIMD transfer, gen reg to element	INS	8	1	L, F0/F1	
ASIMD transpose	TRN1, TRN2	3	2	F0/F1	
ASIMD unzip/zip	UZP1, UZP2, ZIP1, ZIP2	3	2	F0/F1	

Note:

1. For table branches (TBL and TBX), N denotes the number of registers in the table.

3.17 ASIMD load instructions

The latencies shown assume the memory access hits in the Level 1 Data Cache. Compared to standard loads, an extra cycle is required to forward results to FP/ASIMD pipelines.

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD load, 1 element, multiple, 1 reg	VLD1	5	1	L	
ASIMD load, 1 element, multiple, 2 reg	VLD1	5	1	L	
ASIMD load, 1 element, multiple, 3 reg	VLD1	6	1/2	L	
ASIMD load, 1 element, multiple, 4 reg	VLD1	6	1/2	L	
ASIMD load, 1 element, one lane	VLD1	8	1	L, F0/F1	
ASIMD load, 1 element, all lanes	VLD1	8	1	L, F0/F1	
ASIMD load, 2 element, multiple, 2 reg	VLD2	8	1	L, F0/F1	
ASIMD load, 2 element, multiple, 4 reg	VLD2	9	1/2	L, F0/F1	
ASIMD load, 2 element, one lane, size 32	VLD2	8	1	L, F0/F1	
ASIMD load, 2 element, one lane, size 8/16	VLD2	8	1	L, F0/F1	
ASIMD load, 2 element, all lanes	VLD2	8	1	L, F0/F1	
ASIMD load, 3 element, multiple, 3 reg	VLD3	9	1/2	L, F0/F1	
ASIMD load, 3 element, one lane, size 32	VLD3	8	1	L, F0/F1	
ASIMD load, 3 element, one lane, size 8/16	VLD3	9	2/3	L, F0/F1	
ASIMD load, 3 element, all lanes	VLD3	8	1	L, F0/F1	
ASIMD load, 4 element, multiple, 4 reg	VLD4	9	1/2	L, F0/F1	
ASIMD load, 4 element, one lane, size 32	VLD4	8	1	L, F0/F1	
ASIMD load, 4 element, one lane, size 8/16	VLD4	9	1/2	L, F0/F1	
ASIMD load, 4 element, all lanes	VLD4	8	1	L, F0/F1	
(ASIMD load, writeback form)		(1)	Same as before	+I0/I1	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD load, 1 element, multiple, 1 reg, D-form	LD1	5	1	L	
ASIMD load, 1 element, multiple, 1 reg, Q-form	LD1	5	1	L	
ASIMD load, 1 element, multiple, 2 reg, D-form	LD1	5	1	L	
ASIMD load, 1 element, multiple, 2 reg, Q-form	LD1	6	1/2	L	
ASIMD load, 1 element, multiple, 3 reg, D-form	LD1	6	1/2	L	
ASIMD load, 1 element, multiple, 3 reg, Q-form	LD1	7	1/3	L	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD load, 1 element, multiple, 4 reg, D-form	LD1	6	1/2	L	
ASIMD load, 1 element, multiple, 4 reg, Q-form	LD1	8	1/4	L	
ASIMD load, 1 element, one lane, B/H/S	LD1	8	1	L, F0/F1	
ASIMD load, 1 element, one lane, D	LD1	5	1	L	
ASIMD load, 1 element, all lanes, D-form, B/H/S	LD1R	8	1	L, F0/F1	
ASIMD load, 1 element, all lanes, D-form, D	LD1R	5	1	L	
ASIMD load, 1 element, all lanes, Q-form	LD1R	8	1	L, F0/F1	
ASIMD load, 2 element, multiple, D-form, B/H/S	LD2	8	1	L, F0/F1	
ASIMD load, 2 element, multiple, Q-form, B/H/S	LD2	9	1/2	L, F0/F1	
ASIMD load, 2 element, multiple, Q-form, D	LD2	6	1/2	L	
ASIMD load, 2 element, one lane, B/H	LD2	8	1	L, F0/F1	
ASIMD load, 2 element, one lane, S	LD2	8	1	L, F0/F1	
ASIMD load, 2 element, one lane, D	LD2	6	1	L	
ASIMD load, 2 element, all lanes, D-form, B/H/S	LD2R	8	1	L, F0/F1	
ASIMD load, 2 element, all lanes, D-form, D	LD2R	5	1	L	
ASIMD load, 2 element, all lanes, Q-form	LD2R	8	1	L, F0/F1	
ASIMD load, 3 element, multiple, D-form, B/H/S	LD3	9	1/2	L, F0/F1	
ASIMD load, 3 element, multiple, Q-form, B/H/S	LD3	10	1/3	L, F0/F1	
ASIMD load, 3 element, multiple, Q-form, D	LD3	8	1/4	L	
ASIMD load, 3 element, one lane, B/H	LD3	9	2/3	L, F0/F1	
ASIMD load, 3 element, one lane, S	LD3	8	1	L, F0/F1	
ASIMD load, 3 element, one lane, D	LD3	6	1/2	L	
ASIMD load, 3 element, all lanes, D-form, B/H/S	LD3R	8	1	L, F0/F1	
ASIMD load, 3 element, all lanes, D-form, D	LD3R	6	1/2	L	
ASIMD load, 3 element, all lanes, Q-form, B/H/S	LD3R	9	2/3	L, F0/F1	
ASIMD load, 3 element, all lanes, Q-form, D	LD3R	9	1/2	L, F0/F1	
ASIMD load, 4 element, multiple, D-form, B/H/S	LD4	9	1/2	L, F0/F1	
ASIMD load, 4 element, multiple, Q-form, B/H/S	LD4	11	1/4	L, F0/F1	
ASIMD load, 4 element, multiple, Q-form, D	LD4	8	1/4	L	
ASIMD load, 4 element, one lane, B/H	LD4	9	1/2	L, F0/F1	
ASIMD load, 4 element, one lane, S	LD4	8	1	L, F0/F1	
ASIMD load, 4 element, one lane, D	LD4	6	1/2	L	
ASIMD load, 4 element, all lanes, D-form, B/H/S	LD4R	8	1	L, F0/F1	
ASIMD load, 4 element, all lanes, D-form, D	LD4R	6	1	L	
ASIMD load, 4 element, all lanes, Q-form, B/H/S	LD4R	9	1/2	L, F0/F1	
ASIMD load, 4 element, all lanes, Q-form, D	LD4R	9	2/5	L, F0/F1	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
(ASIMD load, writeback form)		(1)	Same as before	+I0/I1	1

Note:

1. Writeback forms of load instructions require an extra *µ*op to update the base address. This update is typically performed in parallel with the load *µ*op (update latency is shown in parentheses).

3.18 ASIMD store instructions

Store *µ*ops can issue after their address operands become available and do not need to wait for data operands. After they are executed, stores are buffered and committed in the background.

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD store, 1 element, multiple, 1 reg	VST1	1	1	S	
ASIMD store, 1 element, multiple, 2 reg	VST1	2	1/2	S	
ASIMD store, 1 element, multiple, 3 reg	VST1	3	1/3	S	
ASIMD store, 1 element, multiple, 4 reg	VST1	4	1/4	S	
ASIMD store, 1 element, one lane	VST1	3	1	F0/F1, S	
ASIMD store, 2 element, multiple, 2 reg	VST2	3	1/2	F0/F1, S	
ASIMD store, 2 element, multiple, 4 reg	VST2	4	1/4	F0/F1, S	
ASIMD store, 2 element, one lane	VST2	3	1	F0/F1, S	
ASIMD store, 3 element, multiple, 3 reg	VST3	3	1/3	F0/F1, S	
ASIMD store, 3 element, one lane, size 32	VST3	3	1/2	F0/F1, S	
ASIMD store, 3 element, one lane, size 8/16	VST3	3	1	F0/F1, S	
ASIMD store, 4 element, multiple, 4 reg	VST4	4	1/4	F0/F1, S	
ASIMD store, 4 element, one lane, size 32	VST4	3	1/2	F0/F1, S	
ASIMD store, 4 element, one lane, size 8/16	VST4	3	1	F0/F1, S	
(ASIMD store, writeback form)			+1	+I0/I1	1

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD store, 1 element, multiple, 1 reg, D-form	ST1	1	1	S	
ASIMD store, 1 element, multiple, 1 reg, Q-form	ST1	2	1/2	S	
ASIMD store, 1 element, multiple, 2 reg, D-form	ST1	2	1/2	S	
ASIMD store, 1 element, multiple, 2 reg, Q-form	ST1	4	1/4	S	
ASIMD store, 1 element, multiple, 3 reg, D-form	ST1	3	1/3	S	
ASIMD store, 1 element, multiple, 3 reg, Q-form	ST1	6	1/6	S	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
ASIMD store, 1 element, multiple, 4 reg, D-form	ST1	4	1/4	S	
ASIMD store, 1 element, multiple, 4 reg, Q-form	ST1	8	1/8	S	
ASIMD store, 1 element, one lane, B/H/S	ST1	3	1	F0/F1, S	
ASIMD store, 1 element, one lane, D	ST1	1	1	S	
ASIMD store, 2 element, multiple, D-form, B/H/S	ST2	3	1/2	F0/F1, S	
ASIMD store, 2 element, multiple, Q-form, B/H/S	ST2	4	1/4	F0/F1, S	
ASIMD store, 2 element, multiple, Q-form, D	ST2	4	1/4	S	
ASIMD store, 2 element, one lane, B/H/S	ST2	3	1	F0/F1, S	
ASIMD store, 2 element, one lane, D	ST2	2	1/2	S	
ASIMD store, 3 element, multiple, D-form, B/H/S	ST3	3	1/3	F0/F1, S	
ASIMD store, 3 element, multiple, Q-form, B/H/S	ST3	6	1/6	F0/F1, S	
ASIMD store, 3 element, multiple, Q-form, D	ST3	6	1/6	S	
ASIMD store, 3 element, one lane, B/H	ST3	3	1	F0/F1, S	
ASIMD store, 3 element, one lane, S	ST3	3	1/2	F0/F1, S	
ASIMD store, 3 element, one lane, D	ST3	3	1/3	S	
ASIMD store, 4 element, multiple, D-form, B/H/S	ST4	4	1/4	F0/F1, S	
ASIMD store, 4 element, multiple, Q-form, B/H/S	ST4	8	1/8	F0/F1, S	
ASIMD store, 4 element, multiple, Q-form, D	ST4	8	1/8	S	
ASIMD store, 4 element, one lane, B/H	ST4	3	1	F0/F1, S	
ASIMD store, 4 element, one lane, S	ST4	3	1/2	F0/F1, S	
ASIMD store, 4 element, one lane, D	ST4	4	1/4	S	
(ASIMD store, writeback form)		(1)	Same as before	+I0/I1	1

Note:

1. Writeback forms of store instructions require an extra μop to update the base address. This update is typically performed in parallel with the store μop (update latency is shown in parentheses).

3.19 Cryptography extensions

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Crypto AES ops	AESD, AESE, AESIMC, AESMC	3	1	F0	1
Crypto polynomial (64x64) multiply long	VMULL.P64	3	1	F0	
Crypto SHA1 xor ops	SHA1SU0	6	2	F0/F1	
Crypto SHA1 fast ops	SHA1H, SHA1SU1	3	1	F0	
Crypto SHA1 slow ops	SHA1C, SHA1M, SHA1P	6	1/2	F0	
Crypto SHA256 fast ops	SHA256SU0	3	1	F0	
Crypto SHA256 slow ops	SHA256H, SHA256H2, SHA256SU1	6	1/2	F0	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
Crypto AES ops	AESD, AESE, AESIMC, AESMC	3	1	F0	1
Crypto AES ops	AESE/AESMC pair, AESD/AESIMC pair	3	1	F0	2
Crypto polynomial (64x64) multiply long	PMULL(2)	3	1	F0	
Crypto SHA1 xor ops	SHA1SU0	6	2	F0/F1	
Crypto SHA1 schedule acceleration ops	SHA1H, SHA1SU1	3	1	F0	
Crypto SHA1 hash acceleration ops	SHA1C, SHA1M, SHA1P	6	1/2	F0	
Crypto SHA256 schedule acceleration op (1 uop)	SHA256SU0	3	1	F0	
Crypto SHA256 schedule acceleration op (2 uops)	SHA256SU1	6	1/2	F0	
Crypto SHA256 hash acceleration ops	SHA256H, SHA256H2	6	1/2	F0	

Note:

1. In the Cortex®-A57 processor r0p0, each AESE/AESMC/AESD/AESIMC will exhibit the described performance characteristics.
2. In the Cortex®-A57 processor r0p1 and later revisions, adjacent AESE/AESMC instruction pairs and adjacent AESD/AESIMC instruction pairs will exhibit the described performance characteristics. See Section 4.12 for additional details.

3.20 CRC

Instruction Group	AArch32 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
CRC checksum ops	CRC32, CRC32C	3	1	M	

Instruction Group	AArch64 Instructions	Exec Latency	Execution Throughput	Utilized Pipelines	Notes
CRC checksum ops	CRC32, CRC32C	3	1	M	

4 SPECIAL CONSIDERATIONS

4.1 Dispatch constraints

Dispatch of μops from the in-order portion to the out-of-order portion of the microarchitecture includes a number of constraints. It is important to consider these constraints during code generation in order to maximize the effective dispatch bandwidth and subsequent execution bandwidth of the Cortex®-A57 processor.

The dispatch stage can process up to three μops per cycle, with the following limitations on the number of μops of each type that can be simultaneously dispatched.

- One μop using the B pipeline.
- Up to two μops using the I pipelines.
- Up to two μops using the M pipeline.
- One μop using the F0 pipeline.
- One μop using the F1 pipeline.
- Up to two μops using the L or S pipeline.

If there are more μops available to be dispatched in a given cycle than can be supported by the constraints above, μops will be dispatched in oldest-to-youngest age order to the extent allowed by the above.

4.2 Conditional execution

The ARMv8 architecture allows many types of A32 instructions to be conditionally executed based upon condition flags (N, Z, C, V). If the condition flags satisfy a condition specified in the instruction encoding, an instruction has its normal effect. If the flags do not satisfy this condition, the instruction acts as a NOP.

This leads to conditional register writes for most types of conditional instructions. In an out-of-order processor such as the Cortex®-A57 processor, this has two side-effects:

- The first side-effect is that the conditional instruction requires the old value of its destination register as an input operand.
- The second side-effect is that all subsequent consumers of the conditional instruction destination register depend upon this operation, regardless of the state of the condition flags (that is, even if the destination register is unchanged if the condition is not met.).

These effects should be taken into account when considering whether to use conditional execution for long-latency operations. The overheads of conditional execution might begin to outweigh the benefits. Consider the following example:

```
MULEQ  R1, R2, R3
MULNE R1, R2, R4
```

For this pair of instructions, the second multiply is dependent upon the result of the first multiply, not through one of its normal input operands (R2 and R4), but through the destination register R1. The combined latency for these instructions is six cycles, rather than the four cycles that would be required if these instructions were not conditional (three cycles latency for the first, and one additional cycle for the second which is fully pipelined behind the first). So if the condition is easily predictable (by the branch predictor), conditional execution can lead to a performance loss. But if the condition is not easily predictable, conditional execution can lead to a performance gain because the latency of a branch mispredict is generally higher than the execution latency of conditional

instructions. In general, ARM recommends that conditional instruction forms be considered only for integer instructions with latency less than or equal to two cycles, loads, and stores.

4.3 Conditional ASIMD

Conditional execution is architecturally possible for ASIMD instructions in Thumb state using IT blocks. However, this type of encoding is considered abnormal and is not recommended for the Cortex®-A57 processor. It is expected to perform worse than the equivalent unconditional encodings.

4.4 Register forwarding hazards

The ARMv8-A architecture allows FP instructions to read and write 32-bit S-registers. In AArch32, each S-register corresponds to one half (upper or lower) of an overlayed 64-bit D-register. Register forwarding hazards might occur when one *µ*op reads a D-register or Q-register operand that has recently been written with one or more S-register results. Consider the following abnormal scenario:

```
VMOV  S0,R0
VMOV  S1,R1
VADD  D2, D1, D0
```

The first two instructions write S0 and S1, which correspond to the bottom and top halves of D0. The third instruction then requires D0 as an input operand. In this scenario, the Cortex®-A57 processor detects that at least one of the upper or lower S0/S1 registers overlayed on D0 were previously written, at which point the VADD instruction is serialized until the prior S-register writes are guaranteed to have been architecturally committed, likely incurring significant additional latency. Note that after the D0 register has been written as a D-register or Q-register destination, subsequent consumers of that register will no longer encounter this register-hazard condition, until the next S-register write, if any.

The Cortex®-A57 processor is able to avoid this register-hazard condition for certain cases. The following rules describe the conditions under which a register-hazard can occur:

- The producer writes an S-register (not a D[x] scalar).
- The consumer reads an overlapping D-register (not as a D[x] scalar, nor as an implicit operand caused by conditional execution).
- The consumer is a FP/ASIMD *µ*op (not a store *µ*op).

To avoid unnecessary hazards, ARM recommends that the programmer uses D[x] scalar writes when populating registers prior to ASIMD operations. For example, either of the following instruction forms would safely prevent a subsequent hazard:

```
VLD1.32     Dd[x], [address]
VMOV.32     Dd[x], Rt
```

The Performance Monitor Unit (PMU) in the Cortex®-A57 processor can be used to determine when register forwarding hazards are actually occuring. The implementation defined PMU event number 0x12C (DISP_SWDW_STALL) has been assigned to count the number of cycles spent stalling because of these hazards.

4.5 Load/Store throughput

The Cortex®-A57 processor includes separate load and store pipelines, which allow it to execute one load *µ*op and one store *µ*op every cycle.

To achieve maximum throughput for memory copy (or similar loops), do the following:

- Unroll the loop to include multiple load and store operations for each iteration, minimizing the overheads of looping.
- Use discrete, non-writeback forms of load and store instructions (such as LDRD and STRD), interleaving them so that one load and one store operation can be performed each cycle. Avoid load-multiple/store-multiple instruction encodings (such as LDM and STM), which lead to separated bursts of load and store *µ*ops and which might not allow concurrent use of both the load and store pipelines.

The following example shows a recommended instruction sequence for a long memory copy in AArch32 state:

```
Loop_start:
      SUBS  r2,r2,#64
      LDRD  r3,r4,[r1,#0]
      STRD  r3,r4,[r0,#0]
      LDRD  r3,r4,[r1,#8]
      STRD  r3,r4,[r0,#8]
      LDRD  r3,r4,[r1,#16]
      STRD  r3,r4,[r0,#16]
      LDRD  r3,r4,[r1,#24]
      STRD  r3,r4,[r0,#24]
      LDRD  r3,r4,[r1,#32]
      STRD  r3,r4,[r0,#32]
      LDRD  r3,r4,[r1,#40]
      STRD  r3,r4,[r0,#40]
      LDRD  r3,r4,[r1,#48]
      STRD  r3,r4,[r0,#48]
      LDRD  r3,r4,[r1,#56]
      STRD  r3,r4,[r0,#56]
      ADD   r1,r1,#64
      ADD   r0,r0,#64
      BGT   Loop_start
```

A recommended copy routine for AArch64 would look similar to the sequence above, but would use LDP/STP instructions.

4.6 Load/Store alignment

The ARMv8-A architecture allows many types of load and store accesses to be arbitrarily aligned. The Cortex®-A57 processor handles most unaligned accesses without performance penalties. However, there are cases which reduce bandwidth or incur additional latency, as described below.

- Load operations that cross a cache-line (64-byte) boundary.
- Store operations that cross a 16-byte boundary.

4.7 Branch alignment

Branch instruction and branch target instruction alignment can affect performance. For best-case performance, consider the following guidelines:

- Try not to include more than two taken branches within the same quadword-aligned quadword of instruction memory.
- Consider aligning subroutine entry points and branch targets to quadword boundaries, within the bounds of the code-density requirements of the program. This will ensure that the subsequent fetch can retrieve four (or a full quadword's worth of) instructions, maximizing fetch bandwidth following the taken branch.

4.8 Setting condition flags

The ARM instruction set includes instruction forms that set the condition flags. In addition to compares, many types of data processing operations set the condition flags as a side-effect. Excessive use of flag-setting instruction forms might result in performance degradation, therefore ARM recommends that, where possible, non-flag-setting instructions and instruction-forms are used except where the condition-flag result is explicitly required for subsequent branches or conditional instructions.

When using the Thumb instruction set, special attention should be given to the use of 16-bit instruction forms. Many of those (such as moves, adds, shifts) automatically set the condition flags. For best performance, consider using the 32-bit encodings which include forms that do not set the condition flags, within the bounds of the code-density requirements of the program.

4.9 Accelerated accumulator forwarding in the floating-point pipelines

As described in chapter 2 of this document, the Cortex®-A57 processor implements two floating point execution pipelines (F0/F1). For versions of the Cortex®-A57 processor prior to r1p3, *µ*ops are steered to one pipeline or the other based upon a load-balancing hardware mechanism. For these versions of the Cortex®-A57 processor, forwarding from a multiply or multiply-accumulate into the accumulator operand of a subsequent multiply-accumulate can be accelerated if all instructions are contained within the same pipeline. Therefore, ARM recommends that when performing a critical sequence of dependent, non-quadword FP/ASIMD floating-point multiply or multiply-accumulate operations, the programmer should ensure the accumulator source and destination registers should both be even or both be odd in order to facilitate accelerated accumulator forwarding.

For the Cortex®-A57 processor r1p3 and later revisions, this constrained register usage model is not required.

4.10 Load balancing in the floating-point pipelines

As described in chapter 2 of this document, the Cortex®-A57 processor implements two floating-point execution pipelines (F0/F1). For versions of the Cortex®-A57 processor prior to r1p3, FP/ASIMD floating-point multiply or multiply-accumulate *µ*ops are steered to one pipeline or the other based upon a load-balancing hardware mechanism. For these versions of the Cortex®-A57 processor, ARM recommends that, subject to the recommendation described in 4.9 above, the programmer uses a balanced mix of odd and even destination D-registers for FP/ASIMD floating-point multiply or multiply-accumulate instructions, to ensure those instructions are evenly distributed across the two floating-point pipelines.

For the Cortex®-A57 processor r1p3 and later revisions, this constrained register usage model is not required.

4.11 Special register access

The Cortex®-A57 processor performs register renaming for general purpose registers to enable speculative and out-of-order instruction execution. However, most special-purpose registers are not renamed. Instructions that read or write non-renamed registers are subjected to one or more of the following additional execution constraints:

- Non-speculative execution – Instructions can only execute non-speculatively.
- In-order execution – Instructions must execute in-order with respect to other similar instructions, or in some cases with respect to all instructions.
- Flush side-effects – Instructions trigger a flush side-effect after executing for synchronization.

The table below summarizes various special instructions and the associated execution constraints or side-effects.

Instructions	Forms	Non-Speculative	In-Order	Flush Side-Effect	Notes
ISB		Yes	Yes	Yes	1
CPS		Yes	Yes	Yes	1
SETEND		Yes	Yes	Yes	1
MRS (read)	APSR, CPSR	Yes	Yes	No	1
MRS (read)	SPSR	No	Yes	No	1
MSR (write)	ASPR_nzcvq, CPSR_f	No	No	No	1, 2, 3
MSR (write)	APSR, CPSR other	Yes	Yes	Yes	1
MSR (write)	SPSR	Yes	Yes	No	1
VMRS (read)	FPSCR to APSR_nzcv	No	No	No	1, 2
VMRS (read)	Other	Yes	Yes	No	1
VMSR (write)	FPSCR, changing only NZCV	Yes	Yes	No	1
VMSR (write)	Other	Yes	Yes	Yes	1
MRC (read)		Some	Yes	No	1, 2, 4
MCR (write)		Yes	Yes	Some	1, 4

Note:

1. Conditional forms of these instructions for which the condition is not satisfied will not access special registers or trigger flush side-effects.
2. Conditional forms of these instructions are always executed non-speculatively and in-order to properly resolve the condition.
3. MSR instructions that write APSR_nzcvq generate a separate *µ*op to write the Q bit. That *µ*op executes non-speculatively and in-order. But the main *µ*op, which writes the NZCV bits, executes as shown in the table above.
4. A subset of MCR instructions must be executed non-speculatively. A subset of MRC instructions trigger flush side-effects for synchronization. Those subsets are not documented here.

4.12 AES encryption/decryption

The Cortex®-A57 processor r0p0 can issue one AESE/AESMC/AESD/AESIMC instruction every cycle (fully pipelined) with an execution latency of three cycles (see Section 3.19). This means encryption or decryption for at least three data chunks should be interleaved for maximum performance:

```
AESE data0, key0
AESMC data0, data0
AESE data1, key0
AESMC data1, data1
AESE data2, key0
AESMC data2, data2
AESE data0, key1
AESMC data0, data0
...
```

In the Cortex®-A57 processor r0p1 and later revisions, pairs of dependent AESE/AESMC and AESD/AESIMC instructions provide higher performance when adjacent, and in the described order, in the program code. Therefore it is important to ensure that these instructions come in pairs in AES encryption/decryption loops, as shown in the code segment above.

4.13 Fast literal generation

The Cortex®-A57 processor r1p0 and later revisions support optimized literal generation for 32-bit and 64-bit code. A typical literal generation sequence in 32-bit code is:

```
MOV rX, #bottom_16_bits
MOVT rX, #top_16_bits
```

In 64-bit code, generating a 32-bit immediate:

```
MOV wX, #bottom_16_bits
MOVK wX, #top_16_bits, lsl #16
```

In 64-bit code, generating the bottom half of a 64-bit immediate:

```
MOV xX, #bottom_16_bits
MOVK xX, #top_16_bits, lsl #16
```

In 64-bit code, generating the top half of a 64-bit immediate:

```
MOVK xX, #bits_47_to_32, lsl #32
MOVK xX, #bits_63_to_48, lsl #48
```

If any of these sequences appear sequentially and in the described order in program code, the two instructions can be executed at lower latency and higher bandwidth than if they do not appear sequentially in the program code, enabling 32-bit literals to be generated in a single cycle and 64-bit literals to be generated in two cycles.

Therefore it is advantageous to ensure that compilers or programmers writing assembly code schedule these instruction pairs sequentially.

4.14 PC-relative address calculation

The Cortex®-A57 processor r1p3 and later revisions support optimized PC-relative address calculation using the following instruction sequence:

```
ADRP xX, #label
ADD xY, xX, #imm
```

If this sequence appears sequentially and in the described order in program code, the two instructions can be executed at lower latency and higher bandwidth than if they do not appear sequentially in the program code.

Therefore it is advantageous to ensure that compilers or programmers writing assembly code schedule these instruction pairs sequentially.

4.15 FPCR self-synchronization

Programmers and compiler writers should note that writes to the FPCR register are self-synchronizing, that is, their effect on subsequent instructions can be relied upon without an intervening context synchronizing operation.