Arm® Cortex®-R82 Processor

Revision: r0p2

Software Optimization Guide

Non-Confidential

Issue 1.0

PJDOC-466751330-590669

Arm® Cortex®-R82 Processor
Software Optimization Guide

Release information

Document history

Issue	Date	Confidentiality	Change
1.0	10 December 2021	Non-Confidential	First release for r0p2

Non-Confidential Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

developer.arm.com

Inclusive language commitment

Contents

1 Introduction

1.1 Product revision status

The rxpy identifier indicates the revision status of the product described in this book, for example, r1p2, where:

rx	Identifies the major revision of the product, for example, r1.
py	Identifies the minor revision or modification status of the product, for example, p2.

1.2 Intended audience

This document is for system designers, system integrators, and programmers who are designing or programming a System-on-Chip (SoC) that uses an Arm core.

1.3 Scope

This document describes elements of the Cortex-R82 micro-architecture that influence software performance so that software and compilers can be optimized accordingly.

Micro-architectural detail is limited to that which is useful for software optimization.

Documentation extends only to software visible behavior of the Cortex-R82 core and not to the hardware rationale behind the behavior.

1.4 Conventions

The following subsections describe conventions used in Arm documents.

1.4.1 Glossary

The Arm Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: **https://developer.arm.com/glossary**.

1.4.2 Terms and Abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AGU	Address Generation Unit
ALU	Arithmetic and Logical Unit
ASIMD	Advanced SIMD
DIV	Divide
MAC	Multiply-Accumulate
SQRT	Square Root
FP	Floating-point
PAC	Pointer-Authentication

1.4.3 Typographical conventions

Convention	Use
italic	Introduces citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace` **`bold`**	Denotes language keywords when used outside example code.
`monospace underline`	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
<and>	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Used in body text for a few terms that have specific technical meanings, that are defined in the Arm® Glossary. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.
⚠ Caution	This represents a recommendation which, if not followed, might lead to system failure or damage.
⚠ Warning	This represents a requirement for the system that, if not followed, might result in system failure or damage.
⚠ Danger	This represents a requirement for the system that, if not followed, will result in system failure or damage.

Convention	Use
Note	This represents an important piece of information that needs your attention.
Tip	This represents a useful tip that might make it easier, better or faster to perform a task.
Remember	This is a reminder of something important that relates to the information you are reading.

1.5 Additional reading

This document contains information that is specific to this product. See the following documents for other relevant information:

Table 1-1: Arm publications

Document name	Document ID	Licensee only
Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile	DDI 0487	No
Arm® Architecture Reference Manual Supplement Armv8, for Armv8-R AArch64 Architecture Profile	DDI 0600	No
Arm® Cortex®-R82 Processor Technical Reference Manual	101548	No
Arm® Cortex®-R82 Processor Configuration and Integration Manual	101549	Yes

1.6 Feedback

Arm welcomes feedback on this product and its documentation.

1.6.1 Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

1.6.2 Feedback on content

If you have comments on content, send an email to errata@arm.com and give:

- The title Arm® Cortex®-R82 Processor Software Optimization Guide.
- The number PJDOC-466751330-590669.
- If viewing a PDF version of a document, the page number(s) to which your comments refer.
- If viewing online, the topic names to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader and cannot guarantee the quality of the represented document when used with any other PDF reader.

---

2 Pipeline

The Cortex-R82 core is a mid-range, low-power core that implements the Armv8-R64 architecture with support for all the mandatory features up to v8.4.

2.1 Pipeline overview

The Cortex-R82 pipeline is 8-stages deep for integer instructions and 10-stages deep for *floating-point* (FP) and *Advanced SIMD* (ASIMD) instructions.

The Advanced SIMD architecture, its associated implementations, and supporting software, are also referred to as NEON™ technology.

The following figure shows the structure of the datapath.

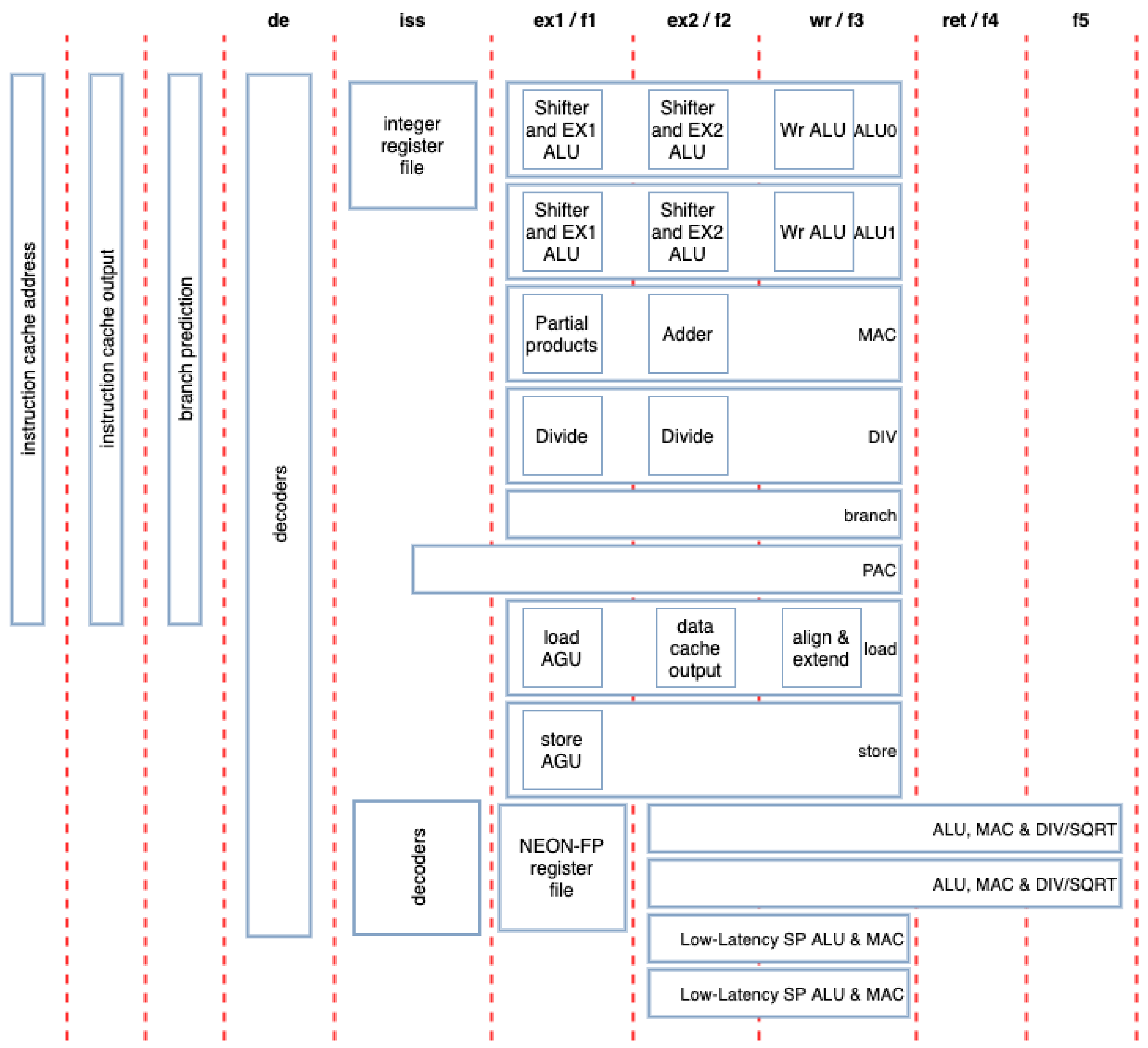

Figure 1 Cortex-R82 pipeline

The pipeline stages in the main datapath are *iss*, *ex1*, *ex2*, *wr*, and *ret*.

The pipeline stages in the NEON-FP datapath are *f1*, *f2*, *f3*, *f4*, and *f5*.

Integer instructions are issued in-order from the *iss* (issue) pipeline stage and complete in the *wr* (writeback) pipeline stage.

Floating-point or NEON instructions read their operands in the *f1* pipeline stage and normally complete in the *f5* pipeline stage.

The length of pipeline stages is equal in the main datapath and in the NEON-FP datapath. Result is written back to the register bank at the end of the pipeline stages, but forwarding paths are available for most pipelines.

2.1.1 Forwarding paths

Forwarding paths are implemented between almost all integer pipeline stages where operands can be consumed.

For example:

- The *Arithmetic and Logical Unit* (ALU) pipeline can forward results from the end of *ex1*, *ex2* and *wr* to earlier stages of both ALU pipelines, and to the *iss* stage of the *Divide* (DIV) and *Multiply-Accumulate* (MAC) pipelines and the load/store *Address Generation Units* (AGUs). There are also dedicated forwarding paths from the *ex1* stage of the ALU0 to the *ex1* stage of the ALU1, *ex2* stage of the ALU0 to the *ex2* stage of the ALU1, and from the *ex2* stage of the ALU0 or ALU1 to the *ex2* stage of the store pipeline.
- The DIV pipeline can only accept operands in the *iss* stage and will only forward results from the *wr* stage.
- The MAC pipeline can only accept multiply operands in the *iss* stage and the accumulate operand in the *ex2* stage. There is a dedicated forwarding path from the *wr* stage to the *ex2* stage for accumulator forwarding within the MAC pipeline. Multiply and MAC results can be forwarded from the *ex2* stage.
- The PAC pipeline can only accept operands in the *iss* stage and cannot forward its results.
- Load-Store instructions require their address operands at the *iss* stage. There is a dedicated forwarding path to forward the address result back to the AGU base operand. There is also a dedicated forwarding path to support pointer-chasing of a load data at *ex2* or *wr* to AGU base operand at *ex1*.
- Except for system register read results, pointer authentication results and the branch and link (BLR) register result, all integer results can be forwarded from the *wr* stage.

Forwarding does not contain bubbles so if a result can be forwarded from the end of the *ex1* stage it can also be forwarded from the *ex2* and *wr* stages. Similarly, if the latest a result can be consumed in *ex2*, it can also be consumed in *ex1* or *iss* if the result is available earlier.

There are two unified pipelines in the FP-NEON datapath. Forwarding in the FP-NEON pipelines is available from *f3*, *f4* and *f5*.

2.2 Multi-issue

The Cortex-R82 core triple-issues under most circumstances. An outline of the rules required to achieve triple-issue are described in the following two tables. In these tables, instruction-0 is the instruction that would otherwise be single-issued (also known as the older instruction) and instruction-1/2 can only dual/triple-issue if instruction-0 also supports multi-issue.

Table 2 Instruction-0 multi-issue conditions

Instruction groups	Instructions	Notes
Data-processing	All integer data-processing instructions (including flag setting instructions) can be multi issued.	-
Load/store	All load/store instructions can be multi issued except: o Atomic LD/ST operations o Atomic compare swap pair. o Load/Store exclusive o Special cases detailed in sections 4.7 - 4.9.	-
Floating-point	All floating-point instructions can be multi issued	-
Advanced SIMD	All Advanced SIMD instructions can be multi issued	-
Branches	Most branches can be multi issued from this position except for branches with fused PAC operations.	-

Instruction groups	Instructions	Notes
Miscellaneous	Generally, control instructions cannot be multi issued. These include **`MRS/MSR, WFI, WFE`**, and barriers. To improve context switching latency, the following register accesses can multi-issue from this position: o MRS/MSR PRBAR_EL1/PRBAR_EL2 o MRS ELR_EL1/ELR_EL2 o MRS SPSR_EL1/SPSR_EL2 o MRS ICC_IAR0_EL1/ICC_IAR1_EL1	1

Table 3 Instruction-1/2 multi-issue conditions

Instruction groups	Instructions	Notes
Data-processing	All data-processing instructions (including flag setting instructions) can be multi issued except: o Divide instructions. o Special cases detailed in sections 4.3 - 4.6.	-
Load/store	All load/store instructions can be multi issued except: o Load/store multiple instructions. o Some load pair instructions. o Atomic LD/ST operations o Atomic compare swap pair. o Fused PAC-LD operations o Special cases detailed in sections 4.7 - 4.9.	Providing there is not a structural hazard (loads cannot be dual issued with loads, and stores cannot be dual issued with stores)
Floating-point	Most floating-point instructions can be multi issued from these positions except: o Special cases detailed in sections 4.10 - 4.13.	-
Advanced SIMD	Most data-processing Advanced SIMD instructions can be multi issued from these positions except: o Special cases detailed in sections 4.14 - 4.18.	-
Branches	All branches can be multi issued from these positions.	Providing there is not a structural hazard (branches cannot be dual issued with branches)
Conditional	Conditional (flag-dependent) instructions can be multi issued with a flag setting instruction-0 except: o Instructions that execute an RRX operation. o Arithmetic with carry instructions. o Instruction-0 is **`MULS/MLAS`**. Instruction-0 is a NEON instruction.	-
Miscellaneous	To improve context switching latency, the following register accesses can dual-issue: o MRS/MSR PRLAR_EL1/PRLAR_EL2	1

Notes:

1. A PRLAR access can only be dual issued if there is a similar access to the equivalent PRBAR register in instruction-0 satisfying the following constraints:
    a. same access type (MSR/MRS)
    b. same register version (EL1/EL2)
    c. same register number (<n>)

2.3 Load/store and address generation

The Cortex-R82 load/store pipeline supports reads of up to 128 bits wide and writes of up to 128 bits wide. Providing the memory address is aligned, this allows instructions such as the A64 LDP and `STP` to be issued in a single cycle and occupy only one stage as the instruction passes through the pipeline.

The alignment requirements for load/store instructions to avoid a performance penalty are:

1. 8-bit loads/stores: Never a penalty cycle.
2. 16-bit, 32-bit, 64-bit loads/stores: Address must not cross a 128-bit boundary.
3. 128-bit loads/stores: Address must be 128-bit aligned.

If the memory address is not aligned, then providing the instruction passes its alignment checks a penalty cycle is incurred.

Load instructions normally return data from *wr* but some load instructions can return data from *ex2* if aligned and little endian. The details of which load instructions can return early result can be found in Section 4.7.

The load-use latency from the data of a load instruction to the ALU of a dependent non-shift datapath instruction is one cycle. The load-use latency from the data of a load instruction to the ALU of a dependent shift datapath instruction is two cycles or one cycle if the early load result is returned.

The first stage (*ex1*) of both the load and store pipeline contains an AGU. To lower the latency on pointer chasing operations, load data from a limited set of load instructions can be forwarded from the beginning of the *wr* pipeline stage or the end of the *ex2* stage if aligned to either the load or store AGU base operand. In general, this is limited to load instructions that do not require sign/zero extension, but more detail is provided in the following table.

Table 4 AGU pointer chasing

Load instruction	Limitation
64-bit LDR, LDUR & LDTR	64-bit aligned addresses (little endian)
64-bit LDP	128-bit aligned addresses and only the first register of the pair (little endian)

Load instructions that do not have a low-latency path in to the AGUs for pointer chasing incur an extra cycle penalty.

Finally, the Cortex-R82 AGUs can calculate the address of all A64 load/store instructions in a single cycle. Base register updates are completed in parallel with the load operation and there is no penalty to the following use of the base register writeback result.

2.4 Integer divide and multiply-accumulate units

The Cortex-R82 core contains an integer divide unit for executing the `UDIV` and `SDIV` instructions. Integer divide instructions are serializing and do not allow younger instructions to retire underneath to ensure that the integer divide results is retired in-order. The divide iteration will terminate as soon as the result has been calculated.

The MAC unit in the Cortex-R82 core can sustain one 32-bit x 32-bit multiply or MAC operation per-cycle. There is a dedicated forwarding path in the accumulate portion of the unit that allows the result of one MAC operation to be used as the accumulate operand of a following MAC operation with no interlock.

The latency for integer divide and multiply instructions are:

- 32-bit multiplies take one cycle.
- 64-bit multiplies take three or four cycles, depending on whether both operands contain '1's in their top 32 bits.
- SMULH/UMULH takes four cycles.
- 32-bit divides take up to 12 cycles.
- 64-bit divides take up to 20 cycles.

2.5 Pointer Authentication instructions

The Cortex-R82 contains one pointer authentication unit for executing PAC instructions. It can sustain one new PAC operation per-cycle, provided all the inflight PAC instructions use the same authentication key. This means that back-to-back instructions will have to use the same key type (Instruction/Data/Generic) and the same key number (A/B). The strip operations are excluded from this constraint and can fully issue 1 per-cycle, regardless of the unit state.

The unit requires the source operands to be ready in *iss*. Adding a new PAC code (PAC* instructions) and striping an existing one (XPAC*) generate a result in *wr*, which cannot be forwarded. Authentication operations (AUT* instructions and any fused forms) require an extra cycle and will generate the result effectively in *ret*.

2.6 Floating-point and NEON instructions

2.6.1 Instructions with out-of-order completion

While the Cortex-R82 core only issues instructions in-order, due to the number of cycles required to complete more complex floating-point and NEON instructions, out-of-order retire is allowed on the instructions described in this section. The nature of the Cortex-R82 microarchitecture is such that NEON and floating-point instructions of the same type have the same timing characteristics.

The out-of-order instructions are detailed in the following table.

Table 5 Out-of-order FP/NEON instruction characteristics

Instructions (FP or NEON)	FP/ASIMD (half-precision)		FP/ASIMD (single-precision)		FP/ASIMD (double-precision)	
	Hazard	Latency	Hazard	Latency	Hazard	Latency
FDIV	5	8	10	13	19	22
FSQRT	5	8	9	12	19	22

The following information describes how to decode the information in the table:

- *Hazard (structural):* The number of cycles that the datapath resource is unavailable to another instruction that wants to use it. For example:
    - A `FDIV` instruction after a `FSQRT` instruction must wait for the datapath resource to free up.
    - A FMLA instruction after a `FSQRT` instruction is not blocked, as `FMLA` does not use the divide or square-root resource.
    - A `FSQRT` after a `FMLA` would be able to issue immediately, provided there was no previous op using the divide or square-root resource, as a value of 1-cycle indicates that the resource will not block and supports single cycle back-to-back operation.
- *Latency:* The maximum number of cycles between when the operands are required, and the result is available for forwarding.
- The hazard and latency values shown in the table are for normal inputs. Each denormal input operand adds an additional hazard and latency cycle.

3 Instruction characteristics

3.1 Instruction tables

This chapter describes high-level performance characteristics for most Armv8-A A64 instructions. A series of tables summarize the effective execution latency and throughput (instruction bandwidth per cycle), pipelines utilized, and special behaviors associated with each group of instructions. Multi-issue corresponds to the execution pipelines described in chapter 3.

In the following tables:

- *Exec latency*, unless otherwise specified, 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 per cycle) of the specified instruction group that can be achieved in the entirety of the Cortex-R82 microarchitecture.
- *Multi-issue* is interpreted as:
  - 00 not multi-issuable.
  - 01 multi-issuable from slot 0.
  - 10 multi-issuable from the last slot which can be slot 1 or slot 2.
  - 11 multi-issuable from all slots.

3.2 Branch instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Branch, immed	`B`	1	1	11	-
Branch, register	`BR, RET`	1	1	10	-
Branch and link, immed	`BL`	1	1	11	-
Branch and link, register	`BLR`	4	1	11	1
Branch conditionally	`B.cond`	1	1	11	-
Compare and branch	`CBZ, CBNZ, TBZ, TBNZ`	1	1	11	-

Notes:

1. The Link Register result cannot be forwarded from

3.3 Arithmetic and logical instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ALU, basic, include flag setting	`ADD{S}, ADC{S}, AND{S}, BIC{S}, EON, EOR, ORN, ORR, SUB{S}, SBC{S}, NGC{S}, TST, CMN, CMP, SETF8, SETF16, RMIF`	1	2	11	-
ALU, extend and/or shift	`ADD{S}, AND{S}, BIC{S}, EON, EOR, ORN, ORR, SUB{S}, TST, CMN, CMP, NEGS`	2	2	11	-
ALU, Conditional compare	`CCMN, CCMP`	1	2	11	-
ALU, Conditional select	`CSEL, CSINC, CSINV, CSNEG, CSET, CSETM`	1	2	11	-

3.4 Move and shift instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Address generation	`ADR, ADRP`	1	2	11	-
Move immed	`MOVN, MOVK, MOVZ`	1	2	11	-
Move register	`MOV`	1	2	11	1
MVN, no shift	`MVN`	1	2	11	1
Arithmetic/Logical shift	`ASR, LSL, LSR`	1	2	11	1
Variable shift	`ASRV, LSLV, LSRV, RORV`	2	2	11	-

Notes:

1. Alias of general forms but with lower latency

3.5 Divide and multiply instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Divide, W-form	`SDIV, UDIV`	`3 - 12 (11)`	`1/12 (11) - 1/3`	01	1
Divide, X-form	`SDIV, UDIV`	`3 - 20 (19)`	`1/20 (19) - 1/3`	01	1
Multiply accumulate (32-bit)	`MADD, MSUB, MUL, MNEG`	`2 (1)`	1	11	2

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Multiply accumulate (64-bit)	`MADD, MSUB, MUL, MNEG`	4 (3)	1/3 (1/2)	11	2
Multiply accumulate long	`SMADDL, SMSUBL, UMADDL, UMSUBL, SMULL, UMULL`	2 (1)	1	11	2
Multiply high	`SMULH, UMULH`	5	1/4	11	-

Notes:

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. Signed division takes one more cycle than unsigned division for non-zero division. The latency for the unsigned division is shown inside the parentheses.
2. There is a dedicated forwarding path in the accumulate portion of the unit that allows the result of one MAC operation to be used as the accumulate operand of a following MAC operation with no interlock.

3.6 Miscellaneous Data-processing instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Bitfield extract	`EXTR`	1	2	11	-
Sign/zero extend	`SXTB, SXTH, SXTW, UXTB, UXTH`	1	2	11	-
Bitfield move, basic	`SBFM, UBFM`	2	2	11	-
Bitfield move, insert	`BFM`	2	2	11	-
Count leading	`CLS, CLZ`	1	2	11	-
Reverse bits	`RBIT`	1	2	11	-
Reverse bytes	`REV, REV16, REVSH`	1	2	11	-
No Operation	`NOP`	1	3	11	-

3.7 Load instructions

- The latencies shown assume the memory access hits in the *Level 1* (L1) data cache or *Tightly Coupled Memories* (TCMs) with 0 wait states. The latency numbers shown indicate the worst-case load-use latency from the load data to a dependent instruction.
- Latencies correspond to “correctly” aligned accesses. There is one cycle penalty for unaligned loads that cross a 128-bit boundary.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Load register, literal	`LDR`	3,2 (2)	1	11	1,2

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Load register, literal	`LDRSW`	3,2	1	11	2
Load register, unscaled immed	`LDUR`	3,2 (2)	1	11	1,2
Load register, unscaled immed	`LDURB, LDURH, LDURSB, LDURSH, LDURSW`	3,2	1	11	1,2
Load register, immed, pre/post-indexed	`LDR`	3,2 (2)	1	11	1,2
Load register, immed, pre/post-indexed	`LDRB, LDRH, LDRSB, LDRSH`	3	1	11	-
Load register, immed, pre/post-indexed	`LDRSW`	3,2	1	11	2
Load register, immed unprivileged	`LDTR`	3,2 (2)	1	11	1,2
Load register, immed unprivileged	`LDTRB, LDTRH, LDTRSB, LDTRSH`	3	1	11	-
Load register, immed unprivileged	`LDTRSW`	3,2	1	11	2
Load register, unsigned immed	`LDR`	3,2 (2)	1	11	1,2
Load register, unsigned immed	`LDRB, LDRH, LDRSB, LDRSH`	3	1	11	-
Load register, unsigned immed	`LDRSW`	3,2	1	11	2
Load register, register offset	`LDR`	3,2 (2)	1	11	1,2
Load register, register offset	`LDRB, LDRH, LDRSB, LDRSH`	3	1	11	-
Load register, register offset	`LDRSW`	3,2	1	11	2
Load register, exclusive	`LDXR`	3,2	1	01	2
Load register, exclusive	`LDXRB, LDXRH`	3	1	01	-
Preload	`PRFM, PRFUM`	1	1	01	-
Load acquire	`LDAR, LDARB, LDARH`	3	1	01	-
Load acquire RCpc Register	`LDAPR, LDAPRB, LDAPRH, LDAPUR, LDAPURB, LDAPURH, LDAPURSB, LDAPURSH, LDAPURSW`	3	1	01	-
Load acquire exclusive	`LDAXR, LDAXRB, LDAXRH`	3	1	01	-
Load pair, exclusive	`LDXP`	3,2	1	01	2

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Load acquire exclusive, pair	`LDAXP`	`3`	`1`	`01`	-
Load pair, W-form, immed offset, normal	`LDP`	`3,2`	`1`	`11`	`2`
Load pair, W-form, immed offset, normal	`LDNP`	`3,2`	`1`	`01`	`2`
Load pair, X-form, immed offset, normal	`LDP`	`3,2 (2)`	`1`	`11`	`1,2`
Load pair, X-form, immed offset, normal	`LDNP`	`3,2 (2)`	`1`	`01`	`1,2`
Load pair, signed words, immed offset	`LDPSW`	`3`	`1`	`11`	-
Load pair, W-form, immed pre/post-index, normal	`LDP`	`3,2`	`1`	`01`	`2`
Load pair, X-form, immed pre/post-index, normal	`LDP`	`3,2 (2)`	`1`	`01`	`1,2`
Load pair, signed words, immed pre/post-index	`LDPSW`	`3`	`1`	`01`	-

Notes:

1. A fast forward path from the 64-bit load data to address (pointer chasing) can be activated in some cases (short latency shown in parentheses). See section 3.3.
2. Some load instructions can return data early in *ex2* with shorter load-use latency (update latency shown after comma). See section 3.3.

3.8 Store instructions

The following table describes performance characteristics for standard store instructions. Stores may issue to L1 at *iss* once their address operands are available and do not need to wait for data operands (which are required at *wr*). Once executed, stores are buffered and committed in the background.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Store register, unscaled immed	`STUR, STURB, STURH`	`1`	`1`	`11`	-
Store register, immed pre/post-index	`STR, STRB, STRH`	`1`	`1`	`11`	-
Store register, immed unprivileged	`STTR, STTRB, STTRH`	`1`	`1`	`11`	-
Store register, unsigned immed	`STR, STRB, STRH`	`1`	`1`	`11`	-
Store register, register offset	`STR, STRB, STRH`	`1`	`1`	`11`	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Store release	STLR, STLRB, STLRH, STLLR, STLLRB, STLLRH	2	1/2	01	-
Store exclusive	STXR, STXRB, STXRH	3	1	01	1
Store release exclusive	STLXR, STLXRB, STLXRH, STLXP	4	1/2	01	1
Store pair, immed, all addressing modes	STP, STNP	1	1	11	-
Store exclusive pair	STXP	3	1	01	1

Notes:

1. Latency number refers to the worst-case latency from the result to a dependent instruction.

3.9 Atomic instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
LD<OP>, without release semantics	LDADD{A}, LDADD{A}B, LDADD{A}H, LDCLR{A}, LDCLR{A}B, LDCLR{A}H, LDEOR{A}, LDEOR{A}B, LDEOR{A}H, LDSET{A}, LDSET{A}B, LDSET{A}H, LDSMAX{A}, LDSMAX{A}B, LDSMAX{A}H, LDSMIN{A}, LDSMIN{A}B, LDSMIN{A}H, LDUMAX{A}, LDUMAX{A}B, LDUMAX{A}H, LDUMIN{A}, LDUMIN{A}B, LDUMIN{A}H	4	1/2	00	-
LD<OP>, with release semantics	LDADDL, LDADDLB, LDADDLH, LDCLRL, LDCLRLB, LDCLRLH, LDEORL, LDEORLB, LDEORLH, LDSETL, LDSETLB, LDSETLH, LDSMAXL, LDSMAXLB, LDSMAXLH, LDSMINL, LDSMINLB, LDSMINLH, LDUMAXL, LDUMAXLB, LDUMAXLH, LDUMINL, LDUMINLB, LDUMINLH, LDADDAL, LDADDALB, LDADDALH, LDCLRAL, LDCLRALB, LDCLRALH, LDEORAL, LDEORALB, LDEORALH, LDSETAL, LDSETALB, LDSETALH, LDSMAXAL, LDSMAXALB, LDSMAXALH, LDSMINAL, LDSMINALB, LDSMINALH, LDUMAXAL, LDUMAXALB, LDUMAXALH, LDUMINAL, LDUMINALB, LDUMINALH	5	1/3	00	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ST<OP>, without release semantics	STADD{A}, STADD{A}B, STADD{A}H, STCLR{A}, STCLR{A}B, STCLR{A}H, STEOR{A}, STEOR{A}B, STEOR{A}H, STSET{A}, STSET{A}B, STSET{A}H, STSMAX{A}, STSMAX{A}B, STSMAX{A}H, STSMIN{A}, STSMIN{A}B, STSMIN{A}H, STUMAX{A}, STUMAX{A}B, STUMAX{A}H, STUMIN{A}, STUMIN{A}B, STUMIN{A}H	1	1	00	-
ST<OP>, with release semantics	STADDL, STADDLB, STADDLH, STCLRL, STCLRLB, STCLRLH, STEORL, STEORLB, STEORLH, STSETL, STSETLB, STSETLH, STSMAXL, STSMAXLB, STSMAXLH, STSMINL, STSMINLB, STSMINLH, STUMAXL, STUMAXLB, STUMAXLH, STUMINL, STUMINLB, STUMINLH, STADDAL, STADDALB, STADDALH, STCLRAL, STCLRALB, STCLRALH, STEORAL, STEORALB, STEORALH, STSETAL, STSETALB, STSETALH, STSMAXAL, STSMAXALB, STSMAXALH, STSMINAL, STSMINALB, STSMINALH, STUMAXAL, STUMAXALB, STUMAXALH, STUMINAL, STUMINALB, STUMINALH	2	1/2	00	-
Compare and swap, without release semantics	CAS{A}, CAS{A}B, CAS{A}H	5	1/3	00	-
Compare and swap, with release semantics	CASL, CASLB, CASLH, CASAL, CASALB, CASALH	6	1/4	00	-
Compare and swap, pair, without release semantics	CASP{A}	5	1/3	00	-
Compare and swap, pair, with release semantics	CASPL, CASPAL	6	1/4	00	-
Swap, without release semantics	SWP{A}, SWP{A}B, SWP{A}H	4	1/2	00	-
Swap, with release semantics	SWPL, SWPLB, SWPLH, SWPAL, SWPALB, SWPALH	5	1/3	00	-

3.10 Floating-point data processing instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
FP absolute value	FABS	4	2	11	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
FP arithmetic	`FADD, FSUB`	4	2	11	-
FP compare	`FCCMP{E}, FCMP{E}`	2	1	11	-
FP divide, H-form	`FDIV`	8	1/5	01	1
FP divide, S-form	`FDIV`	13	1/10	01	1
FP divide, D-form	`FDIV`	22	1/19	01	1
FP min/max	`FMIN, FMINNM, FMAX, FMAXNM`	2	2	11	-
FP multiply	`FMUL, FNMUL`	4	2	11	-
FP multiply accumulate	`FMADD, FMSUB, FNMADD, FNMSUB`	4	2	11	-
FP negate	`FNEG`	2	2	11	-
FP round to integral	`FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ`	4	2	11	-
FP select	`FCSEL`	2	1	01	-
FP square root, H-form	`FSQRT`	8	1/5	01	1
FP square root, S-form	`FSQRT`	12	1/9	01	1
FP square root, D-form	`FSQRT`	22	1/19	01	1

Notes:

1. Refer to section 2.6.1 for details.

3.11 Floating-point miscellaneous instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
FP convert, from vec to vec reg	`FCVT`	4	2	11	-
FP convert, from vec to vec reg	`FCVTXN`	4	2	11	-
FP convert, from gen to vec reg	`SCVTF, UCVTF`	4	2	11	1

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
FP convert, from vec to gen reg	`FCVTAS`, `FCVTAU`, `FCVTMS`, `FCVTMU`, `FCVTNS`, `FCVTNU`, `FCVTPS`, `FCVTPU`, `FCVTZS`, `FCVTZU`	3	2	11	1
FP move, immed	`FMOV`	2	2	11	1
FP move, register	`FMOV`	1	2	11	-
FP transfer, from gen to half/double/single	`FMOV`	2	2	11	1
FP transfer, from double/single to gen reg	`FMOV`	3	2	11	1
FP transfer, from half to gen reg	`FMOV`	3	2	11	1

Notes:

1. Latency number refers to the worst-case latency from the result to a dependent instruction.

3.12 Floating-point load instructions

FP load data is available for forwarding from *f4* or *f3* if they are simple loads. The latency numbers shown indicate the worst-case load-use latency from the load data to a dependent instruction. Latencies assume the memory access hits in the L1 data cache. Latencies also assume that 64-bit element loads are aligned to 64-bit. If this is not the case, one extra cycle is required.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Load vector reg, literal, S/D-form	`LDR`	2	1	11	-
Load vector reg, literal, Q-form	`LDR`	2	1	01	-
Load vector reg, unscaled immed, B/H-form	`LDUR`	3	1	01	-
Load vector reg, unscaled immed, S/D-form	`LDUR`	2	1	11	-
Load vector reg, unscaled immed, Q-form	`LDUR`	2	1	01	-
Load vector reg, immed pre/post-index, B/H-form	`LDR`	3	1	01	-
Load vector reg, immed pre/post-index, S/D-form	`LDR`	2	1	11	-
Load vector reg, immed pre/post-index, Q-form	`LDR`	2	1	01	-
Load vector reg, unsigned immed / register offset, B/H-form	`LDR`	3	1	01	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Load vector reg, unsigned immed / register offset, S/D-form	`LDR`	2	1	11	-
Load vector reg, unsigned immed / register offset, Q-form	`LDR`	3	1	01	-
Load vector pair, immed offset, S/D-form	`LDP, LDNP`	2	1	01	-
Load vector pair, immed offset, Q-form	`LDP, LDNP`	3	1/2	01	-
Load vector pair, immed pre/post-index, S/D-form	`LDP`	2	1	01	-
Load vector pair, immed pre/post-index, Q-form	`LDP`	3	1/2	01	-

3.13 Floating-point store instructions

Stores may issue to L1 at *iss* once their address operands are available and do not need to wait for data operands (which are required at *f2*). Once executed, stores are buffered and committed in the background.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Store vector reg, unscaled immed	`STUR`	1	1	11	-
Store vector reg, immed	`STR`	1	1	11	-
Store vector reg, register offset	`STR`	1	1	11	-
Store vector pair, immed, S/D-form	`STP, STNP`	1	1	11	-
Store vector pair, immed, Q-form	`STP, STNP`	2	1/2	01	-

3.14 Advanced SIMD integer instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD absolute diff	`SABD, UABD`	3	2 (1)	11	1
ASIMD absolute diff accum	`SABA, UABA, SABAL(2), UABAL(2)`	4	1/2	01	-
ASIMD absolute diff long	`SABDL(2), UABDL(2)`	3	1	11	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD arith	`ADD, SUB, NEG, SHADD, UHADD, SHSUB, UHSUB, SRHADD, URHADD`	2	2 (1)	11	1
ASIMD arith	`ABS, ADDP, SADDLP, UADDLP, SQADD, UQADD, SQNEG, SQSUB, UQSUB, SUQADD, USQADD`	3	2 (1)	11	1
ASIMD arith	`SADDL(2), UADDL(2), SADDW(2), UADDW(2), SSUBL(2), USUBL(2), SSUBW(2), USUBW(2)`	3	1	11	-
ASIMD arith	`ADDHN(2), SUBHN(2)`	3	1	01	-
ASIMD arith	`SQABS`	4	2 (1)	11	1
ASIMD arith	`RADDHN(2), RSUBHN(2)`	4	1/2	01	-
ASIMD arith, reduce	`ADDV, SADDLV, UADDLV`	3	1	01	-
ASIMD compare	`CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE, CMLT`	2	2 (1)	11	1
ASIMD compare	`CMTST`	3	2 (1)	11	1
ASIMD logical	`AND, BIC, EOR, MVNI, ORN, ORR, NOT`	1	2 (1)	11	1
ASIMD max/min, basic	`SMAX, SMAXP, SMIN, SMINP, UMAX, UMAXP, UMIN, UMINP`	2	2 (1)	11	1
ASIMD max/min, reduce	`SMAXV, SMINV, UMAXV, UMINV`	4	1	01	-
ASIMD multiply	`MUL, SQDMULH, SQRDMULH`	4	2 (1)	11	1
ASIMD multiply	`PMUL`	3	2 (1)	11	1
ASIMD multiply accumulate	`MLA, MLS`	4 (1)	2 (1)	11	1,2
ASIMD multiply accumulate high half	`SQRDMLAH, SQRDMLSH`	4	2 (1)	11	1
ASIMD multiply accumulate long	`SMLAL(2), SMLSL(2), UMLAL(2), UMLSL(2)`	4 (1)	1	11	2
ASIMD multiply accumulate long	`SQDMLAL(2), SQDMLSL(2)`	4	1	11	-
ASIMD multiply accumulate long	`SQDMLAL, SQDMLSL`	4	2	11	

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD dot product	`UDOT, SDOT`	4 (1)	2 (1)	11	1,3
ASIMD multiply long	`SMULL, SMULL(2), UMULL, UMULL(2), SQDMULL, SQDMULL(2)`	4	2 (1)	11	1
ASIMD polynomial (8x8) multiply long	`PMULL(2)`	3	1	11	4
ASIMD pairwise add and accumulate	`SADALP, UADALP`	4	1/2	01	-
ASIMD shift accumulate	`SSRA, USRA`	3	2 (1)	11	1
ASIMD shift round accumulate	`SRSRA, URSRA`	4	1/2	01	-
ASIMD shift by immed	`SHL, SHRN(2), SSHR, USHR`	2	2 (1)	11	1
ASIMD shift by immed and insert	`SLI, SRI`	2	2 (1)	11	1
ASIMD shift by immed	`SHLL(2), SSHLL(2), USHLL(2), SXTL(2), UXTL(2)`	2	1	11	-
ASIMD shift by immed	`RSHRN(2), SRSHR, URSHR, SQSHRN, SQSHRN(2), UQSHRN, UQSHRN(2)`	3	2 (1)	11	1
ASIMD shift by immed	`SQSHL{U}, UQSHL, SQRSHRN(2), UQRSHRN(2), SQRSHRUN(2), SQSHRUN(2)`	4	2 (1)	11	1
ASIMD shift by register	`SSHL, USHL`	2	2 (1)	11	1
ASIMD shift by register	`SRSHL, URSHL`	3	2 (1)	11	1
ASIMD shift by register	`SQSHL, UQSHL, SQRSHL, UQRSHL`	4	2 (1)	11	1

Notes:

1. If the instruction has Q-form, the execution throughput is `1`. The throughput for the Q-form of the instruction is shown in parentheses.
2. Multiply-accumulate pipelines support forwarding of accumulate operands from similar instructions, allowing a typical sequence of integer multiply-accumulate instructions to issue every cycle (accumulate latency shown in parentheses).
3. Multiply-accumulate pipelines support forwarding of accumulate operands between Dot Product instructions, allowing a sequence of Dot Product instructions to issue every cycle (accumulate latency shown in parentheses).
4. This category includes instructions of the form "`PMULL Vd.8H, Vn.8B`, `Vm.8B`" and "`PMULL2 Vd.8H`, `Vn.16B`, `Vm.16B`".

3.15 Advanced SIMD floating-point instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD FP arith	`FABS, FABD, FADD, FSUB, FADDP`	4	2 (1)	11	1
ASIMD FP compare	`FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT`	2	2 (1)	11	1
ASIMD FP convert, long	`FCVTL(2)`	4	1	11	-
ASIMD FP convert, narrow	`FCVTN(2), FCVTXN(2)`	4	1	01	-
ASIMD FP convert, other	`FCVTAS, VCVTAU, FCVTMS, FCVTMU, FCVTNS, FCVTNU, FCVTPS, FCVTPU, FCVTZS, FCVTZU, SCVTF, UCVTF`	4	2 (1)	11	1
ASIMD FP divide, H-form	`FDIV`	8	1/5	01	2
ASIMD FP divide, S-form	`FDIV`	13	1/10	01	2
ASIMD FP divide, D-form	`FDIV`	22	1/19	01	2
ASIMD FP max/min, normal	`FMAX, FMAXNM, FMIN, FMINNM`	2	2 (1)	11	1
ASIMD FP max/min, pairwise	`FMAXP, FMAXNMP, FMINP, FMINNMP`	2	2 (1)	11	1
ASIMD FP max/min, reduce	`FMAXV, FMAXNMV, FMINV, FMINNMV`	3	1	01	-
ASIMD FP multiply	`FMUL, FMULX`	4	2 (1)	11	1
ASIMD FP multiply accumulate	`FMLA, FMLS`	4	2 (1)	11	1
ASIMD FP negate	`FNEG`	2	2 (1)	11	1
ASIMD FP round	`FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ`	4	2 (1)	11	1
ASIMD FP complex	`FCADD, FCMLA`	4	1	01	1
ASIMD FP multiply and accumulate/subtract long, by element	`FMLAL, FMLAL2, FMLSL, FMLSL2`	4	2 (1)	11	1
ASIMD FP JavaScript conversion	`FJCVTZS`	3	2	11	1

Notes:

1. If the instruction has Q-form, the execution throughput is 1. The throughput for the Q-form of the instruction is shown in parentheses.
2. Refer to section 3.5.1 for more details.

3.16 Advanced SIMD miscellaneous instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD bit reverse	`RBIT`	2	2 (1)	11	1
ASIMD bitwise insert	`BIF, BIT, BSL`	2	2 (1)	11	1
ASIMD count	`CLZ, CNT`	2	2 (1)	11	1
ASIMD count	`CLS`	3	2 (1)	11	1
ASIMD duplicate, gen reg	`DUP`	3	2	11	3
ASIMD duplicate, element	`DUP`	2	2 (1)	11	1
ASIMD extract	`EXT`	2	2 (1)	11	1
ASIMD extract narrow	`XTN(2)`	2	2	11	-
ASIMD extract narrow, saturating	`SQXTN(2), SQXTUN(2), UQXTN(2)`	4	2	11	-
ASIMD insert, element to element	`INS`	2	2	11	-
ASIMD move, integer immed	`MOVI`	2	2 (1)	11	1,3
ASIMD move, FP immed	`FMOV`	2	2	11	3
ASIMD reciprocal estimate	`FRECPE, FRECPX, FRSQRTE, URECPE, URSQRTE`	4	2 (1)	11	1
ASIMD reciprocal step	`FRECPS, FRSQRTS`	4	2 (1)	11	1
ASIMD reverse	`REV16, REV32, REV64`	2	2 (1)	11	1
ASIMD table lookup	`TBL`	2+N-1	1/N	01	2
ASIMD table lookup	`TBX`	2+N	1/(N+1)	01	2
ASIMD transfer, element to gen reg	`SMOV, UMOV`	2	2	11	-
ASIMD transfer, gen reg to element	`INS`	2	2	11	3
ASIMD transpose, 64-bit (.2D)	`TRN1, TRN2`	2	1	11	-
ASIMD transpose, other	`TRN1, TRN2`	2	2 (1)	11	1
ASIMD unzip/zip	`UZP1, UZP2, ZIP1, ZIP2`	2	2 (1)	11	1

Notes:

1. If the instruction has Q-form, the execution throughput is 1. The throughput for the Q-form of the instruction is shown in parentheses.
2. For table branches (`TBL` and `TBX`), `N` denotes the number of registers in the table

3. Latency number refers to the worst-case latency from the result to a dependent instruction

3.17 Advanced SIMD load instructions

Advanced SIMD load data can be available for forwarding from *f4 or f3* for simple loads. The latency numbers shown indicate the worst-case load-use latency from the load data to a dependent instruction. The latencies shown assume the memory access hits in the L1 data cache.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD load, 1 element, multiple, 1 reg, D-form	`LD1`	2	1	11	-
ASIMD load, 1 element, multiple, 1 reg, Q-form	`LD1`	2	1	01	-
ASIMD load, 1 element, multiple, 2 reg, D-form	`LD1`	2	1	01	-
ASIMD load, 1 element, multiple, 2 reg, Q-form	`LD1`	4	1/2	01	-
ASIMD load, 1 element, multiple, 3 reg, D-form	`LD1`	4	1/2	01	-
ASIMD load, 1 element, multiple, 3 reg, Q-form	`LD1`	5	1/3	01	-
ASIMD load, 1 element, multiple, 4 reg, D-form	`LD1`	4	1/2	01	-
ASIMD load, 1 element, multiple, 4 reg, Q-form	`LD1`	6	1/4	01	-
ASIMD load, 1 element, one lane, B/S/H	`LD1`	3	1	01	-
ASIMD load, 1 element, one lane, D	`LD1`	2	1	01	-
ASIMD load, 1 element, all lanes	`LD1R`	3	1	01	-
ASIMD load, 2 elements, multiple, D-form	`LD2`	3	1	01	-
ASIMD load, 2 elements, multiple, Q-form	`LD2`	4	1/2	01	-
ASIMD load, 2 elements, one lane, B/H/S	`LD2`	3	1	01	-
ASIMD load, 2 elements, one lane, D	`LD2`	2	1	01	-
ASIMD load, 2 elements, all lanes	`LD2R`	3	1	01	-
ASIMD load, 3 elements, multiple, D-form, B/H/S	`LD3`	5	1/3	01	-
ASIMD load, 3 elements, multiple, Q-form, B/H/S	`LD3`	6	1/4	01	-
ASIMD load, 3 elements, multiple, Q-form, D	`LD3`	5	1/3	01	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD load, 3 elements, one lane	LD3	4	1/2	01	–
ASIMD load, 3 elements, all lanes	LD3R	4	1/2	01	–
ASIMD load, 4 elements, multiple, D-form, B/H/S	LD4	5	1/3	01	–
ASIMD load, 4 elements, multiple, Q-form, B/H/S	LD4	7	1/5	01	–
ASIMD load, 4 elements, multiple, Q-form, D	LD4	6	1/4	01	–
ASIMD load, 4 elements, one lane	LD4	4	1/2	01	–
ASIMD load, 4 elements, all lanes	LD4R	4	1/2	01	–

3.18 Advanced SIMD store instructions

Store instructions may issue once their address operands are available and do not need to wait for data operands. Once executed, stores are buffered and committed in the background.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD store, 1 element, multiple, 1 reg	ST1	1	1	11	–
ASIMD store, 1 element, multiple, 2 reg, D-form	ST1	1	1	01	–
ASIMD store, 1 element, multiple, 2 reg, Q-form	ST1	2	1/2	01	–
ASIMD store, 1 element, multiple, 3 reg, D-form	ST1	2	1/2	01	–
ASIMD store, 1 element, multiple, 3 reg, Q-form	ST1	3	1/3	01	–
ASIMD store, 1 element, multiple, 4 reg, D-form	ST1	2	1/2	01	–
ASIMD store, 1 element, multiple, 4 reg, Q-form	ST1	4	1/4	01	–
ASIMD store, 1 element, one lane	ST1	1	1	01	–
ASIMD store, 2 element, multiple, D-form	ST2	1	1	01	–
ASIMD store, 2 element, multiple, Q-form	ST2	2	1/2	01	–
ASIMD store, 2 element, one lane	ST2	1	1	01	–
ASIMD store, 3 element, multiple, D-form	ST3	3	1/3	01	–

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
ASIMD store, 3 element, multiple, Q-form	`ST3`	4	1/4	01	-
ASIMD store, 3 element, one lane	`ST3`	2	1/2	01	-
ASIMD store, 4 element, multiple, D-form, B/H/S	`ST4`	3	1/3	01	-
ASIMD store, 4 element, multiple, Q-form, B/H/S	`ST4`	5	1/5	01	-
ASIMD store, 4 element, multiple, Q-form, D	`ST4`	4	1/4	01	-
ASIMD store, 4 element, one lane, B/H	`ST4`	2	1/2	01	-
ASIMD store, pre, post, reg/unsigned offset	`STR`	1	1	11	-

3.19 Low Latency SP instructions

When the low latency single-precision pipeline is enabled, the following instructions override the previously declared ones, by producing the result already in stage *f3*, reducing the latency by 2 cycles.

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
FP convert, from gen to vec reg	`SCVTF, UCVTF`	2	2	11	1
ASIMD FP arith	`FABD, FADD, FSUB,`	2	2	11	-
ASIMD FP multiply	`FMULX`	2	2	11	-
ASIMD reciprocal step	`FRECPS, FRSQRTS`	2	2	11	-
ASIMD reciprocal estimate	`FRECPE, FRECPX, FRSQRTE`	2	2	11	-
FP multiply	`FMUL, FNMUL`	2	2	11	-
FP multiply accumulate	`FMADD, FMSUB, FNMADD, FNMSUB`	2	2	11	-
ASIMD FP convert, other	`SCVTF, UCVTF`	2	2	11	-
FP round to integral	`FRINTA, FRINTI, FRINTM, FRINTN, FRINTP, FRINTX, FRINTZ`	2	2	11	-

Notes:

1. Latency number refers to the worst-case latency from the result to a dependent instruction

3.20 Pointer Authentication instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
Branch with Link to Register, with pointer authentication	`BLRAA, BLRAB, BLRAAZ, BLRABZ`	2	1(1/5)	10	1
Branch to Register, with pointer authentication	`BRAAZ, BRABZ, BRAA, BRAB`	1	1(1/5)	10	1
Return from subroutine, with pointer authentication	`RETAA, RETAB`	1	1(1/5)	00	1
Load Register, with pointer authentication	`LDRAA, LDRAB`	4	1(1/5)	01	1
Pointer Authentication Code for Data address, using key A/B	`PACDA, PACDZA, PACDB, PACDZB`	4	1(1/4)	11	1
Pointer Authentication Code for Instruction address, using key A/B	`PACIA, PACIA1716, PACIASP, PACIAZ, PACIZA, PACIB, PACIB1716, PACIBSP, PACIBZ, PACIZB`	4	1(1/4)	11	1
Pointer Authentication Code, using Generic key	`PACGA`	4	1(1/4)	11	1
Authenticate Data address, using key A/B	`AUTDA, AUTDZA, AUTDB, AUTDZB`	5	1(1/5)	11	1
Authenticate Instruction address, using key A/B	`AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA, AUTIB, AUTIB1716, AUTIBSP, AUTIBZ, AUTIZB`	5	1(1/5)	11	1
Strip Pointer Authentication Code	`XPACD, XPACI, XPACLRI`	4	1	11	-

Notes:

1. Execution throughput is highest when the same key type (Instruction/Data/Generic) and number(A/B) is used for back-to-back instructions. If interleaving operations with different keys, execution throughput is reduced to the number denoted inside the parenthesis.

3.21 CRC instructions

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
CRC checksum ops	`CRC32B, CRC32H, CRC32CB, CRC32CH, CRC32X, CRC32CX`	2	2	11	-

Instruction group	AArch64 instructions	Exec latency	Execution throughput	Multi-issue	Notes
CRC checksum ops	`CRC32W, CRC32CW`	1	2	11	-

4 General

This section covers aspects of the micro-architecture which are not related to the pipeline or branch prediction, but will improve performance if the software is optimized accordingly.

4.1 Support for three outstanding loads

While the Cortex-R82 core does not support any instruction reordering at issue or hit-under-miss, it can support three outstanding data cache misses. Providing that three load instructions are within four pipeline stages of each other, if the first load misses the data cache the second and third can also lookup and, if they both miss, generate a request to the *Level 2* (L2) cache or main memory.

4.2 Automatic hardware-based prefetch

The Cortex-R82 core has a data prefetch mechanism that looks for cache line fetches with regular patterns and automatically starts prefetching ahead. Prefetches will end once the pattern is broken, a DSB is executed, or a WFI or WFE is executed.

For read streams the prefetcher is based on virtual addresses and so can cross page boundaries provided that the new page is still cacheable and has read permission. The prefetcher in Cortex-R82 does not train on write streams. The Cortex-R82 core is capable of tracking multiple streams in parallel. Once the prefetcher is confident in the stream, it will progressively start increasing the prefetch distance ahead of the current demand access, fetching a certain distance from the demand access into the L1 Cache, and fetching a certain farther distance from the demand access into L2 Cache. This allows better utilization of the larger resources available at L2, and also reduces the amount of pollution of the L1 cache if the stream ends or is incorrectly predicted. If the prefetching to L2 was accurate then the line will be allocated from the L2 Cache into the L1 Cache when the demand stream reaches that address.

When there is a continuous stream of writes which misses in the cache, Cortex-R82 relies on a write streaming mechanism, instead of prefetching. Write streaming mode is triggered after encountering a stream of writes to sequential cache lines. Once triggered, all further stores would then be allocated into the L2 cache, instead of the L1 cache. If the stream continues above a certain threshold, Cortex R82 will then start sending the writes outside the Cluster instead of allocating into L2.

4.3 Software load prefetch performance

The Cortex-R82 core supports all load prefetching instructions. When executed load prefetches are non-blocking so they do not stall while the data is being fetched:

- Data fetched by a `PRFM PLD/PST` instruction is placed in the cache level encoded in the instruction.
- Data fetched by a `PRFM PLI` instruction is always placed in the L2 cache

On the Cortex-R82 core it is not advisable to use explicit load prefetch instructions if the access pattern falls within the capabilities of the hardware based prefetcher since load prefetch instructions consume an issue slot.

4.4 Non-temporal loads

The Cortex-R82 core supports the Non-temporal load and store instructions in the AArch64 instruction set. Non-temporal loads will allocate the line to the L1 cache as normal. Non-temporal store instructions will update the cache if they hit, but will not cause an L1 allocation if they miss. They will allocate in L2 cache.

4.5 Cache line size

All caches in the Cortex-R82 core implement a 64-byte cache line.

4.6 MemCopy performance

As the load/store pipeline width is 128 bits, the Cortex-R82 core will provide the highest performance if the instructions used can utilize the full width of this interface. The LDP/`STP` instructions can consume all 128 bits in a single-cycle.

The Cortex-R82 core includes separate load and store pipelines, which allow it to execute a load and a store instruction in the same cycle.

To achieve maximum throughput for memory copy (or similar loops):

- 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 `LDP` and `STP`), interleaving them so that one load and one store operation can be performed each cycle.
- Integer instructions can triple-issue on top of the pairs of LDP/STP, so try to maximize this opportunity by interleaving those as well.
- Separate the load and corresponding store instruction by at least one other instruction to avoid interlocks on the store source registers.

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

```
; x0 = source pointer, aligned to 64 bytes
; x1 = destination pointer, aligned to 64 bytes
; x2 = number of bytes to copy, multiple of 64 bytes
LDP x12, x13, [x0,#0x0]
STP x12, x13, [x1,#0x0]
ADD x4, x0, x2

LDP x6, x7, [x0,#0x10]
ADD x5, x1, x2

LDP x8, x9, [x0,#0x20]
SUBS x2, x2, #8 // We work over 8 of 8 bytes elements on each iteration

LDP x10, x11, [x0,#0x30]

LDP x12, x13, [x0,#0x40]!
```

```
B.EQ copy_loop_end

copy_loop_start:
  STP x6, x7, [x1,#0x10]
  LDP x6, x7, [x0,#0x10]

  STP x8, x9, [x1,#0x20]
  LDP x8, x9, [x0,#0x20]
  SUBS x2, x2, #8 // We work over 8 of 8 bytes elements on each iteration

  STP x10, x11, [x1,#0x30]
  LDP x10, x11, [x0,#0x30]
  ADD x0, x0, #0x40

  STP x12, x13, [x1,#0x40]!
  LDP x12, x13, [x0]
  B.HI copy_loop_start

copy_loop_end:
  LDP x15, x16, [x4,#-0x40]
  STP x6, x7, [x1,#0x10]

  LDP x6, x7, [x4,#-0x30]
  STP x8, x9, [x1,#0x20]

  LDP x8, x9, [x4,#-0x20]
  STP x10, x11, [x1,#0x30]

  LDP x10, x11, [x4,#-0x10]
  STP x12, x13, [x1,#0x40]

  STP x15, x16, [x5,#-0x40]
  STP x6, x7, [x5,#-0x30]
  STP x8, x9, [x5,#-0x20]
  STP x10, x11, [x5,#-0x10]
```

4.7 A64 low latency pointer forwarding

In the A64 instruction set the following pointer sequence is expected to be common to generate load-store addresses:

```
      ADRP x0, <const>
      LDR x0, [x0, #lo12 <const>]
```

In the Cortex-R82 core there are dedicated forwarding paths that always allow this sequence to be executed without incurring a dependency-based stall.

Appendix A Revisions

This appendix describes the technical changes between released issues of this document.

Table A-1: Issue 01

Change	Location	Affects
First release for r0p2	-	r0p2