FUJITSU

A64FX®

Microarchitecture Manual

English

1.6, 30 September 2021

Revision History

Change Date	Edition	Description of Change
2/28/2020	1.0	First Release
4/28/2020	1.1	Correct typos
7/31/2020	1.2	Update following chapters and sections: ・6.5.1. SVE Instruction with Merging Predication ・7.8.1. Multiple Structures Instruction ・7.8.2. Gather Load/Scatter Store ・9.6. Zero Fill ・16. List of Instruction Attribute and Latency: the description of extra μOP and latency
10/31/2020	1.3	Update and modify a following chapter: ・16. List of Instruction Attribute and Latency: ARMv8 Base and SVE instructions
3/31/2021	1.4	Update and modify following chapter and section: ・6.5.1. SVE Instruction with Merging Predication ・16. List of Instruction Attribute and Latency: SVE instructions
6/30/2021	1.5	Update and modify following sections: ・7.8.2. Gather load / Scatter store ・11.5.2. Behavior of Stream Detect Mode ・12.2. Sector Cache Behavior
9/30/2021	1.6	Update and modify a following section: ・14.1. Instruction Mix Change following sections: ・9.2.2. L2 Cache ・9.5. Move-In Bypass ・10.2. Performance

Release Notes

Fixes

- In Table 14-2, the formula of “Base insts. excluding load/store” is updated to correct the imprecise description.

Changes

- The description of Move-In Bypass on L1D cache is removed corresponding the hardware setting change.
- Because of above, L2 cache access latency is increased by 9 cycles and the memory latency is also increased by 4.5 ns.

Contents

List of Figures

List of Tables

Preface

The purpose of this manual is to explain the A64FX processor microarchitecture and provide reference information for software tuning.

The manual was written with reference to the following documents. They define terms used in this manual without any particular annotations.

- *A64FX Specification (Scheduled to be released in Sep. 2020)*
- *ARM® Architecture Reference Manual* (ARMv8, ARMv8.1, ARMv8.2, ARMv8.3)
- *ARM® Architecture Reference Manual Supplement The Scalable Vector Extension*

Typographical and Notational Conventions

This manual uses the following notational conventions.

Assembler notation

The syntax of the assembler complies with the *ARM® Architecture Reference Manual* (called the *ARM Manual* below). All characters are written in lowercase `Consolas`.

Instruction notation

The notation for instructions basically complies with the *ARM Manual*. Characters are written in uppercase. However, characters in "List of Instruction Attribute and Latency" are written in Cambria. To express multiple instructions as a group, expanding expressions like regular expressions have been adopted for instruction names in this manual.

The notation for expanding instructions is shown below.

*	Asterisk	Expands to a character string, including any with a length of 0.
[and]	Brackets	Expands to any of the characters in "[ ]". A hyphen (-) in "[ ]" represents a character range, which means expanding to a character within the range.
{ and }	Curly brackets	Expands to one of the character strings separated by a pipe (|) in "{ }". If no character string preceded or followed in "|", that means a NULL string.

Instruction class notation

Some instructions cannot be distinguished by an instruction name alone. For example, there are seven ADD instructions. ADD (extended register), ADD (immediate), and ADD (shifted register) belong to Base Instructions. ADD (vector) belongs to SIMD&FP. ADD (immediate), ADD (vectors, predicated), and ADD (vectors, unpredicated) belong to SVE. When expressed, these instructions are modified with "( )" in accordance with the *ARM Manual*. If all the ADD instructions that are Base Instructions are expressed, a class name like "ADD (base)" is used.

Variant notation

The behavior of hardware is affected by not only instruction operations but also data types. Particularly with some SIMD&FP and SVE instructions, the behavior of hardware and the number of operations vary greatly depending on the data type, even in cases with the same instruction. Therefore, a variant modification is added as required. A variant is written immediately after a hyphen (-) placed after an expressed instruction as shown below. Basically, although the notation for variants varies depending on the instruction class, the register notation that represents data types is used.

Example:

Base instruction:	ADD (immediate) - W
SIMD&FP instruction:	FADD (scalar) - [HS]
	FADD (vector) - {8B|16B}
SVE instruction:	ADD (immediate) - [SD]

Terminology

Instruction

Term that refers to the individual instructions defined in Section C6.2, "Alphabetical list of A64 Base Instructions," in the *ARMv8 Manual* or Chapter 5, "SVE Instruction Set," in the *SVE Manual*. As with the *ARMv8 Manual*, this manual uses "( )" to distinguish differences in the form of instructions. When not distinguishing differences in the form of instructions, the manual omits "( )" or uses the notation for class names in accordance with the above-described instruction class notation. When such an instruction must be distinguished explicitly from an µOP instruction, it is expressed as an architecture instruction.

µOP instruction

Term that refers to the format of instructions decoded by the processor. Basically, the out-of-order execution engine of the processor handles all operations as µOP instructions.

Integer instruction

Term that refers to an instruction defined as an A64 Base instruction in the *ARMv8 Manual*. Since the instruction mainly handles integer values, this manual refers to it as an integer instruction.

SIMD&FP instruction

Term that refers to an instruction defined as an A64 Advanced SIMD and floating-point instruction in the *ARMv8 Manual*

SVE instruction

Term that refers to an instruction defined in the *SVE Manual*

Variant

Term that refers to an expression that allows multiple register sizes or per-element data sizes to be specified in the same instruction. Note that a variant has a different meaning for each instruction class.

For integer instructions, there are 32-bit and 64-bit variants, which are W and X, respectively, in the variant notation.

For SIMD&FP instructions, there are variants that represent register sizes themselves and variants that represent per-element data sizes. For example, a variant of FADD (scalar) represents a register size, and a variant of FADD (vector) represents a per-element data size.

An SVE instruction can specify a per-element data size for an operation independently from a memory access size. Accordingly, esize expresses the data size variant for an operation, and memsize expresses a memory access size.

The esize and memsize variants of integer and SIMD&FP instructions basically match each other. Nonetheless, when mentioning either of them, this manual respectively specifies esize or memsize.

Vector length (VL)

Term that refers to a vector length defined in the *SVE Manual*. Vector lengths are expressed by number of bits in this manual.

Vector data width

Generic term for an effective register size or memory access size of SIMD&FP and SVE instructions.

Number of vector elements

Term that refers to the number obtained by dividing the vector data width of a SIMD&FP or SVE instruction by esize or memsize. This is equivalent to the number of elements in vector data.

Operation instruction

Term that refers to an instruction whose input and output operands are closed in the same register file. The term applies to arithmetic, logic, and bitwise operation instructions. In a broad sense, the meaning includes transfer instructions, such as the MOV instruction. However, this manual does not treat transfer instructions between different register files as operation instructions.

Load/Store instruction

The load instruction is an instruction to transfer data from a memory space to a register. The store instruction is an instruction to transfer data from a register to a memory space.

Instruction dispatch

Term that refers to the mechanism that allocates an operation-flow from the decoder to a reservation station.

Instruction issue

Term that refers to the operation of submitting an operation-flow from a reservation station to an execution pipeline.

Execution complete

Term that refers to the condition when the execution of an architecture instruction, µOP instruction, or operation-flow has completed. The term also refers to the completion of a speculation condition.

Instruction commit

Term that refers to the updating of the processor architecture state after the execution of an architecture instruction has completed and a speculation condition has been determined. When expressing the term, this manual clearly distinguishes it from "execution complete."

Operation-flow, operation-request

Terms that refer to the distinct pipeline behavior of executing µOP instructions. The processor executes µOP instructions by combining operation-flows. An operation-flow is the minimum unit for hardware resource consumption. Additionally, the terms refer to not only objects (e.g.,µOP instructions) derived simply from architecture instructions but also a cache or memory access processing unit. Despite not clearly distinguishing between operation-flows and operation-requests, this manual may use the term "operation-request" to focus on the generation source and execution destination of a flow. Note that these terms may be abbreviated respectively to "flow" and "request."

1. Introduction

1.1. A64FX Processor Overview

The A64FX processor (called A64FX, below) is a superscalar processor of the out-of-order execution type. The A64FX is designed for high-performance computing (HPC) and complies with the ARMv8-A architecture profile and the Scalable Vector Extension for ARMv8-A. The processor integrates 52 processor cores including redundant cores; a memory controller supporting HBM2; a Tofu-D interconnect controller; and a root complex supporting PCI-Express Gen3.

The A64FX adopts several characteristic architectures for HPC.

Scalable Vector Extension

The A64FX supports the Scalable Vector Extension (SVE), which is the vector extension of the ARM instruction set architecture. Defining vector lengths of up to 2,048 bits as instruction sets, SVE features the ability to select and implement a vector length in hardware from multiples of 128 bits. The A64FX supports vector lengths of 128, 256, and 512 bits.

Core Memory Group

The A64FX has internal groups called core memory groups (CMGs), each of which consists of 13 processor cores, an independent L2 cache, and an independent memory controller. The processor has four CMGs and a non-uniform memory access (NUMA) configuration between the CMGs. Each physical memory space is split, and cache coherence is implicitly guaranteed by hardware.

Sector Cache

This function can virtually split a cache in units of ways and specify the area available at the instruction level. A program can specify an area by using a tagged address. The L1 cache has flat four sectors, and the L2 cache has two groups of two sectors.

Hardware Barrier

This function uses hardware to support synchronization between software processes or threads. The function enables synchronization processing without memory access.

Hardware Prefetch Assist

This function can control the behavior of hardware prefetch from a program. The program can provide information to the prefetch mechanism of hardware by using the system register and tagged addresses.

High Bandwidth Memory

High Bandwidth Memory Gen2 (HBM2) is the main memory adopted to provide a very high memory bandwidth.

1.2. A64FX Processor Specification

Table 1-1 lists the main specifications of the A64FX processor.

Table 1-1 A64FX Processor Specifications

	Specification
Number of processor cores	52 (13 cores / CMG)
Number of CMGs	4
L1I cache size	64 KiB / 4-way
L1D cache size	64 KiB / 4-way
L2 cache size	32 MiB / 16-way (8 MiB / CMG)
Cache-line size	256 bytes
Memory controller	4 (1 MAC / CMG)
Interconnect	Tofu-D
I/O	PCI-Express Gen3 16 Lanes
Instruction set architecture	ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 (*1), SVE
SVE-implemented Vector Length	128 / 256 / 512 bits

(*1) ARMv8.3 supports only complex-number supported instructions.

1.3. A64FX Processor Block Diagram

Figure 1-1 shows the main functional blocks on the A64FX processor chip. Table 1-2 shows the correspondence between the block markings and functional units.

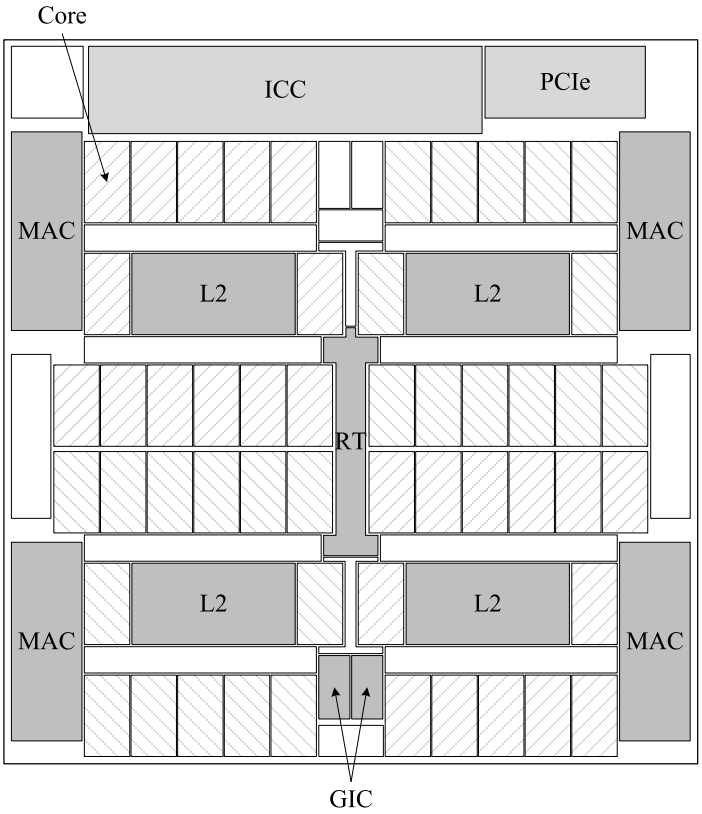


Figure 1-1 Main Functional Blocks on A64FX Processor Chip

Table 1-2 Correspondence Between Processor Chip Block Markings and Functional Units

Block Marking in Figure	Functional Unit
Core	Processor core
L2	L2 cache
ICC	Tofu-D interconnect controller
PCIe	PCI-Express Gen3 root complex
RT	Routing controller between CMGs
MAC	Memory controller
GIC	Interrupt controller

2. Out-of-Order Architecture

This chapter describes the basic architecture of the A64FX processor core.

2.1. Overview

Figure 2-1 shows an outline of the basic A64FX pipelines and stages. The A64FX can be roughly divided into five functional stages.

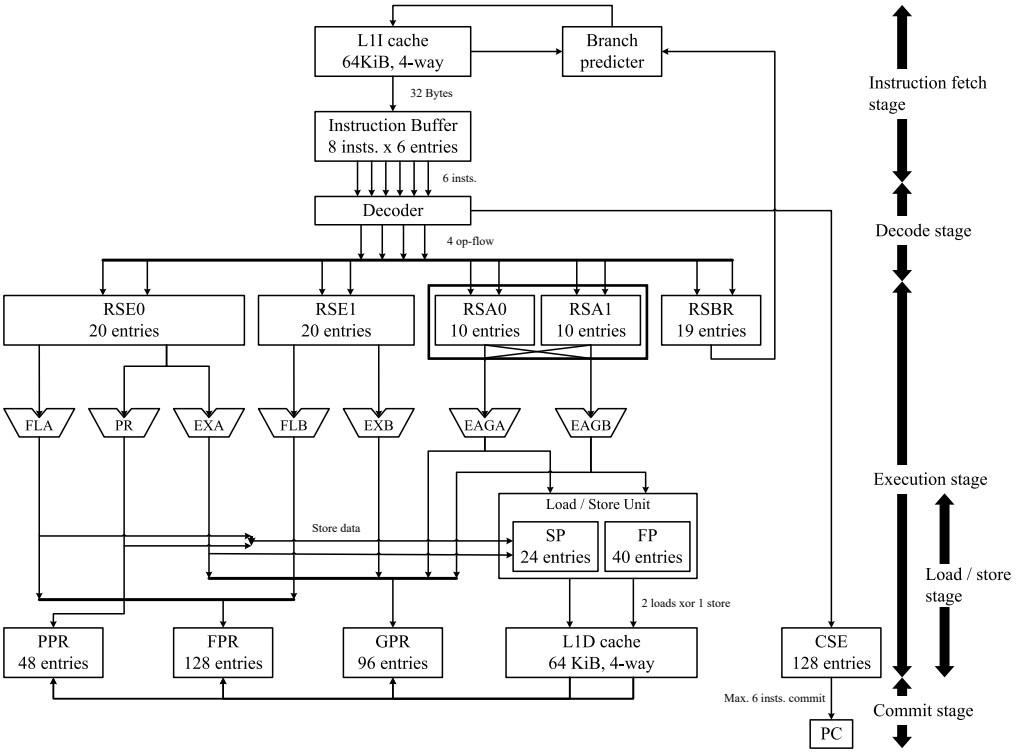


Figure 2-1 Overall Illustration of Stages

Instruction Fetch Stage

This stage consists of the L1I cache, L1-ITLB/L2-ITLB, a branch prediction mechanism, and an instruction fetch control module. The instruction fetch unit can fetch up to eight instructions simultaneously from the L1I cache. The branch prediction mechanism predicts the branch directions of up to four branch instructions per cycle and predicts the branch target of up to one Taken branch instruction. The fetched instructions are temporarily stored in the instruction buffer (IBUFF).

Decode Stage

Decode at this stage usually obtains up to four instructions or, when a MOVPRFX instruction is included, up to six instructions per cycle from IBUFF. The obtained architecture instructions are decoded into µOP instructions, which have the internal instruction format. Basically, one architecture instruction is split into one µOP instruction, but any architecture instruction requiring complex operations is split into multiple µOP instructions. On the other hand, a MOVPRFX instruction is packed together with a modified instruction, and they are decoded like a single architecture instruction. After the decode, the µOP instructions are dispatched to a reservation station as an in-order operation-flow.

Execution Stage

Five reservation stations (RSs) are implemented, each which is connected for an execution pipeline group. The RSs schedule operation-flows dispatched to them and issue the flows out of order. The

execution pipelines consist of the integer operation pipeline (EXA/EXB), floating-point operation pipeline (FLA/FLB), predicate operation pipeline (PR), address calculation pipeline (EAGA/EAGB), and branch execution pipeline. EXA/EXB mainly performs integer operations. FLA/FLB performs SIMD&FP and SVE instruction operations. PR performs predicate instruction operations. EAGA/EAGB generates addresses for load/store instructions and performs some integer operations. The branch execution pipeline executes branch instructions.

Load/Store Stage

This stage consists of one L1D cache, one L1-DTLB, one L2-DTLB, and two load/store pipelines. The address calculation pipelines are connected directly to the load/store pipelines. The load/store pipelines can execute two load operation-flows or one store operation-flow simultaneously.

Commit Stage

This stage judges the completion of instructions, including checking for exceptions in execution results and checking branch prediction results. µOP instructions whose execution has completed are committed in order. After all the µOP instructions paired with architecture instructions are committed, the architecture state of the processor is updated. Up to four µOP instructions can be committed per cycle.

2.2. Micro-Operation Instruction

The A64FX decodes architecture instructions into micro-operation instructions (µOP instructions), which are in the internal instruction format. A µOP instruction is an instruction unit suitable for hardware instruction execution. A complex architecture instruction is split into multiple µOP instructions. In contrast, some architecture instructions are combined into a single µOP instruction to optimize them for hardware operations. "List of Instruction Attribute and Latency" shows the number of architecture instruction splits.

The two types of decode where one architecture instruction is split into two or more µOP instructions are normal decode and sequential decode. For the latter, dispatch, commit, and resource allocation are limited. Whether sequential decode is performed depends on the decode source architecture instruction.

2.3. Operation-Flow

An operation-flow is the distinct circuit operation of executing an µOP instruction and is also the minimum unit of pipeline processing by the execution engine. µOP instructions are executed by combining operation-flows. This means that all processing units at the execution and load/store stages are operation-flows. µOP instructions are converted into operation-flows when they are dispatched from decoders to reservation stations. Operation-flows include not only derivatives of µOP instructions but also instructions spontaneously generated by hardware, such as, for example, ones for hardware prefetch and cache miss processing.

2.4. Out-of-Order Resources

Table 2-1 lists the main resources for out-of-order execution and the quantity of the respective resources.

Table 2-1 Out-of-Order Resources

Resource	Quantity of Resource		
Commit Stack Entry (CSE)	128 entries		
Group ID (GID)	32 entries		
General-purpose physical register (GPR)	96 entries	Architecture register	32 entries
		Renaming register	64 entries
Floating-point physical register (FPR)	128 entries	Architecture register	32 entries
		Renaming register	96 entries
Predicate physical register (PPR)	48 entries	Architecture register	16 entries
		Renaming register	32 entries
Reservation Station for EAG (RSA)	10 entries x 2 (split)		
Reservation Station for EXE (RSE) (shared by Integer, SIMD&FP, SVE)	20 entries x 2 (split)		
Reservation Station for Branch (RSBR)	19 entries		
Temporary Operand Register (TOR)	3 entries		
Fetch Port (FP)	Virtual	160 entries	
	Real	40 entries	
Store Port (SP)	Virtual	192 entries	
	Real	24 entries	
Write Buffer (WB)	8 entries		

The main functions of the respective resources are as follows.

Commit stack entry (CSE)

This resource is used to reorder instructions executed out of order into program order. Architecture instructions are decoded into µOP instructions and allocated to CSEs.

Group ID (GID)

This ID is used to manage a µOP instruction dispatch group. Up to four µOP instructions are allocated per GID.

General-purpose physical register (GPR)

This register is the physical entity of the architecture and renaming registers allocated to a general-purpose register described in the *ARM Manual*.

Floating-point physical register (FPR)

This register is the physical entity of the architecture and renaming registers allocated to a SIMD&FP register described in the *ARM Manual* and a vector register described in the *SVE Manual*.

Predicate physical register (PPR)

This register is the physical entity of the architecture and renaming registers allocated to a predicate register described in the *ARM Manual*.

Reservation station for EAG (RSA)

This RS is a scheduler for temporarily storing operation-flows to be executed in the EAGA/EAGB pipeline and issuing them out of order.

Reservation station for EXE (RSE)

This RS is a scheduler for temporarily storing operation-flows to be executed in the EXA/EXB/FLA/FLB/PR pipeline and issuing them out of order.

Reservation station for branch (RSBR)

This RS is a scheduler for temporarily storing branch instruction operation-flows and executing them out of order.

Temporary operand register (TOR)

This register is used to transfer a program counter (PC) value from the instruction fetch stage to the execution stage. Basically, the register is used only for PC-relative instructions and branch and link instructions.

Fetch port (FP)

This resource is used to manage the execution order of load/store instructions. The A64FX adopts a new function called "virtual fetch port" (VFP). In contrast to the VFP, a fetch port that has the original function is called "real fetch port" (RFP).

Store port (SP)

This resource is used to manage the execution order of store instructions. Like with the fetch port, the A64FX has the virtual store port (VSP) and the real store port (RSP).

Write buffer (WB)

This resource is used to temporarily store post-commit store data until it is written to the L1D cache.

2.5. Pipeline Stage

This section describes pipeline stages for out-of-order execution. The pipeline stages vary depending on the execution pipeline and the types of instructions, operations, and load/store executed in the pipeline.

Figure 2-2 to Figure 2-6 show operations at the main pipeline stages. Table 2-2 shows the correspondence between stage symbols in the figures and operations.

Table 2-2 Correspondence Between Pipeline Stage Symbols and Operations

Stage Symbol	Operation
Common to all pipelines	
D, DT	Instruction decode
P, PT	Instruction scheduling
B*	Physical register read
C	Commit
W, W2	Architecture register update
Specific to operation pipelines	
Xn	Operation execution (The number of stages varies depending on the instruction.)
U, UT*	Operation result update
EXP	Exception judgment
Specific to branch execution pipeline	
BS	Scheduling
BR	Branch direction judgment
BC	Branch direction determination
Specific to load/store pipelines	
A	Effective address generation
T	Tag and TLB access
M, B, XT, XM, XB	Data access
R, RT*	Result out
W3 – W5	WB write

	1	2	3	4	5	6	7	8	9	10	11
Integer	D	DT	P	PT	B1	B2	X	U C	UT W	W2	

Figure 2-2 Integer Operation Pipeline Stages

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
SIMD&FP/ SVE	D	DT	P	PT	PT2	PT3	B1	B2	X1	X2	X3	X4	X5	X6	X7	X8	X9	U	UT	UT2	EXP	C	W	W2

Figure 2-3 SIMD&FP and SVE Operation Pipeline Stages

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Predicate	D	DT	P	PT	PT2	B1	B2	X1	X2	X3	U C	UT W	W2	
Predicate (update NZCV)	D	DT	P	PT	PT2	B1	B2	X1	X2	X3	U	UT C	W	W2

Figure 2-4 Predicate Operation Pipeline Stages

	1	2	3	4	5	6	7	8	9	10	11	12	13
Unconditional branch	D	DT	BS	BC	C	W							
Conditional branch	D	DT	BS	BR	BC	C	W						
Unconditional indirect branch	D	DT	P BS	PT	B1	B2	X	U	UT	BR	BC	C	W
Compare & branch	D	DT	P BS	PT	B1	B2	X	U	UT	BR	BC	C	W

Figure 2-5 Branch Pipeline Stages

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Integer load	D	DT	P	PT	B1	B2	XA	T	M	B	R	RT	RT2	RT3C	W	W2						
Integer store	D	DT	PP	PTPT	B1B1	B2B2	XA	T	M	B	R	RT	RT2	RT3C	W	W2	W3	W4	W5			
SIMD&FP/SVE load (short)	D	DT	P	PT	B1	B2	XA	T	M	B	R	RT	RT2	RT3C	W	W2						
SIMD&FP/SVE load (long)	D	DT	P	PT	B1	B2	XA	T	M	B	XT	XM	XB	R	RT	RT2	RT3C	W	W2			
SIMD&FP/SVE store (short)	D	DT	PP	PTPT	B1B1	B2B2	XA	T	M	B	R	RT	RT2	RT3C	W	W2	W3	W4	W5			
SIMD&FP/SVE store (long)	D	DT	PP	PTPT	B1B1	B2B2	XA	T	M	B	XT	XM	XB	R	RT	RT2	RT3C	W	W2	W3	W4	W5
Predicate load	D	DT	P	PT	B1	B2	XA	T	M	B	XT	XM	XB	R	RT	RT2	RT3C	W	W2			
Predicate store	D	DT	PP	PTPT	B1B1	B2B2	XA	T	M	B	XT	XM	XB	R	RT	RT2	RT3C	W	W2	W3	W4	W5

Figure 2-6 Load/Store Pipeline Stages

2.6. Execution Latency

The basic latency in the execution of an instruction is determined by the number of stages from either: start to completion of operations at the operation stages among the above pipeline stages; or start to completion of memory access at the load/store stages. Table 2-3 summarizes the execution start and completion stages for each instruction in each pipeline. In addition, "List of Instruction Attribute and Latency" shows the latency of each instruction.

Table 2-3 Execution Start and Completion Stages for Each Instruction in Each Pipeline

Pipeline	Instruction	Start Stage	Completion Stage
EXA / EXB		X	Xn
EAGA / EAGB	(Operation instruction only)		
FLA / FLB		X1	Xn
PR		X1	X3
Load / Store	Integer load	A	R
	SIMD&FP / SVE load	A	RT3
	Predicate load	A	RT

A store instruction has an execution latency different from typical execution latencies since the execution of the instruction is completed after commit. For this reason, it is omitted here.

The number of stages of each of the EXA/EXB, EAGA/EAGB, and FLA/FLB pipelines varies depending on the contents of the instruction operations. For this reason, the completion stage is expressed as Xn.

2.7. Operand Bypass

Operand bypass refers to passing a value generated by a producer to a consumer without going through a register when an operand of the consumer depends on the execution results of the producer. Basically, a bypass route is implemented to connect the start stage of the next instruction immediately after the completion stage shown in Table 2-3 above. However, depending on the combination of execution pipelines or instructions, bypassing cannot be done without a penalty. Table 2-4 shows penalty cycles determined by the combination of operands that depend on the type of pipeline and type of instruction. One column in the table lists instructions that generate operands, and one row lists instructions that use operands as input.

Table 2-4 Penalties for Operand Bypass Between µOP Instructions

	Consumer	EXA	EXB	EAGA / PIPE0							EAGB / PIPE1							FLA	FLB	PR
Producer		Integer operation	Integer operation	Integer operation	Integer load	Integer store	SIMD&FP/SVE load	SIMD&FP/SVE store	Predicate load	Predicate store	Integer operation	Integer load	Integer store	SIMD&FP/SVE load	SIMD&FP/SVE store	Predicate load	Predicate store	SIMD&FP/SVE operation	SIMD&FP/SVE operation	Predicate operation
EXA	Integer operation	0	1	1	1	1	1	1	1	-	1	1	1	1	1	1	-	-	-	-
	Integer operation (update NZCV)	0	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	7	7	6
EXB	Integer operation	1	0	1	1	1	1	1	1	-	1	1	1	1	1	1	-	-	-	-
	Integer operation (update NZCV)	1	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	7	7	6
EAGA / PIPE0	Integer operation	1	1	0	0	0	0	0	0	-	1	1	1	1	1	1	-	-	-	-
	Integer load	0	0	0	0	0	0	0	0	-	0	0	0	0	0	0	-	-	-	-
	Integer store	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	SIMD&FP/SVE load (short)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP/SVE load (long)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP/SVE store (short)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	SIMD&FP/SVE store (long)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Predicate load	-	-	-	-	-	0	0	-	-	-	-	-	0	0	-	-	3	3	1
	Predicate store	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
EAGB / PIPE1	Integer operation	1	1	1	1	1	1	1	1	-	0	0	0	0	0	0	-	-	-	-
	Integer load	0	0	0	0	0	0	0	0	-	0	0	0	0	0	0	-	-	-	-
	Integer store	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	SIMD&FP/SVE load (short)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP/SVE load (long)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP/SVE store (short)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	SIMD&FP/SVE store (long)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	Predicate load	-	-	-	-	-	0	0	-	-	-	-	-	0	0	-	-	3	3	1
	Predicate store	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PR	Predicate operation	-	-	-	-	-	1	1	-	-	-	-	-	1	0	-	-	3	3	0

	Consumer	EXA	EXB	EAGA / PIPE0							EAGB / PIPE1							FLA	FLB	PR
	Predicate operation (update NZCV)	6	6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	8	8	7
FLA	SIMD&FP/SVE operation	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP/SVE operation (update NZCV)	5	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	6
	SVE CMP instruction (update PR)	-	-	-	-	-	2	2	-	-	-	-	-	2	2	-	-	1	1	2
	SVE CMP instruction (update NZCV)	9	9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	11	11	10
FLB	SIMD&FP/SVE operation	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	-
	SIMD&FP operation (update NZCV)	5	5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0	6

Basically, penalty cycles are determined by the types of instructions and the combinations of pipelines on the generation and input sides. Some instructions can have dependency on multiple operands. For example, some integer operation instructions simultaneously output data to a general-purpose register and the NZCV register. In their case, the penalty cycle when the consumer depends on an operand in the general-purpose register differs from that when the consumer depends on an operand in the NZCV register. SVE instructions basically use floating-point and predicate registers as input operands. In their case, the penalty cycle varies depending on which operand depends on the producer. The generation instructions in Table 2-4 are also shown in combinations of types of operands. The corresponding operands are used as input by input instructions.

Note that the above rules do not apply to the FTMAD instruction, for which the penalty cycle varies depending on the instruction that generates the operand. Table 2-5 shows the penalty cycles for the FTMAD instruction.

Table 2-5 Penalties for Operand Bypass Between µOP Instructions (FTMAD Instruction)

	FTMAD
SVE load (Long)	0
FTSMUL	0
Other floating-point operation instruction	1

2.8. Resource Allocation and Release

Out-of-order resources are allocated every time an instruction is executed, and released when the execution ends. Table 2-6 shows the allocation and release stages of individual resources.

Table 2-6 Out-of-Order Resource Allocation and Release Stages

Resource		Allocation Stage	Release Stage	Supplemental Remarks
CSE		D	W	The resource is allocated and released in order.
General-purpose renaming register		D	W2	
Floating-point renaming register		D	W2	
Predicate renaming register		D	W2	
RSE	EXA / EXB pipeline	D	B1, B2, X, X+1	The release stage depends on the operand bypass timing. The resource is released out of order
	FLA / FLB pipeline	D	PT2, PT3, PT3+1	The release stage depends on the operand bypass timing. The resource is released out of order.
RSA	Load/Store instruction	D	B2, A, A+1	The release stage depends on the operand bypass timing. The resource is released out of order.
	Integer operation instruction	D	B1, B2, X, X+1	The release stage depends on the operand bypass timing. The resource is released out of order.
Virtual FP		D	Same as for Real FP	The resource is allocated in order.
Virtual SP		D	Same as for Real SP	
Real FP	Integer load / store, SIMD&FP load / store, SVE load / store (excluding predicate)	B1	RT3	The resource is released in order and without waiting for commit.
	Predicate load / store		RT3	
Real SP	Integer store	B1	W5	The resource is released in order after commit.
	SIMD&FP / SVE store	PT2		

2.9. Execution Latency Changing

Depending on the type of instruction, latency varies even between operations that pass through the same pipeline. At such time, operations may conflict with each other at later stages (C, W) even when multiple instructions are submitted to a pipeline at different times as shown in Figure 2-7. Since such a pipeline condition is not allowed, the A64FX prevents conflicts by changing latency during execution stages as shown in Figure 2-8.

	1	2	3	4	5	6	7	8	9	10	11
Preceding instruction	X1	X2	X3	X4	X5	X6	X7	X8	X9	C	W
Following instruction						X1	X2	X3	X4	C	W

Figure 2-7 Example of Conflict Between C Stages of Instructions with Different Latencies

	1	2	3	4	5	6	7	8	9	10	11	12	13
Preceding instruction	X1	X2	X3	X4	X5	X6	X7	X8	X9	C	W		
Following instruction						X1	X2	X3	X4	~~C~~ X5	~~W~~ X6	C	W

Figure 2-8 Example of Latency Changing

Table 2-7 lists the type of instruction whose latency is changed and lists its latency after change.

Table 2-7 Instructions Whose Latency Changed, and Their Latencies

Instruction Type	Basic Latency	Latency After Change
Instruction executed in floating-point operation pipeline	4	6 or 9
	6	9
	9	No change
Load instruction	5	8
	8	11
	9	No change
	11	No change

3. Instruction Fetch

3.1. Overview of Fetch Stage

The instruction fetch stage fetches instructions from the L1I cache and provides them to the decode stage. The instruction fetch stage contains the L1I cache, L1-ITLB, and branch prediction mechanism. Figure 3-1 shows an outline of the instruction fetch stage. IFEAG is an adder that updates a program counter (PC). The PC is sent to the branch prediction mechanism, L1-ITLB, and the L1I cache. Based on either the PC from the result of branch prediction or the PC from IFEAG, L1-ITLB and the L1I cache are accessed to read instructions. Instructions are read in units of aligned 32 bytes and stored in the instruction buffer (IBUFF) with images of the read instructions maintained without change. IBUFF consists of 6 entries and can store 32 bytes (8 instructions) per entry.

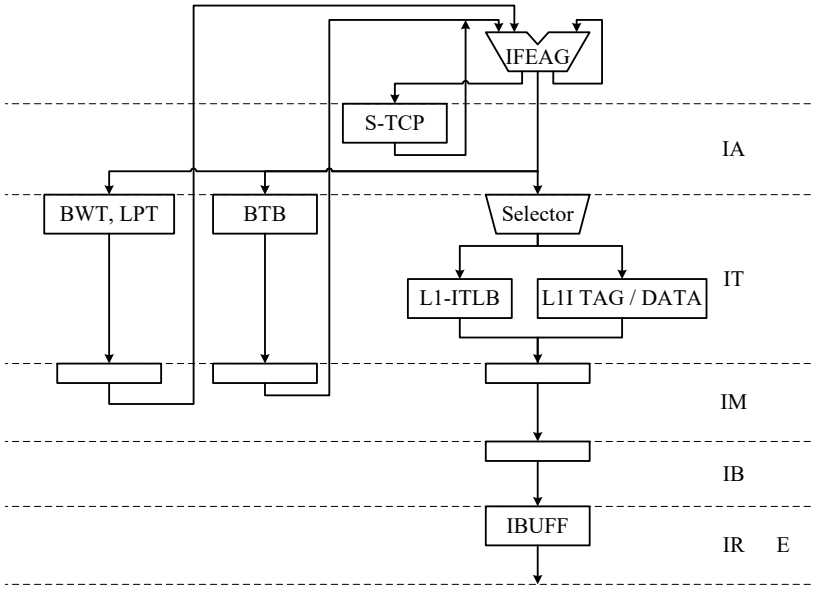


Figure 3-1 Instruction Fetch Stage

The branch prediction mechanism predicts the branch directions and branch target addresses of branch instructions. If a fetched instruction sequence contains a branch instruction for which "Taken" is predicted, the fetch destination of the next instruction is a predicted branch target address. The basic access latency of the branch prediction mechanism is 3 cycles, in which case bubbles will occur because the fetch of instructions following the Taken branch instruction is canceled as shown in Figure 3-2.

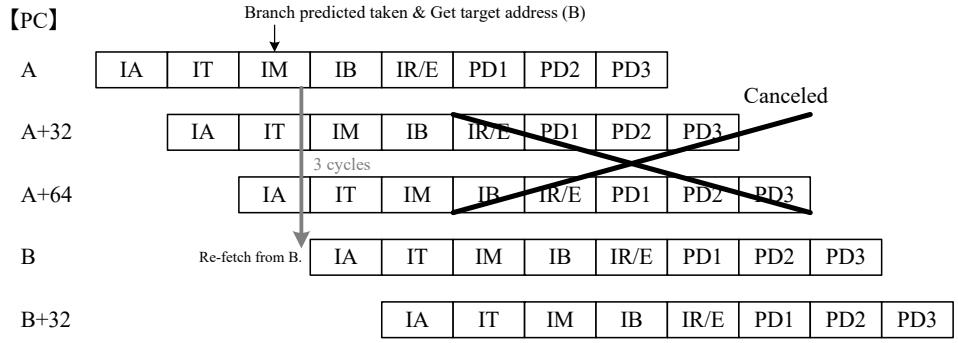


Figure 3-2 Bubbles Due to Instructions Following Taken Branch Instruction

3.2. Branch Prediction Mechanism

The branch prediction mechanism of the A64FX consists of the branch predictors shown in Table 3-1.

Table 3-1 Branch Predictors of Branch Prediction Mechanism

Role	Branch Predictor
Branch direction & Branch target address prediction	Small Taken Chain Predictor (S-TCP)
Branch direction prediction	Branch Weight Table (BWT) Loop Prediction Table (LPT) Return Address Stack (RAS)
Branch target address prediction	Branch Target Buffer (BTB)

Among them, BWT and BTB are the major predictors of the branch prediction mechanism, which predicts branch directions and branch target addresses by combining the two predictors. BWT makes a prediction in combination with the global history register (GHR) by using a piecewise linear algorithm. S-TCP, which is a low-capacity, low-latency buffer for storing the branch target addresses of Taken branch instructions, makes a prediction when a loop structure is identified. LPT, which is a counter-type local branch predictor, predicts the number of times that branch directions will be the same, like a conditional branch instruction that makes a judgment to continue in a loop structure. RAS is a predictor for return addresses from subroutines. The following sections describe these branch predictors in detail.

3.2.1. Small Taken Chain Predictor

The small taken chain predictor (S-TCP) is a mechanism that detects a chain of program execution paths of Taken branch instructions and makes a prediction. As shown in Figure 3-3, S-TCP detects a chain structure consisting of the execution paths of multiple Taken branch instructions. This chain forms a loop. When the number of execution iterations exceeds a certain number of times, an instruction is given for an instruction fetch according to the detected execution paths. S-TCP can store the branch target addresses of four Taken branch instructions. Since S-TCP does not store information on Not-Taken branch instructions, the number of Not-Taken branch instructions included in a chain will be unlimited.

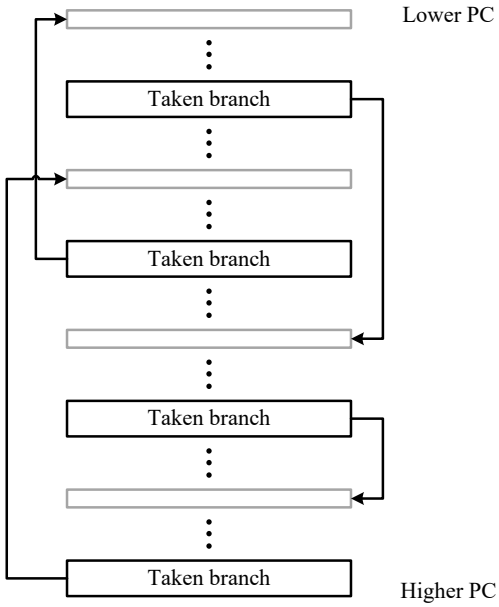


Figure 3-3 Chain Structure Consisting of Multiple Taken Branch Instructions

S-TCP does not immediately delete detected information even when a misprediction occurs. In other words, S-TCP resumes prediction when an instruction in the same execution path is fetched.

In addition, since S-TCP can be accessed in one cycle, no pipeline bubbles occur at the instruction fetch.

3.2.2. Loop Prediction Table

The loop prediction table (LPT) is a counter-type local historical branch direction predictor. LPT records the number of consecutive times of Taken or Not-Taken conditional branch instructions, and predicts branch directions based on the history. It consists of eight entries and can record up to eight branch instructions.

3.2.3. Branch Weight Table

The branch weight table (BWT) uses a piecewise linear algorithm to make predictions in combination with the global history register (GHR). BWT is a weight table in a piecewise-linear format and has a 2,048-entry configuration. As shown in Figure 3-4, the history of conditional branches and the history of weights calculated in the past at the branch instruction execution time are used for prediction. Both two histories are global histories. The history of conditional branches has a value of 1 at the time of Taken and -1 at the time of Not Taken. A weight (W) is a signed integer. With the histories of conditional branches and weights shown in Figure 3-4, conditional branch instruction B_0 is predicted as result P in Figure 3-5 (Eq.1). If P is 0 or greater, "Taken" is predicted. If it is less than 0, "Not Taken" is predicted. Subsequently, after the branch instruction is executed and the result of the branch is established, the history of weights is updated. The history of weights is updated only when a misprediction occurs. For example, if branch prediction B_0 is predicted as "Taken" and the execution result is "Not Taken," the weight update equation shown in Figure 3-5 (Eq.2) is used.

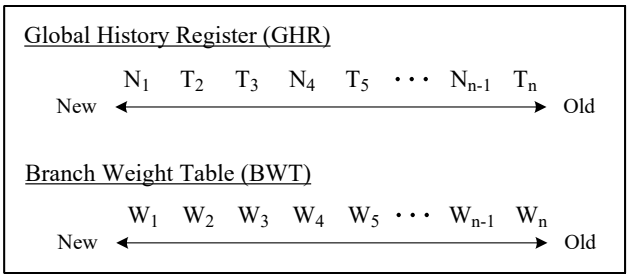


Figure 3-4 Histories of Conditional Branches and Weights

(Eq.1) Predicted threshold

$$P = W_0 + N_1W_1 + T_2W_2 + T_3W_3 + N_4W_4 + T_5W_5 + \cdots + N_{n-1}W_{n-1} + T_nW_n$$

(Eq.2) Update weights of the conditional branch instruction B_0

$$[W_0\ W_1\ W_2\ W_3\ W_4\ W_5\ \cdots\ W_{n-1}\ W_n] = [W_0\ W_1\ W_2\ W_3\ W_4\ W_5\ \cdots\ W_{n-1}\ W_n] + [T_c\ N_1\ T_2\ T_3\ N_4\ T_5\ \cdots\ N_{n-1}\ T_n] * N_0$$

* Tc is constant.

Figure 3-5 Prediction Equation for Conditional Branch Instruction B_0

To make the explanation easy to understand, this manual uses "Taken" and "Not Taken" as prediction results, whereas the implemented prediction mechanism adopts the Agree Prediction scheme.

3.2.4. Branch Target Buffer

The branch target buffer (BTB) is a buffer that records the branch target addresses of relative and indirect branch instructions. BTB has a 4-way, 2,048-entry configuration. It stores only the branch history from one execution in the past since the branch target addresses of relative branch instructions are statically determined. The branch target addresses of indirect branch instructions may change dynamically. Therefore, a mechanism called Rehash is adopted, which makes a prediction by storing multiple branch histories. As shown in Figure 3-6, hashing is performed on the BTB indexes by using branch direction and branch target histories. This enables prediction of branch target addresses that dynamically change.

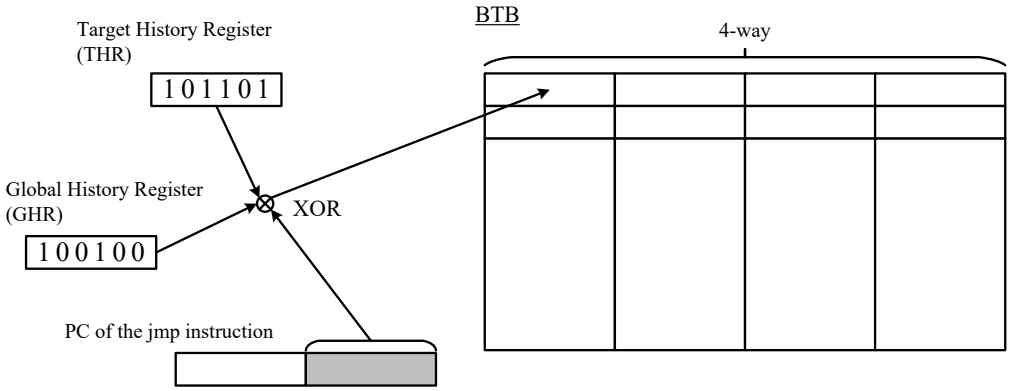


Figure 3-6 Outline of Branch Target Buffer (BTB)

3.2.5. Return Address Stack

The return address stack (RAS) is a stack that stores return addresses at the subroutine call. RAS records return addresses when a BL or BLR instruction is executed, and they are referenced when a RET instruction is fetched. It has an 8-entry configuration. RAS has 2 access cycles, which is shorter than for BTB.

3.3. Combination of Predictors

As described above, the A64FX makes a branch instruction prediction by combining multiple predictors. Table 3-2 shows the types of branch instructions, predictors used, and prediction result adoption rankings.

Table 3-2 Relationship Between Predictors Used for Branch Prediction and Prediction Result Adoption Rankings

Adoption Ranking	Conditional Branch Instruction	Indirect Branch Instruction, Unconditional Relative Branch Instruction
High	S-TCP	S-TCP
	LPT, BTB	RAS
Low	BWT, BTB	BTB

Due to its operational characteristics, S-TCP predicts branch targets even for indirect or conditional branch instructions. In addition, since the access latency is the shortest, its prediction result is adopted with the highest priority. LPT and BWT only predict branch directions. and branch targets predicated by BTB are used when "Taken" is predicted by them. The predicted results of LPT take priority over those of BWT.

For indirect branches, the predicted results of RAS take priority over those of BTB. However, RAS makes a prediction only for return instructions in subroutines, whereas BTB learns all branch instructions.

3.4. Short Loop Detector

The A64FX has a mechanism called a short loop detector. This mechanism detects a loop structure in the instruction sequences stored in IBUFF. When a short loop is detected, reading from the instruction cache stops, and IBUFF can provide instructions. When instructions are provided from IBUFF, the penalties for reading instructions following a Taken branch do not apply, and no pipeline bubbles occur.

The conditions for short loop detection are as follows.

- IBUFF should be able to store the entire instruction sequence that composes a loop. As described above, IBUFF can store 48 instructions. However, alignment restrictions on instruction sequences must be taken into consideration.
- All branch directions must be constant in a loop. The loop can contain multiple branch instructions.

IBUFF starts providing instructions when the above conditions are met and the number of loop iterations exceeds the threshold value. If the direction of branch instructions in a short loop changes, IBUFF stops providing instructions, and reading from the instruction cache resumes.

4. Instruction Decode and Commit

4.1. Micro-Operation Instruction

The A64FX decodes architecture instructions into µOP instructions, which are in the internal format specific to the hardware. A µOP instruction is the basic unit to allocate it for resources, such as renaming registers, CSEs, FPs, and SPs. Most architecture instruction may be split into one or more µOP instructions. Certain pair of architecture instructions may be combined to generate one µOP instruction. "List of Instruction Attribute and Latency" shows the number of splits for µOP instructions. Section 4.3 describes combination of architecture instructions.

There are two types of decode for splitting one architecture instruction into two or more µOP instructions: normal decode and sequential decode. Sequential decode has limitations in instruction dispatch, commit, and GID allocation. Normal decode can simultaneously decode multiple architecture instructions and dispatch multiple µOP instructions. In contrast, sequential decode decodes only one architecture instruction and sequentially dispatches µOP instructions. And one GID is allocated to each µOP instruction. "List of Instruction Attribute and Latency" shows instructions to which sequential decode applies.

4.2. Multi-Operation

µOP instructions are executed in operation-flows. The required number of operation-flows depends on the complexity of the µOP instructions. For example, operations like simple integer calculations can be executed in one flow, but operations that combine an arithmetic operation with a shift, like ADD (shifted register) instruction, are split at the operation time and executed in multiple flows. The number of flows is 0 (no operation exists) for NOP and other instructions that do not require operations. On the other hand, SVE gather load instructions are complex, and they are executed across a few address generation flows and many memory access flows. This is because the number of flows required for executing the essential functions varies at the respective pipeline stages. That is also because the A64FX has an architecture that splits operation-flows at a stage as downstream as possible for the purpose of suppressing resource consumption. Operation-flow splitting is performed at the following pipeline stages.

Decode Stage

Splitting is performed when a µOP instruction is dispatched to the reservation stage. The instructions mainly subject to this splitting are those requiring processing in execution pipelines with different functions for instruction execution. Representative examples include store instructions. A store instruction is split into two flows: one calculates effective addresses for dispatch to the RSA, and the other is a data transfer flow for dispatch to the RSE.

Execution Stage

Splitting is performed when an instruction is issued from a reservation station to an execution pipeline. The instructions subject to this splitting are those that repeatedly perform multiple different operations. Representative examples include ADD (shifted register) instruction. When dispatched, the µOP instructions of an ADD instruction are converted into one flow only. However, when issued from a reservation station to an execution pipeline, they are split into two flows: one performs shift operation, and the other performs add operation.

Load/Store Stage

Splitting is performed when load/store is executed at the load/store stage. Like gather/scatter instructions and multiple-structures instructions, instructions that access discrete memory space are subject to this splitting.

4.3. MOVPRFX Instruction Packing

The A64FX basically combines a MOVPRFX instruction with a modified instruction following it, and decodes them so that they behave as though the modified instruction was a non-destructive instruction. This

join processing is called packing. Packing is performed at the first stage of pre-decode, and then µOP instruction splitting is performed. It means that the number of µOP instructions at the decode time is not affected by whether a MOVPRFX instruction exists or not. The number of µOP instruction splits is determined only by the attributes of the modified instruction.

On the other hand, MOVPRFX instruction packing has the following restrictions on the number of parallel processes.

- Up to six instructions per cycle can be input to the first stage of the pre-decoder.
- Up to three pairs per cycle can be packed at the first stage of the pre-decoder.
- Up to four instructions after packing can be output from the first stage of the pre-decoder.

These restrictions may cause the throughput in pre-decode to decrease depending on the sequence of architecture instructions in a program. Figure 4-1 and Figure 4-2 show examples.

```
loop:
   movprfx   z0.d, p0/m, z1.d          ┐
   fmad      z1.d, p0/m, z2.d, z3.d    │
   add       z1.d, p0/m, z4.d          │ 4 insts.
   movprfx   z10.d, p0/m, z11.d        │ decoded
   fmad      z11.d, p0/m, z12.d, z13.d │
   add       z11.d, p0/m, z14.d        ┘
   movprfx   z5.d, p5/m, z6.d          ┐
   fmad      z6.d, p5/m, z7.d, z8.d    │
   sub       z6.d, p5/m, z9.d          │ 4 insts.
   movprfx   z15.d, p5/m, z16.d        │ decoded
   fmad      z16.d, p5/m, z17.d, z18.d │
   sub       z16.d, p5/m, z19.d        ┘
   b.ne      loop
```

Figure 4-1 Example of Efficient Packing with MOVPRFX

```
loop:
   movprfx   z0.d, p0/m, z1.d          ┐
   fmad      z1.d, p0/m, z2.d, z3.d    │
   movprfx   z10.d, p0/m, z11.d        │ 3 insts.
   fmad      z11.d, p0/m, z12.d, z13.d │ decoded only
   movprfx   z5.d, p5/m, z6.d          │
   fmad      z6.d, p5/m, z7.d, z8.d    ┘
   movprfx   z15.d, p5/m, z16.d        ┐
   fmad      z16.d, p5/m, z17.d, z18.d │
   add       z1.d, p0/m, z4.d          │ 4 insts.
   add       z11.d, p0/m, z14.d        │ decoded
   sub       z6.d, p5/m, z9.d          ┘
   sub       z16.d, p5/m, z19.d        ┐ 2 insts.
   b.ne      loop                      ┘ decoded only
```

Figure 4-2 Example of Inefficient Packing Due to Instruction Order

As shown in Figure 4-1 and Figure 4-2, throughput varies depending on the sequence of MOVPRFX and modified instructions, even when the same number of instructions are decoded. We recommend taking the above restrictions into consideration when instruction scheduling is flexible.

4.4. Instruction Decode

Figure 4-3 shows an outline of the pipeline stages of the A64FX instruction decode stage. The decoder obtains instructions from IBUFF, decodes them into µOP instructions, and allocates them to out-of-order resources.

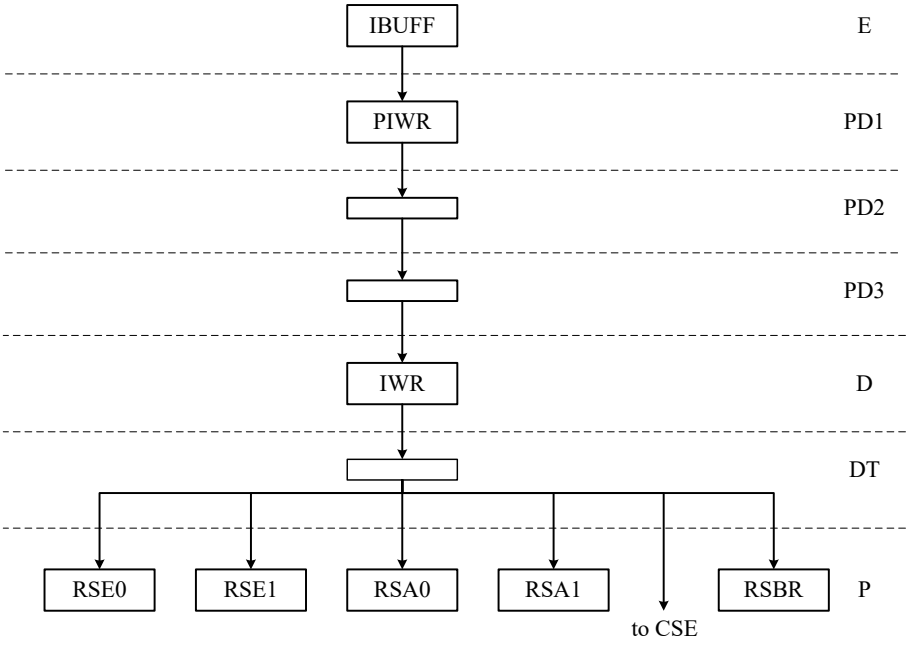


Figure 4-3 Instruction Decode Stage

4.4.1. Pre-Decode

Pre-decode is performed at the PD1 to PD3 stages. Mainly, MOVPRFX instruction packing and splitting into µOP instructions are performed in pre-decode. The PD1 has the pre-decode instruction windows register (PIWR) with a 6-instruction width in terms of architecture instructions and accepts input from IBUFF. The PD2 stage performs µOP instruction splitting. It has an instruction register with a 7-instruction width in terms of µOP instructions to absorb an increase in the number of instructions due to the split. The PD3 stage outputs instructions to the decoder in the subsequent stage and has an instruction register with a 4-instruction width in terms of µOP instructions.

The pre-decoder first takes instructions out of IBUFF and stores them in the PIWR at the PD1 stage. It can read instructions from IBUFF across entries, starting at any address. The read instructions are cleared from IBUFF. Note the following restrictions on reading from IBUFF.

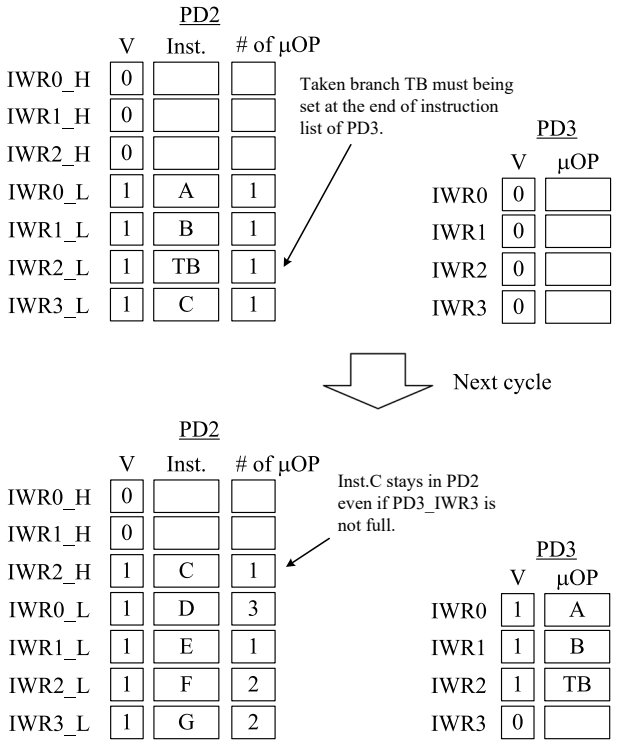
- The restrictions described in Section 4.3 for MOVPRFX instruction packing also apply here. Packing is performed at the PD1 stage. Instructions are read in a way that satisfies those restrictions.
- Although multiple branch instructions can be read at a time, only one branch instruction can be read if "Taken" is predicted for it.
- If "Taken" is predicted for a branch instruction, the branch target of the instruction cannot be read simultaneously.

At the PD2 to PD3 stages, µOP instruction splitting is performed on instructions packed at the PD1 stage. The µOP instruction splitting has the following restrictions.

- The Taken branch instruction is placed at the end in PD3.
- When one architecture instruction is split into three or more µOP instructions, the set of µOP instructions must be placed at the beginning and subsequent positions in PD3.

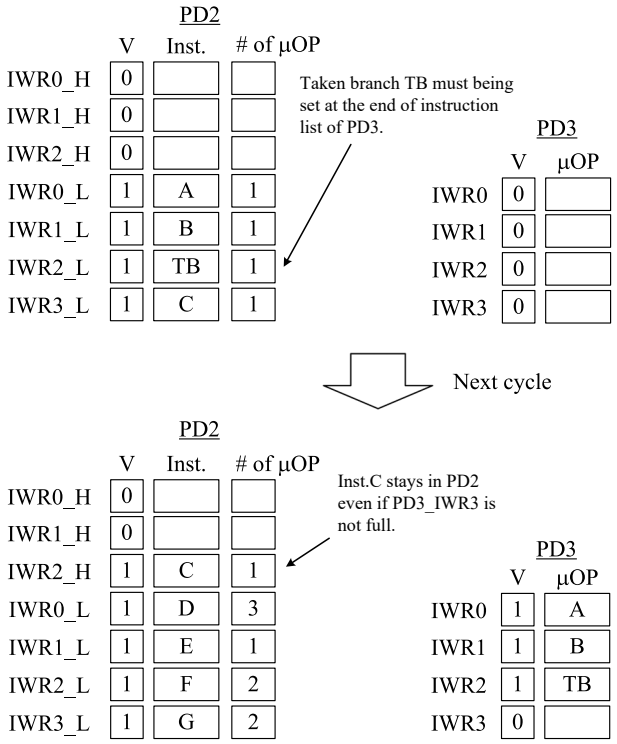
In sequential decode, µOP instructions are expanded at the D stage, where there is a restriction that an instruction subject to sequential decode must be placed at the end. Therefore, when instructions are sent from the PD2 stage to the PD3 stage, the instruction sequence is cut to ensure there is only one target instruction, and the instruction is placed at the end.

Figure 4-4 to Figure 4-6 show respective examples.



TB : Taken branch instruction.

Figure 4-4 Restriction on Taken Branch Instruction When Splitting µOP Instructions



TB : Taken branch instruction.

Figure 4-5 Restriction Related to Three or More µOP Splits Resulting from µOP Instruction Splitting

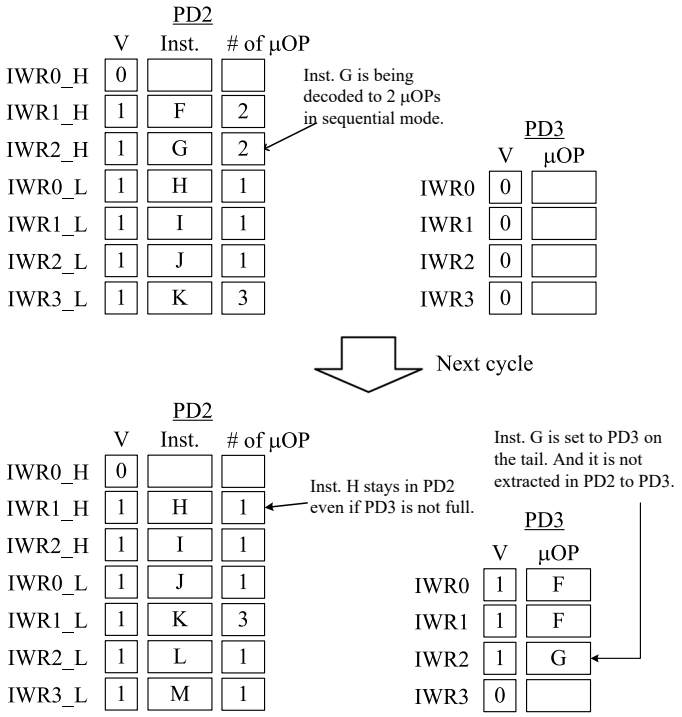


Figure 4-6 Restriction on µOP Instruction Splitting for Sequential Decode

4.4.2. Decode

Decode is performed at the D to DT stages. Mainly, allocation of out-of-order resources and dispatch to reservation stations, as subsequently described, are performed in decode.

The decoder allocates GIDs, CSEs, and physical registers as well as VFPs and VSPs. Up to four µOP instructions per cycle are sent to the decoder from the pre-decoder at the previous stage. This set of input µOP instructions is called a dispatch group, to which resources are allocated. Table 4-1 shows the relation between the main instructions and quantities of allocated resources.

Table 4-1 Relation Between Instructions and Quantities of Allocated Resources

Resource	Allocation Unit	Instruction Type
GID	Dispatch group	All instructions
CSE	µOP instruction	All instructions
Physical register	µOP instruction	Instructions that have destination registers
VFP/VSP	Processing unit for load/store unit	Load/Store instructions

As shown in Table 4-1, a GID is allocated to each dispatch group. On the other hand, one restriction is that a set of four CSEs can have only one GID. If the number of µOP instructions in a dispatch group is less than 4, a GID is allocated to the group when there are unused entries.

The number of allocated FPs and SPs is the processing unit for the load/store unit. For details, see Section 7.3.

µOP instructions are dispatched to reservation stations when the allocation of out-of-order resources is completed.

4.5. Instruction Commit

The commit stage determines speculatively executed instructions. After confirming the branch paths of branch instructions, checking for any exceptions, and judging that completing the instructions is not a

problem, the stage commits the instructions in program order. Then, it determines the architecture state of the processor. As shown in Figure 4-7, CSEs are organized in groups of four, and a GID is allocated to each group. Since µOP instructions are allocated to CSEs in units of GIDs as described above, CSE release (i.e., instruction commit) is also performed in units of GIDs.

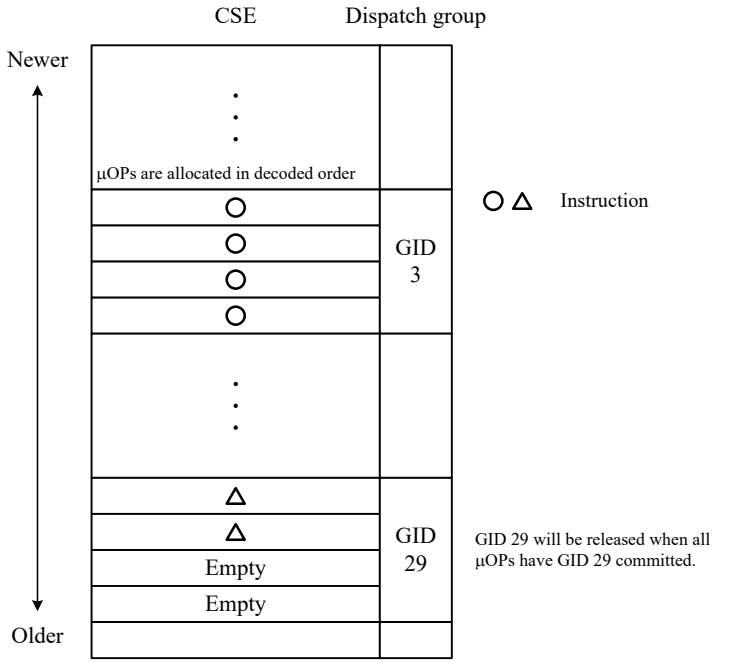


Figure 4-7 CSE Structure

Since instruction commit is performed in program order, it starts with the CSE group that has the oldest instruction among CSEs. The mechanism of instruction commit is as follows.

- Only one GID can be committed at a time. That is, it is not possible to commit across CSE groups.
- Multiple µOP instructions that are allocated the same GID can be committed simultaneously.
- The oldest µOP instruction among those belonging to a committable GID can be committed without waiting for the end of execution of newer µOP operations with the same GID.
- Multiple Taken branch instructions cannot be committed simultaneously.
- If a misprediction of a branch instruction has occurred, instructions up to the branch instruction are committed, and subsequent instructions are discarded.
- If one architecture instruction is split into multiple µOP instructions, commit can be done in each µOP instruction. However, the PC is updated at the point in time when the last µOP instruction is committed.

4.5.1. No Exception Mode

The A64FX has a function to skip the exception judgment and to bring commit forward as possible, when configured not to issue a notification for a floating-point operation exception interrupt. This can shorten pipeline stages, which would reduce the occupation time of out-of-order resources.

No Exception mode is enabled when the FPCR system register settings are as shown in Table 4-2.

Table 4-2 FPCR Register When No Exception Mode Is Enabled

FPCR Register Field	Field Value
FZ, FZ16	1
IDE, IXE, UFE, OFE, DZE, IOE	0

4.6. Pipeline Flush

A pipeline flush occurs when instruction execution results are judged as incorrect in the results of instruction completion judgment at the commit stage. The two types of pipeline flush are described below.

Asynchronous Flush

This occurs when a branch misprediction occurs. The instructions following a branch instruction that has occurred the branch misprediction cannot be committed because they are in the wrong program path. Therefore, the following instructions are discarded. A branch misprediction is found at the point in time when a branch instruction is executed, that is, at the execution stage. When a branch misprediction is found, the front-end is flushed first, and instruction fetch in the correct path begins. Newly fetched instructions are decoded and wait at the D stage. In subsequent, after the instructions preceding the branch instruction that has occurred the misprediction and its branch instruction are committed, the back-end is flushed. After the flush of the back-end is completed, instruction dispatch resumes.

Synchronous Flush

This occurs due to a trap, an exception, a violation of the load/store order guarantee, or other causes that need to discard the internal state in out-of-order execution. If any of these events occurs, the execution of following instructions produces incorrect results, and thus the instructions cannot be committed. Since these events are identified at the instruction commit time, both the front-end and back-end pipelines are flushed at the point in time of the instruction commit. Then, instruction fetch resumes when the processor enters the correct state. In the case of synchronous flush, the front-end cannot be flushed in advance, unlike asynchronous flush. Thus, instruction fetch and decode processing cannot be hidden. Therefore, the penalty imposed until instruction execution resumes is greater than that in asynchronous flush caused by a branch misprediction.

4.7. Particular Instruction Controls

The dependency of some architecture instructions is created via a processor architecture state other than registers and memory. These instructions must be executed after producers have been committed. In addition, for the execution of consumers, it is necessary to guarantee that the producers have completed them. To guarantee these operations, instruction decode and instruction commit have the following two particular instruction controls.

Pre-Sync

Instructions subject to this control remain at the decode stage until the immediately preceding instruction is committed.

Post-Sync

Instructions subject to this control make their consumers remain at the decode stage until they are committed.

These instruction controls are performed only for architecture instructions that require control. "List of Instruction Attribute and Latency" shows instructions subject to the controls.

5. Instruction Dispatch

At the decode stage, scheduling of dispatch to a reservation stations (RS) is performed in addition to allocation of out-of-order resources. The A64FX has multiple RSs, but the execution pipeline connected to each RS varies in its function. Therefore, scheduling is performed in consideration of instruction types and the dependency between instructions.

5.1. Reservation Station

Decoded μOP instructions are dispatched to reservation stations (RSs) in the form of operation-flows. The RSs issue instructions out of order sequentially from the oldest among those that can be executed and allocated to execution pipelines. The RSs of the A64FX are divided into five, each of which has different execution pipelines connected. Table 5-1 shows the number of entries and connected execution pipelines of each RS.

Table 5-1 Number of Entries and Connected Execution Pipelines of Each RS

RS	Number of Entries	Execution Pipeline
RSE0	20	EXA, FLA, PR
RSE1	20	EXB, FLB
RSA0	10	EAGA, EAGB
RSA1	10	
RSBR	19	BR

RSE0 and RSE1 cannot issue instructions to each other's pipelines due to the connection relationship of execution pipelines, whereas RSA0 and RSA1 can do so. Unlike CSEs, FPs, SPs, and other out-of-order resources, entries in the RSs are released when instructions are issued. For details on allocation and release stages, see Table 2-6 in Section 2.8.

Each RS has two write ports and two issue ports. Therefore, the number of instructions that can be dispatched to the same RS is limited to two. Likewise, up to two instructions can be issued from the same RS.

5.2. Instruction Dispatch Attribute

Due to the connection relationship between RSs and execution pipelines, the A64FX limits the RSs to which instructions can be dispatched based on the type of operation-flow. For example, operations that can only be executed in the EXA pipeline can be dispatched only to RSE0. Therefore, the dispatch destination of an operation-flow is strongly tied to the pipeline that can execute the operation-flow. In this section, the RSs to which operation-flows can be dispatched are defined as attributes, shown in Table 5-2, for the purpose of describing the dispatch mechanism. "List of Instruction Attribute and Latency" shows the relevance among architecture instructions, operation-flows, and their execution pipelines.

Table 5-2 Attributes of Instructions and Operation-Flows

Attribute	Dispatch-Enabled RS	Destination Execution Pipeline
RSX	RSE0, RSE1, RSA0, RSA1	EXA, EXB, EAGA, EAGB
RSE	RSE0, RSE1	EXA, EXB, FLA, FLB
RSA	RSA0, RSA1	EAGA, EAGB
RSE0 only	RSE0	EXA, FLA, PR
RSE1 only	RSE1	EXB, FLB

Some instructions require the temporary operand register (TOR) in addition to an RS as the resources necessary for dispatch. The TOR is a register that relays operands from a program counter to a functional unit. Table 5-3 lists instructions that require the TOR.

Table 5-3 Instructions That Require TOR

Attribute	Instruction
TOR	LDR{SW|} (literal) ADR{P|} BL{R|} MRS

5.3. Dependency Group Detection

As described in Section 2.7, a penalty occurs when operand bypass is performed between different execution pipelines in the A64FX. Particularly, the percentage of the penalty for the original operation latency of integer operation instructions is high. Therefore, if instructions have an operand dependency between them, it is desirable to issue the instructions to the same execution pipeline whenever possible. To achieve this, the A64FX detects operand dependencies between instructions at the decode time. The decoder judges that there is a dependency between instructions when all following conditions are satisfied:

- The instructions are arithmetic operation, logic operation, or shift instructions to be executed in the EXA, EXB, EAGA, or EAGB pipeline.
- There is an operand dependency between two consecutive instructions, or both two instructions use the NZCV register.
- The following instruction has the RSX or RSE attribute.

When a dependency between instructions is detected, the target instructions form a dependency group. Dependency detection is only performed between two consecutive instructions. Therefore, only if instructions which are successive have dependencies as shown in Figure 5-1, the dependency group is formed. If two instructions are in different decode windows as shown in Figure 5-2, the decoder can detect a dependency only between the instruction in slot 0 in the window and the instruction in slot 3 in the previous window.

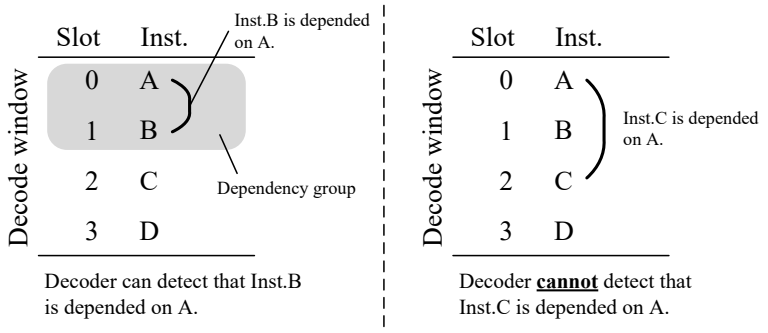


Figure 5-1 Example of Two Instructions That Have Dependency in Same Decode Window

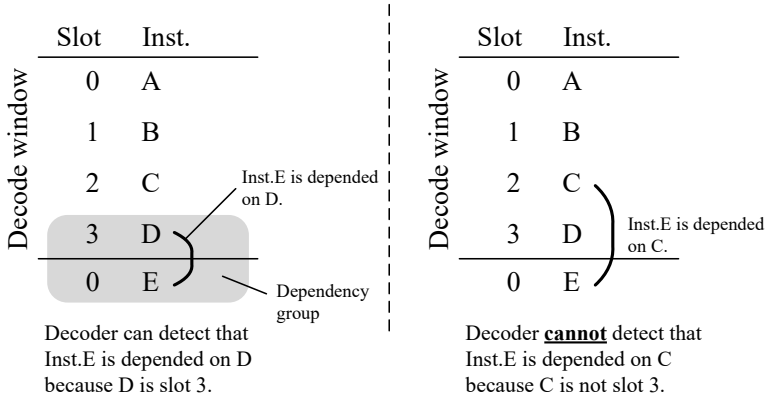


Figure 5-2 Example of Two Instructions That Have Dependency Across Different Decode Windows

5.4. Instruction Dispatch Mechanism

μOP instructions are dispatched as operation-flows when the allocation of CSEs, renaming registers, VFPs, and VSPs is completed. At this time, flow split may be performed for some instructions. Allocation for dispatch to RSs is determined in consideration of dispatch attributes, dependency groups, and the number of entries used in RSs. The decoder has an RS allocation rule for each of the instruction dispatch attributes described above. The rule is used to decide a basic RS allocation destination. This section summarizes individual allocation rules.

Instructions with RSX Attribute

Instructions with the RSX attribute can be dispatched to any of the following RSs: RSE0, RSE1, RSA0, and RSA1. Allocation to RSs is performed based on a table that defines allocation destinations for each decoder slot, such as shown in Table 5-4. This table provides five allocation patterns to balance the number of RSs used. The table uses RSEm/f. RSEm indicates the RS has more free entries, and RSEf indicates the RS has fewer free entries, when compared between RSE0 and RSE1. The table also uses similar indications for the RSA.

Table 5-4 Allocation Table for Instructions with RSX Attribute

	Table 1	Table 2	Table 3	Table 4	Table 5
Slot 0	RSEm	RSEm	RSAm	RSEm	RSAm
Slot 1	RSEm	RSEf	RSAf	RSEf	RSAf
Slot 2	RSEm	RSEm	RSAm	RSAm	RSEm
Slot 3	RSEm	RSEf	RSAf	RSAf	RSEf

The number of free entries in the RSs are a consideration in selecting a table from among these tables. The selection rule is determined by the following conditions and a combination of those shown in Figure 5-3.

- Condition 1: Neither RSA0 nor RSA1 has free entries, and both RSE0 and RSE1 have some free entries. Alternatively, the value obtained by subtracting the total number of free entries in RSA0 and RSA1 from the total number of free entries in RSE0 and RSE1 is equal to or greater than the threshold.
- Condition 2: Neither RSE0 nor RSE1 has free entries, and both RSA0 and RSA1 have some free entries. Alternatively, the value obtained by subtracting the total number of free entries in RSE0 and RSE1 from the total number of free entries in RSA0 and RSA1 is equal to or greater than the threshold.
- Condition 3: The difference between the number of free entries in RSE0 and that in RSE1 is equal to or greater than the threshold.
- Condition 4: Either RSE0 or RSE1 has the greatest number of free entries out of all RSs excluding the RSBR.

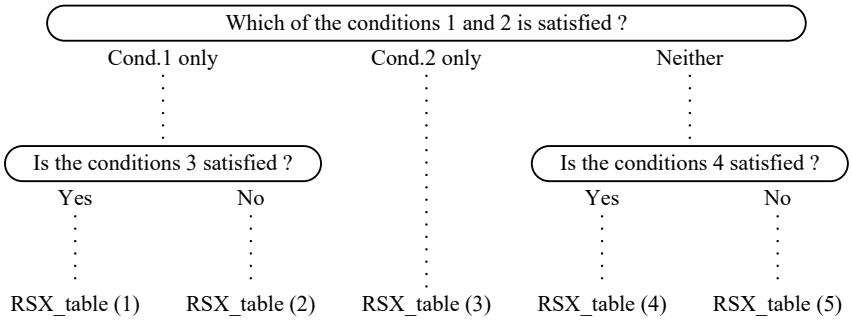


Figure 5-3 Allocation Table Selection Rule for Instructions with RSX Attribute

Instructions with RSE or RSA Attribute

Instructions with either of these attributes can be respectively dispatched to the RSE or RSA only. Therefore, it is enough to set an allocation destination for number 0 or 1 in each RS. Table 5-5 shows an allocation table.

Table 5-5 Allocation Table for Instructions with Either RSE or RSA Attribute

	Table 6	Table 7
Slot 0	RS{E|A}m	RS{E|A}0
Slot 1		RS{E|A}1
Slot 2		RS{E|A}0
Slot 3		RS{E|A}1

An allocation table is selected based on condition 5 and a combination from Figure 5-4.

- Condition 5: RSEm or RSAm has some free entries, and neither RSEf nor RSAf has free entries.

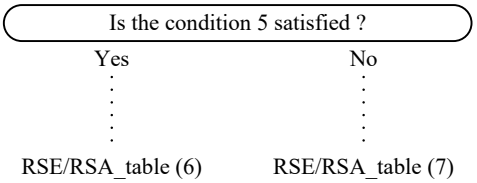


Figure 5-4 Allocation Table Selection Rule for Instructions with RSE or RSA Attribute

Instructions with RSE0 Only or RSE1 Only Attribute

Since instructions with either of these attributes have only one allocatable RS, there is no allocation table and they are allocated to the RS indicated by the attribute.

Instructions in Dependency Group

Of the instructions that form a dependency group, only the first instruction in the group is allocated based on instruction attributes. The implicit decision is to allocate following instructions to the same RS as the first instruction.

The RS to which to allocate the instruction in each slot at the decode stage is individually decided based on the above allocation rules. Therefore, three or more instructions may be allocated to the same RS over the entire decode stage. Since each RS has only two write ports, up to two instructions can be dispatched in the same cycle. In this situation, only the first two instructions are dispatched, and the remaining instructions are not dispatched. They are dispatched afresh in the next cycle.

Instructions that use the TOR have a restriction that the TOR must have an empty entry and only one instruction can be dispatched in the same cycle.

6. Instruction Execution

Operation-flows dispatched to reservation stations (RSs) are scheduled out of order and issued to execution pipelines. The issued flows are executed by the functional units implemented in execution pipelines. In a broad sense, instruction execution also includes memory access by load/store instructions. However, the execution pipelines covered in this chapter are the pipelines that perform operations for operation instructions, and parts of address calculation stages that calculate effective addresses for load/store instructions.

6.1. Instruction Issue

Operation-flows wait in RSs until the source operands are ready, at which time the flows transition to an executable state. From the flows that can be executed, the RSs select and issue older flows whose dispatch destination pipelines are available. This also applies to load/store operation-flows that are scheduled in the RSA. Though RSE0, RSE1, and the RSA are basically controlled independently for issuing operation-flows, they are controlled simultaneously when multiple execution pipelines must work together to execute a flow. Since either RSA0 or RSA1 can submit flows to the EAGA and EAGB pipelines, they are always controlled synchronous with each other.

Flows that are subject to operation-flow splitting at the execution stage are split at this time. That is, two or more flows are issued from one entry in an RS.

6.2. Execution Pipeline

Table 6-1 shows combinations of execution pipelines and main functional units. The execution pipelines are roughly categorized into five groups, each of which is equipped with an operation unit that is an assembly of functional units.

Table 6-1 Execution Pipelines

Pipeline Group	Pipeline	Function
Integer operation pipelines	EXA	Arithmetic & logic, shift, multiplication
	EXB	Arithmetic & logic, shift, division
Address calculation pipelines	EAGA	Address calculation, arithmetic & logic
	EAGB	
Floating-point operation pipelines	FLA	Integer arithmetic & logic, shift, floating-point arithmetic & multiply-add, floating-point division, crypto calculation, vector address calculation
	FLB	Integer arithmetic & logic, shift, floating-point arithmetic & multiply-add
Predicate operation pipeline	PR	Predicate manipulation
Branch pipeline	BR	

The integer operation pipelines are mainly responsible for integer instruction operations. The address calculation pipelines are equipped with a functional unit for effective address generation and an integer ALU (Arithmetic Logic Unit) with partial functions that enable the pipeline to perform some integer operation instructions. The floating-point operation pipelines can execute SIMD&FP and SVE operation instructions.

As shown in Figure 6-1, each operation pipeline is allocated to a reservation station (RS). Although each pipeline basically operates independently, one restriction does not allow instructions to be simultaneously

submitted to the EXA pipeline and the predicate operation pipeline because they share the instruction issue port in RSE0.

The address calculation pipelines are connected to load/store pipelines. The results of calculating effective addresses for load/store instructions are sent directly to the load/store pipelines.

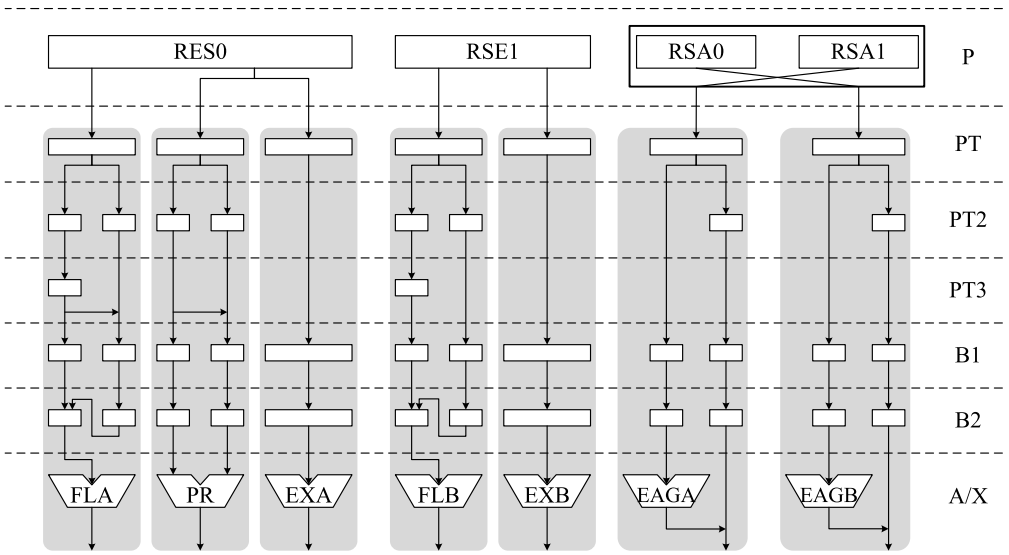


Figure 6-1 Outline of Execution Unit

6.3. Blocking Control

Some operation-flows cannot be performed with pipeline processing. The control for handling those operation-flows is called blocking. The blocking control includes pipeline blocking and operation blocking. During the execution of an operation in a pipeline, pipeline blocking prevents the pipeline from accepting the issuing of following instructions until the operation ends. During the execution of an instruction with an operation blocking attribute, operation blocking prevents the pipeline from accepting the issuing of instructions with the same operation blocking attribute until the instruction ends. Pipeline blocking and operation blocking differ in the following way: during pipeline blocking, no following instruction can be issued to the pipeline; and during operation blocking, the only instructions that cannot be issued are following instructions of the operation blocking type. For example, an SDIV instruction, which is of the operation blocking type, requires 9 to 42 cycles for execution. In this situation, instructions of the operation blocking type cannot be issued until the execution is completed, but other instructions can be issued. "List of Instruction Attribute and Latency" shows the blocking attribute of each instruction.

6.4. Physical Register File

The implemented physical register files of the A64FX are categorized as architecture register types. Figure 6-2 shows the connection relationship of read ports, write ports, and execution pipelines for the physical register files.

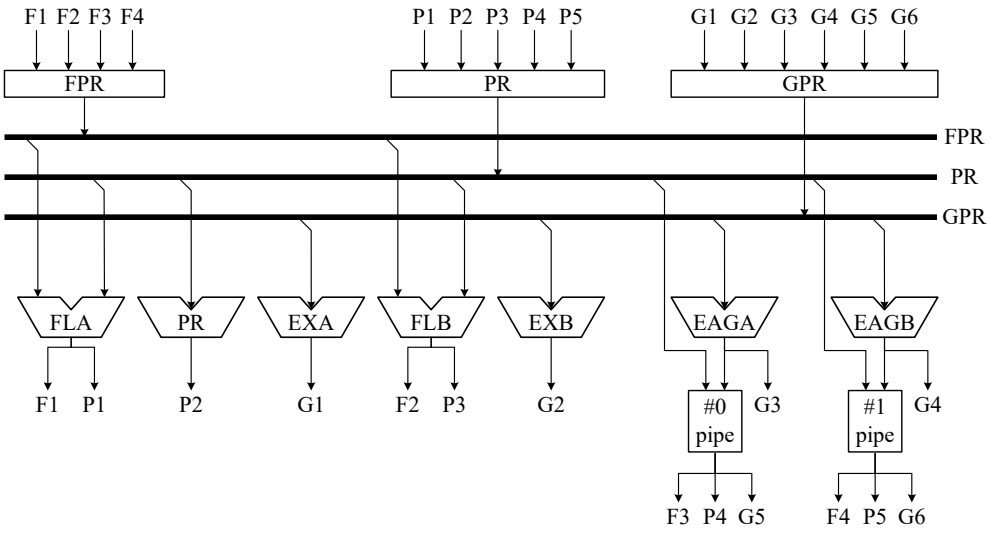


Figure 6-2 Connection Relationship Between Physical Register Files and Execution Pipelines

6.5. Execution of Particular Instructions

6.5.1. SVE Instruction with Merging Predication

An SVE instruction specified to merging predication can update a destination register partially based on a condition of the predicate register. In this operation, inactive elements in the destination register keep their previous value. A64FX is implemented the conditional selection and concatenation operation between the destination register and the result of calculation as the operation to retain the previous value. Note that the destination register of the instruction is also used for their source operand the same as the destructive instruction. Furthermore, it can inhibit to use the destination register as the source operand if the instruction is specified to zeroing predication with MOVPRFX instruction.

On the other hand, as described in Section 4.3, A64FX packs the instruction modified with MOVPRFX instruction and decodes it as one architecture instruction. The number of splits for µOP instructions from a packed instruction is basically equal to that from its original instruction. However, only if the MOVPRFX modification with merging predication, an additional µOP instruction might be generated. It is because the merging operation needs to read from the destination register as described above, and the number of source operands for packed instruction might be over the upper limit of the number of reads per one instruction depending on the original instruction. In this case, A64FX generates an extra µOP for reading from the register to avoid the limitation of the number of reads a register. The extra µOP is one at the most and its latency is 4 cycles. Note that the extra µOP has a dependency on the last calculation µOP, which is generated from its original instruction. "List of Instruction Attribute and Latency" shows the information whether an instruction needs an extra µOP if it is modified with MOVPRFX instruction and predicated merging.

6.5.2. Inter-Register-File MOV Operation

An instruction that transversely uses general-purpose, floating-point, and predicate registers requires particular control to transfer values from the respective registers. This is because the read and write ports of each physical register are not connected to all execution pipelines. Thus, transfer is processed by synchronizing multiple pipelines. Since the control varies depending on the combination of registers and the transfer direction, this section provides an explanation with examples.

Transfer from General-Purpose Register to Floating-Point Register

This behavior is performed when an instruction, such as an FMOV (general) or SCVTF instruction, specifies a general-purpose register as the source register and the SIMD&FP register as the destination register. These instructions are divided into two operation-flows and executed. One operation-flow is executed in EXA (EXA flow), which reads the source operand from the general-purpose physical register.

The other operation-flow is executed in FLA (FLA flow), which writes the operand to the floating-point physical register. Figure 6-3 shows a time chart of flow execution.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
EXA flow	D	DT	P	PT	B1	B2	X												
FLA flow					P	PT	PT2	PT3	B1	B2	X1	X2	X3	X4	X5	X6	U C	UT W	W2

Figure 6-3 Flow Time Chart of Transfer Instruction from General-Purpose Register to Floating-Point Register

Transfer from Floating-Point Register to General-Purpose Register

This behavior is performed when an instruction, such as an FMOV (general) or FCVTZ* instruction, specifies the SIMD&FP register as the source register and a general-purpose register as the destination register. These instructions use a load/store pipeline to transfer an operand. They are divided into two flows and executed. One is an operation-flow executed in FLA (FLA flow), which reads the source operand from the floating-point physical register. The other is the LD flow executed in a load/store pipeline, which writes the operand to the general-purpose register. The LD flow can be executed in either pipeline 0 or 1. Figure 6-4 shows a time chart of flow execution. Note that the timing of LD flow execution may be even later since the FLA flow and the LD flow are actually asynchronous.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
FLA flow	D	DT	P	PT	PT2	PT3	B1	B2	X	U	UT	UT2	UT3	UT4									
1st LD flow			P	PT	B1	B2	A	T	M	B	R	RT											
2nd LD flow														A	T	M	B	R	RT	RT2	C	W	W2

Figure 6-4 Flow Time Chart of Transfer Instruction from Floating-Point Register to General-Purpose Register

6.5.3. Denormalized Number Operation

The A64FX performs floating-point denormalized number operations with hardware in a special processor mode. Thus, latency and throughput in denormalized number operations are completely different from those in normalized number operations. For example, the latency of a double-precision FADD (scalar) instruction is about 90 cycles. Since each element of operation processing is completely in-order and blocked in this mode, the execution time increases in proportion to the number of operation-flows for the operation.

7. Memory Access

Memory access is processed in load/store pipelines. After an effective address is calculated, a load/store operation-flow issued from a reservation station is submitted to a load/store pipeline. The load/store pipeline performs virtual address translation and accesses the L1D cache. If the operation-flow is a load operation-flow, it reads data. If the operation-flow is a store operation-flow, it writes data. The load/store pipeline also handles the processing when a cache miss occurs.

7.1. Overview of Load/Store Pipeline

Figure 7-1 shows the main configuration modules and pipeline stages of the load/store pipelines.

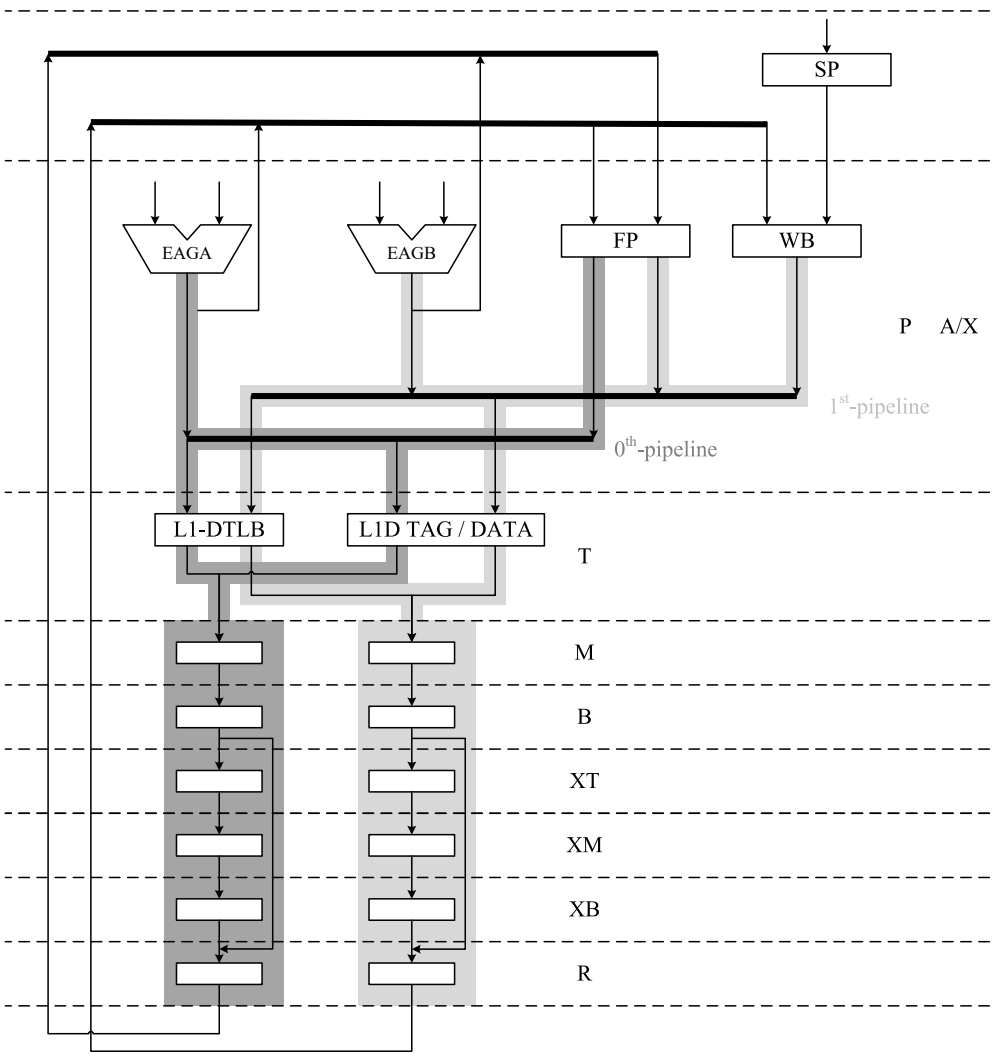


Figure 7-1 Outline of Load/Store Unit

The rest of this section summarizes the role of each module in the figure.

Fetch port (FP)

Queue used to manage the execution order of load/store instructions. The FP is equipped with the virtual fetch port (VFP), which manages only the order of instructions, and the real fetch port (RFP), which manages the load/store access order as well. The VFP has load/store instructions allocated in order at the decode time. The RFP has load/store operation-flows allocated out of order

when they are issued. From the point in time when the execution of instructions is completed, the VFP and RFP can both be released in this order.

Store port (SP)

Queue used to manage the execution order of store instructions and store data. The SP is equipped with the virtual store port (VSP), which only manages the order of instructions, and the real store port (RSP), which manages the load/store access order as well. The VSP has store instructions allocated in order at the decode time. The RSP has store operation-flows allocated out of order when they are issued. Both the VSP and RSP are released when the store instructions are committed, and the store data is written to the write buffer.

Write buffer (WB)

Buffer that temporarily holds store data from the SP before the data is written to the L1D cache. The buffer is provided to separate the store instruction commit operation from the L1D cache write operation.

L1-DTLB

Primary TLB that holds the address translation information required for data access

L2-DTLB

Secondary TLB that holds the address translation information required for data access. Although this TLB does not appear on the pipeline, it is implemented in the load/store unit.

L1D cache

RAM that holds L1D cache data and tag information.

EAGA, EAGB

Functional units that generate effective addresses for load/store instructions. Although they are categorized as operation units, they are positioned at the P stage of the pipeline stages.

The load/store unit is equipped with two pipelines (0^{th}-pipeline, 1^{st}-pipeline). The two pipelines have basically the same functions. However, only pipeline 1^{st}-pipeline can process write requests to the L1D cache.

The load/store pipelines have two functional modes: short and long. Basically, the mode that is used depends on the instruction. Table 7-1 shows the functional mode of each instruction and the load-to-use latency of each load instruction. However, if any operation-flows conflict with each other at a later stage, latency changes as described in Section 2.9, "Execution Latency Changing."

Table 7-1 Latencies of Load/Store Instructions

Instruction	Functional Mode	Load-to-Use Latency	After Latency Change
Integer load instruction	Short	5 cycles	8 cycles
Integer store instruction	Short	-	-
SIMD&PF load instruction	Short	8 cycles	11 cycles
SIMD&FP store instruction	Short	-	-
SVE load instruction, and part of SIMD&PF instructions	Long	11 cycles	-
SVE store instruction	Long	-	-
Predicate load instruction	Long	9 cycles	-
Predicate store instruction	Long	-	-

7.2. Basic Execution Mechanism of Load/Store

The load/store pipelines mainly process memory access for load/store instructions and data fill/writeback for the L1D cache. They are processed in the following basic operation-flows:

LD flow

Flow that performs memory access for load instructions. It performs both virtual address translation and L1D cache access.

ST0 flow

Flow that performs virtual address translation for store instructions and tag access to the L1D cache. It does not write data to the L1D cache. It accesses tags to check L1D cache hits.

ST2 flow

Flow that writes store data to the L1D cache. It can be executed only in 1st-pipeline.

MI flow

Flow that performs a move-in operation for data fill. Move-in for one cache line requires four flows. Pipelines 0th and 1st are synchronized and execute two flows each.

MO flow

Flow that performs a move-out operation for writeback. Basically, move-out for one cache line requires four flows. However, it requires two flows when writing back to a cache line that is clean. Pipelines 0th and 1st are synchronized and execute four or two flows each.

7.2.1. Load Instruction

This section describes basic load instruction behavior.

1. A load instruction is decoded into a µOP instruction that is allocated to a VFP entry. The load µOP instruction is dispatched to the RSA as one LD flow.
2. When the LD flow can be executed, it is issued from the RSA. EAGA/EAGB calculates an effective address and submits the LD flow to 0th/1st-pipeline.
3. When selected by the arbitration scheduler, the LD flow submitted from EAGA/EAGB is submitted to pipeline 0th/1st without a penalty. At the same time, the effective address is written to an RFP entry. Not only EAGA/EAGB but also the RFP and WB are connected to 0th/1st-pipeline. The pipeline executes operation-flows for them as well. The arbitration scheduler decides which operation-flow to execute.
4. If the LD flow submitted from EAGA/EAGB is not selected by the scheduler in step 3, it is not submitted to 0th/1st-pipeline. The LD flow waits in an executable state in the RFP.
5. The LD flow performs address translation with L1D-TLB and in concurrently reads a tag and data from L1D cache in 0th/1st-pipeline.. If the LD flow hits L1-DTLB and the L1D cache, it reads data along the predefined pipeline stages and writes data in a register. The execution of the LD flow is completed.
6. If the LD flow does not hit L1-DTLB in step 5, it searches L2-DTLB and then the translation table to obtain virtual address translation information. If an L1D cache miss occurs, data fill is performed by obtaining data from a lower cache level. In such cases, the MI flow and MO flow is executed.
7. After the L1-DTLB or L1D cache miss is resolved, the LD flow is submitted from the RFP to 0th/1st-pipeline again and executed.
8. Once the execution of the LD flow is completed, the load µOP instruction can be committed. If the preceding load has been completed, the entries in the VFP and RFP are released without waiting for commit.

The processing of the LD flow is basically completed in one flow, provided that the flow hits L1-DTLB and the L1D cache. However, if a cache miss occurs, the LD flow must be resubmitted to the pipeline. In such cases, the LD flow is executed multiple times. In addition, each multiple-structures instruction and gather/scatter instruction, which are described later, is executed after being split into multiple flows depending on the address pattern even when they hit the cache.

7.2.2. Store Instruction

This section describes basic store instruction behavior.

1. A store instruction is decoded into a µOP instruction that is allocated to VFP and VSP entries. The store µOP instruction is divided into the ST0 flow and a data transfer flow before being dispatched. The ST0 flow is dispatched to the RSA. The data transfer flow is dispatched to RSE0. Data transfer flows are specific to store instructions, and the data transfer flows are used to transfer store data from a register to the RSP.
2. When the data transfer flow can be executed, it is issued from RSE0. The execution of this data transfer flow is asynchronous with the ST0 flow. The one that is executed first is undetermined.
3. When the ST0 flow can be executed, it is issued from the RSA. EAGA/EAGB calculates an effective address and submits the ST0 flow to 0th/1st-pipeline.
4. When selected by the arbitration scheduler, the ST0 flow submitted from EAGA/EAGB is submitted to 0th/1st-pipeline. At the same time, the effective address is written to RFP and RSP entries.

5. If the ST0 flow is not selected by the arbitration scheduler in step 4, it is not submitted to $0^{th}/1^{st}$-pipeline but waits in an executable state in the RFP.
6. The ST0 flow performs address translation with L1-DTLB and only reads a tag from L1D cache in $0^{th}/1^{st}$-pipeline concurrently. If the ST0 flow hits L1-DTLB, it stores the physical address after address translation in the SP. If it does not hit the L1D cache, it initiates a data request to a lower cache level. In either case, the execution of the ST0 flow is completed.
7. If the ST0 flow does not hit L1-DTLB in step 6, it searches L2-DTLB and then the translation table to obtain virtual address translation information. After the information is obtained, the ST0 flow is resubmitted.
8. For the data request initiated in step 6, cache miss processing is performed independently of the execution of the ST0 flow. At this time, the MI and MO flows are executed to perform data fill.
9. Once the execution of the ST0 and data transfer flows is completed, the store µOP instruction can be committed. The store physical address and data are moved from the RSP to the WB when the store µOP instruction is committed.
10. The ST2 flow is started from the WB and submitted to 1^{st}-pipeline. If the ST2 flow hits the L1D cache, data is written, and the execution of the ST2 flow is completed.
11. If an L1D cache miss occurs in step 10, cache miss processing is performed again. After the cache miss is resolved, the ST2 flow is resubmitted and data is written.

The ST0 flow of a store instruction does not write data to L1D cache, but the ST2 flow does after the instruction is committed. In addition, it is necessary to pay attention for any L1D cache miss that may occur in each of the ST0 and ST2 flows.

7.3. Fetch Port/Store Port

7.3.1. Virtual Fetch Port/Virtual Store Port

The A64FX is equipped with the virtual fetch port (VFP) and virtual store port (VSP) in addition to the fetch port and store port, which have the original functions. The VFP/VSP manages only the program order of load/store instructions. In addition to doing that, the fetch port/store port manages the execution order of memory access by holding load/store addresses. This manual refers to the original fetch port and store port as the real fetch port (RFP) and real store port (RSP), respectively, in order to distinguish them from the virtual fetch port and virtual store port. Figure 7-2 shows the relationship between the VFP/VSP and RFP/RSP. A window as part of the VFP/VSP are mapped to the RFP/RSP. Mapping to RFP/RSP performs when an operation-flow is issued from a reservation station. Mapping exceeding the number of RFP/RSP entries is not possible. In the event of such mapping, instruction issue is limited. While the VFP/VSP is allocated at the decode time, there is no problem with mapping the RFP/RSP at the instruction execution time, which can reduce the number of RFP/RSP entries required for instruction decode. This prevent stalls caused by a shortage of RFP/RSP entries at the decode time.

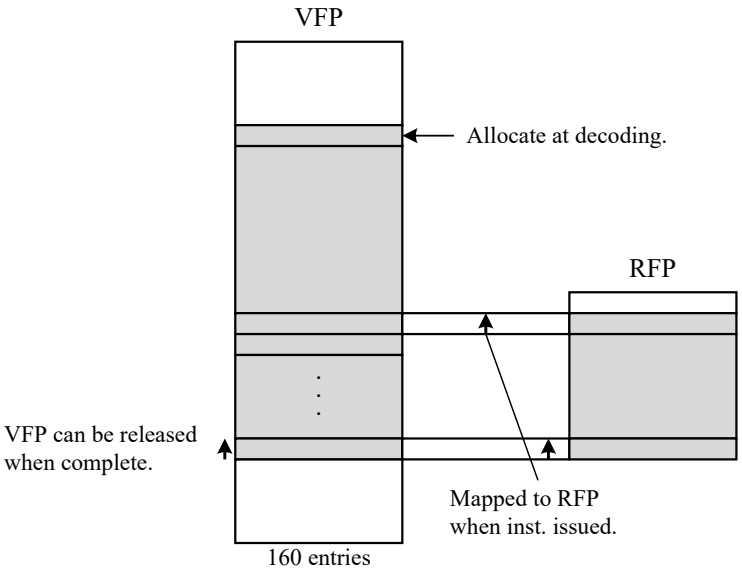


Figure 7-2 Relationship Between VFP/VSP and RFP/RSP

7.3.2. Fetch Port/Store Port Allocation

The number of allocated FP/SP entries is predetermined according to the instruction type. Basically, for load instructions, one FP entry is allocated to each μOP operation. For store instructions, one FP entry and one SP entry are allocated to each μOP instruction. On the other hand, for SVE gather/scatter instructions and SVE LD[234][BH]/ST[234][BH] instructions, multiple FP/SP entries are allocated to each μOP instruction. "List of Instruction Attribute and Latency" shows the number of allocated FP/SP entries for each instruction.

7.4. Write Buffer

In the A64FX, data of committed store instruction is moved to the WB and then written to the L1D cache by the scheduler at an arbitrary timing. The purpose of this process is to reduce commit stalls by separating two operations, instruction commit and write to cache. The WB consists of eight entries, each of which is basically 64 bytes. Since the WB holds data immediately before writing to the L1D cache, it has a restriction regarding address alignment, unlike the SP. Data held in the WB requires appropriate address alignment according to its size. Due to the alignment restriction, an operation called data merge or data split may occur when data is written to the WB.

Figure 7-3 shows examples of data merge and data split. If the WB holds write data with a length of 64 bytes, the alignment must be 64 bytes. If the address of store data is not aligned to 64 bytes, the data is split on a 64-byte boundary and then written to the WB. On the other hand, if a preceding entry of the WB has empty space at the point in time when store data is written from the SP, data can be partially written to the entry and then merged. Since multiple merges can be performed, the number of write times to the L1D cache may be relatively low for instructions with a shorter data length. Note that WB merge is performed only within memory ordering restrictions and therefore may not be performed even when the above condition is satisfied.

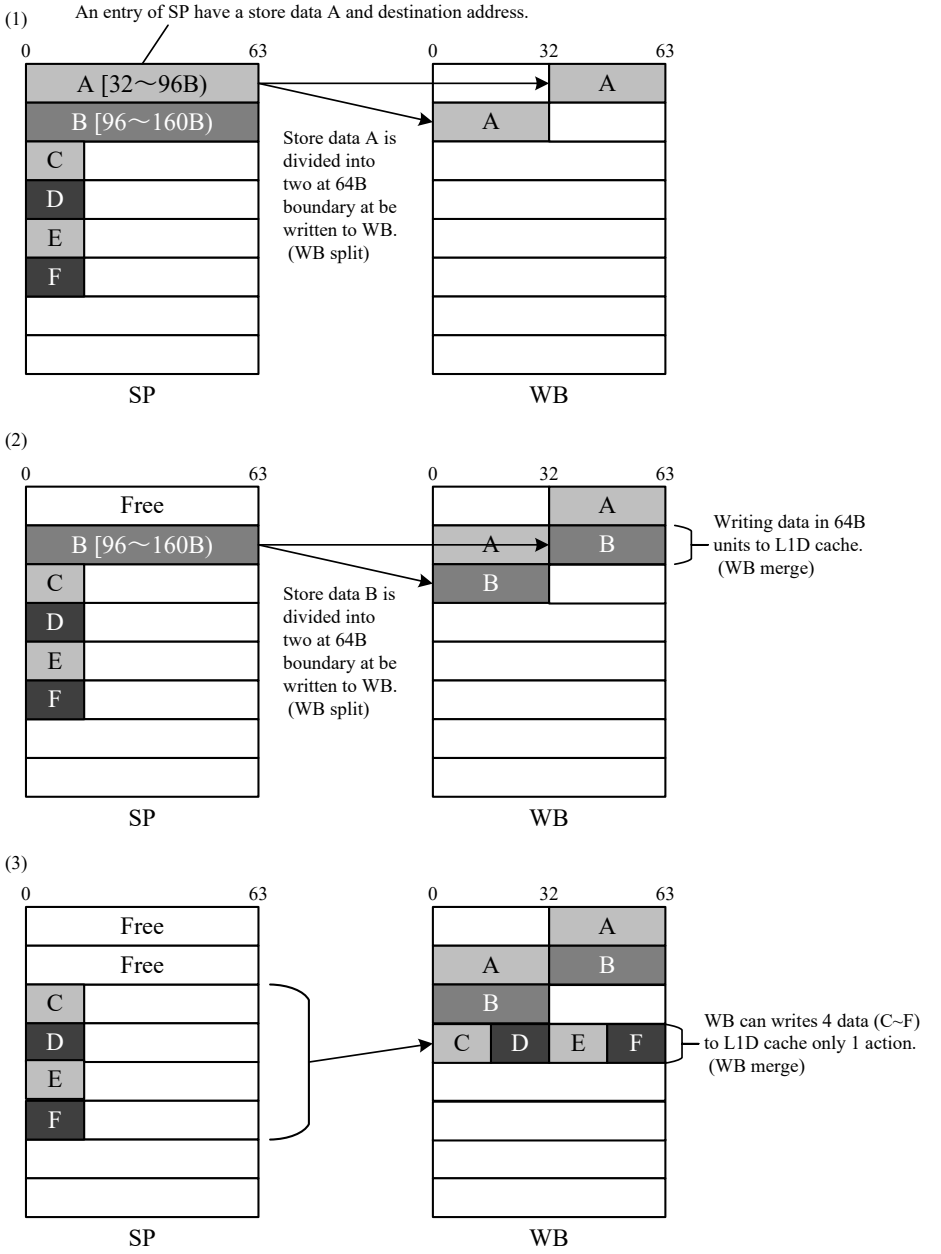


Figure 7-3 Store Data Write from SP to WB

The data length managed by WB entries and the behavior of the merge function vary depending on the instruction. Table 7-2 shows the correspondence.

Table 7-2 Data Length and Merge Function Availability for Each Instruction Managed by WB Entry

	Instruction	Data Length	Merging
Integer instruction	STLR*, STNP, STP, STR*, STTR*, STUR*	16 bytes	✓
SIMD&FP instruction	STNP, STP, STR, STUR	16 bytes	✓
	ST[1234] (single structure)	16 bytes	✓
	ST1 (multiple structures) − {1D|2D|2S|4H|8B}	16 bytes	✓
	ST1 (multiple structures) − {4S|8H|16B}	64 bytes	✓
	ST[234] (multiple structures)	64 bytes	✓
SVE instruction	ST1[BHWD] (contiguous)	64 bytes	✓
	ST1[BHWD] (scatter)	64 bytes	✓
	ST[234][BH]	64 bytes	✓
	ST[234][WD]	128 bytes	
	STR (vector), STR (predicate)	64 bytes	

7.5. Out-of-Order Execution of Load/Store

Basically, the A64FX executes load/store instructions out of order. On the other hand, it must execute memory-dependent store and load instructions in the correct order by detecting dependency. The order of memory-dependent store and load instructions is basically guaranteed by the following two mechanisms.

Store Fetch Interlock (SFI)

This mechanism makes the execution of a following load instruction wait until the preceding store instruction is written to the L1D cache, that is, until the completion of the ST2 flow. When the physical address of the preceding store instruction is determined, the LD flow of the following load instruction is submitted to $0^{th}/1^{st}$-pipeline. However, if its address matches the address of the preceding store instruction, execution is canceled, and its LD flow wait in the RFP until the store completed.

Pipeline Flush

This mechanism is used when the physical address of the preceding store instruction has yet to be determined. Memory dependency with the following load instruction cannot be detected while the physical address of the preceding store instruction is undetermined. Thus, the LD flow of the following load instruction is speculatively executed. After that, when the ST0 flow of the preceding store instruction is executed, the address of the executed load instruction is compared. If the memory dependency is detected, the pipeline flush is performed when the preceding store is committed. And the load instruction is re-executed.

7.5.1. Store Fetch Bypass

As described above, the store fetch interlock (SFI) has a significant impact on performance because it stops the execution of a following load instruction until the store operation is fully completed. To lessen this penalty, store fetch bypass (SFB) is implemented. SFB is a function that allows the following load instruction to read data of the preceding store instruction from the SP or WB. This enables the load instruction to be executed without waiting for the completion of the store instruction. However, due to hardware implementation restrictions, SFB cannot be executed for every combination of load and store instructions. Table 7-3 and Table 7-4 show the combinations for which SFB is available. As described above, the WB has an alignment restriction and does not allow bypass data to be read from across two entries. Therefore, data subject to bypass must be contained in one entry of the WB to execute SFB.

Table 7-3 SFB Availability for Each Combination of Load and Store Instructions

Store Instruction \ Load Instruction	LD (1)	LD (2-1)	LD (2-2)	LD (4-1)	LD (4-2)	LD (8-1)	LD (8-2)	LD (16)	LD (32)	LD (64)
ST (1)	✓									
ST (2-1)	✓	✓								
ST (2-2)			✓							
ST (4-1)	✓	✓		✓						
ST (4-2)					✓					
ST (8-1)	✓	✓		✓		✓				
ST (8-2)							✓			
ST (16)								✓		
ST (32)									✓	
ST (64)										✓

Table 7-4 Specific Instructions of Each Group Shown in SFB Availability Table

Group Name	Instruction	Group Name	Instruction
LD (1)	Length = 1 byte LDR*B (general) LDTR*B (general) LDR (SIMD&FP) – B LDUR (SIMD&FP) – B	ST (1)	Length = 1 byte STRB (general) STR (SIMD&FP) – B STUR (SIMD&FP) – B
LD (2-1)	Length = 2 bytes LDR*H (general) LDTR*H (general) LDR (SIMD&FP) – H LDUR (SIMD&FP) – H	ST (2-1)	Length = 2 bytes STRH (general) STR (SIMD&FP) – H STUR (SIMD&FP) – H
LD (2-2)	Length = 2 bytes LDR (predicate) : VL = 128-bit	ST (2-2)	Length = 2 bytes STR (predicate) : VL = 128-bit
LD (4-1)	Length = 4 bytes LDR (general) – W LDTR (general) – W LDR (SIMD&FP) – S LDUR (SIMD&FP) – S	ST (4-1)	Length = 4 bytes STR (general) – W STR (SIMD&FP) – S STUR (SIMD&FP) – S
LD (4-2)	Length = 4 bytes LDR (predicate) : VL = 256-bit	ST (4-2)	Length = 4 bytes STR (predicate) : VL = 256-bit
LD (8-1)	Length = 8 bytes LDR (general) – X LDTR (general) – X LDR (SIMD&FP) – D LDUR (SIMD&FP) – D LD1 (multiple structure, 1 register) - {8B|4H|2S|1D}	ST (8-1)	Length = 8 bytes STR (general) – X STR (SIMD&FP) – D STUR (SIMD&FP) – D ST1 (multiple structure, 1 register) - {8B|4H|2S|1D}
LD (8-2)	Length = 8 bytes LDR (predicate) : VL = 512-bit	ST (8-2)	Length = 8 bytes STR (predicate) : VL = 512-bit
LD (16)	Length = 16 bytes LDR (vector) : VL = 128-bit	ST (16)	Length = 16 bytes STR (vector) : VL = 128-bit
LD (32)	Length = 32 bytes LDR (vector) : VL = 256-bit	ST (32)	Length = 32 bytes STR (vector) : VL = 256-bit
LD (64)	Length = 64 bytes LDR (vector) : VL = 512-bit	ST (64)	Length = 64 bytes STR (vector) : VL = 512-bit

7.5.2. Restriction of Out-of-Order Execution

Basically, executing load/store instructions out of order is not a problem if different addresses are to be loaded/stored. However, due to hardware implementation restrictions, address match detection is not ideally designed. Therefore, memory dependency is detected in a pseudo manner under some conditions, resulting in limitations in out-of-order execution. This section summarizes the conditions.

- Restriction on inactive elements of predicate-modified load/store instructions
  In memory dependency detection, all the elements of load/store instructions are handled as active. Therefore, pseudo memory dependency occurs between the addresses of elements that are originally inactive. If all the elements of a store instruction are exceptionally inactive, pseudo memory dependency does not occur because the store operation itself is omitted.
- Restriction on gather load instructions
  When an operation-flow for two elements paired in a gather load instruction is executed without being split, the entire cache line that contains the pair is used as the unit of memory dependency detection. Pseudo memory dependency occurs outside the actual range of access of the two elements.
- Restriction on load/store instructions with memsize less 4 bytes
  For SIMD&FP vector load/store instructions and SVE load/store instructions, memory dependency detection is performed on every 4-byte boundary. The start and end addresses are extended so that each of them becomes a 4-byte boundary address. Therefore, pseudo memory dependency occurs in the extended part.
- Restriction on multiple structures instructions
  Memory accesses of multiple structure instructions are performed in unit of registers as described in chapter 7.8.1. Accessed space for each its memory access is dealt as product of its memsize, the number of elements and the number of registers. And memory dependency is detected in unit of cache lines. Thus, the cache lines subject to the detection are all cache lines included into the accessed space. As a result, pseudo memory dependency occurs outside the actual range of access of the instruction. Moreover, for SVE LD[234]/ST[234] instructions, the vector length is always set as 512-bit.
- Restriction on load/store instructions across a 4-KiB boundary
  In access across a 4-KiB boundary by a load/store instruction aligned in units of at least memsize, memory dependency detection is performed with incomplete physical address. Therefore, pseudo memory dependency may occur even when the physical address is actually different.
- Restriction on in-flight load/store instructions
  The detection of memory dependency between load and store instructions in flight in load/store pipelines is performed with incomplete physical address. Therefore, pseudo memory dependency may occur even when the physical address is different. This restriction does not occur when the operation-flow waits at the FP or WB.
- Restriction due to an L1D cache miss of a store instruction
  When a store instruction results in an L1D cache miss, the following load instruction performs memory dependency detection with incomplete physical address. Therefore, pseudo memory dependency may occur even when the physical address is different. This restriction is eliminated when the cache miss of the store instruction is resolved.

7.6. Operation-Flow Conflict

Load/Store 0^{th}-pipeline and 1^{st}-pipeline differ in function. Therefore, they cannot execute arbitrary operation-flows, and some flows have a restriction on executable pipelines. In addition, some flows cannot be submitted to 0^{th}-pipeline and 1^{st}-pipeline simultaneously, depending on the combination of the flows. The conditions for use of the pipelines by each operation-flow are described below.

- MI flow, MO flow
  These two flows are always paired and executed in 0^{th}-pipeline and 1^{st}-pipeline simultaneously. That is, they are not executed simultaneously with other flows.
- LD flow
  This flow can be executed in either 0^{th}-pipeline or 1^{st}-pipeline. However, it cannot be executed simultaneously with the ST2 flow.
- ST0 flow

This flow can be executed in either 0th-pipeline or 1st-pipeline. The ST0 flow of some store instructions can be executed simultaneously with the ST2 flow.

- ST2 flow

  This flow can be executed only in 1st-pipeline. At this time, 0th-pipeline can simultaneously execute only the ST0 flow of some store instructions.

The only operation-flow that can be executed simultaneously with the ST2 flow is the ST0 flow. However, due to implementation restrictions, this does not apply to the ST0 flow of every store instruction. Table 7-5 shows store instructions for which ST0 flows can be executed simultaneously and their conditions.

Table 7-5 ST0 Flow Conditions

	Instruction	Condition
Integer instruction	ST{T|U|}R − X STP{N|}P − X	Addresses are at least 8-byte aligned.
SIMD&FP instruction	ST{U|}R − [DQ] ST{N|}P − [DQ] ST1 (single structure) − D ST1 (multiple structures) ST[234] (single structure) – D	
SVE instruction	ST[1234]D (contiguous) STR (vector) STR(predicate) : VL = 512-bit	
	ST1B (contiguous)	Addresses are at least 8-byte aligned and all of Element[8n] − Element[8n+7] are either active or inactive. Figure 7-4 shows an example.
	ST1H (contiguous)	Addresses are at least 8-byte aligned and all of Element[4n] − Element[4n+3] are either active or inactive.
	ST1W (contiguous)	Addresses are at least 8-byte aligned and both Element[2n] and Element[2n+1] are either active or inactive.
	ST1[BHW] (scatter)	When Element[n] is inactive, the ST0 flow corresponding to this element can be executed simultaneously.
	ST1D (scatter)	When the address in Element[n] is at least 8-byte aligned or its element is inactive, the ST0 flow corresponding to this element can be executed simultaneously.

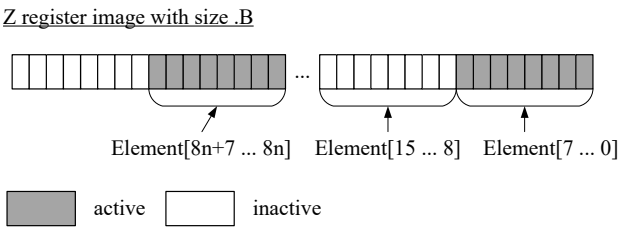


Figure 7-4 Example of active/inactive in ST1B (Contiguous)

7.7. Cache Line Cross

Even with a guarantee of the same address alignment as when at least memsize is used, the memory access range of some load instructions spans across two cache lines. Such memory access is called cache line cross (line cross).

If a line cross occurs, no penalty occurs if there are cache hits in both cache lines. In such cases, the LD flow is completed in one flow. On the other hand, even if a cache miss occurs in only one lines, the entire LD flow is handled as a cache miss. And if cache misses occur in both lines, processing for its two cache misses can be initiated simultaneously.

A line cross occurs only with load instructions. For store instructions, write addresses are aligned in the WB.

7.8. Execution of Noncontiguous Load/Store

7.8.1. Multiple Structures Instruction

This section describes the behavior of SIMD&FP LD[234] (multiple structures)/ST[234] (multiple structures) instructions and SVE LD[234][BHWD]/ST[234][BHWD] instructions. According to the notation in the specifications of instructions, LD1/ST1 (multiple structures) also belongs to the group of multiple-structures instructions. However, due to differences in behavior, it is not covered in this section.

Decode Stage Split and Execution Stage Split

Each multiple-structures instruction has multiple destination registers. The instruction is decoded into multiple μOP instructions at the decode stage. Basically, it is split into as many μOP instructions as the number of destination registers. Depending on the addressing mode, an auxiliary μOP instruction for address generation may be further added. Since this auxiliary μOP instruction only calculates addresses and does not perform memory access, it is not submitted to the load/store unit.

Only μOP instructions for memory access are submitted to the load/store unit. Although the number of instructions is basically the same as that of destination registers, but only LD[234][BH]/ST[234][BH] instructions are concerned, an instruction is further split into four at the execution stage. Table 7-6 shows the number of flows required for each instruction to be submitted to the load/store unit. This number of flows is essentially equal to the number of allocated FP/SP entries. "List of Instruction Attribute and Latency" shows the number of splits of each instruction.

Table 7-6 Required Number of Flows for μOP Instructions Split from Architecture Instruction to Send to Load/Store

Architecture Instruction	Required Number of Flows
LD2 (multiple structures) LD2[WD] ST2 (multiple structures) ST2[WD]	2
LD3 (multiple structures) LD3[WD] ST3 (multiple structures) ST3[WD]	3
LD4 (multiple structures) LD4[WD] ST4 (multiple structures) ST4[WD]	4
LD2[BH] / ST2[BH]	8
LD3[BH] / ST3[BH]	12
LD4[BH] / ST4[BH]	16

Load/Store Stage Split

For LD[234][WD]/ST[234][WD] instructions, memory access flows submitted to the load/store unit are further split depending on the access address pattern. Memory access by LD[234][WD]/ST[234][WD] instructions is performed in units of destination registers. That is, the memory access space per flow is larger than the vector data length of the register. Moreover, that access may be a wider space than the read width of the L1D cache. The read width of the L1D cache is 128 bytes and 128-byte aligned. Flows across a 128-byte boundary are split and performed in the pipeline sequentially. In this case, executing each flow

must wait to finish the preceding flow and there are at least 5 cycles penalty from finishing the preceding flow to submitting next flow into the pipeline.

Figure 7-5 is a diagram showing examples of flow splitting for an LD3D (multiple structures) instruction.

```
ld3d {z3.d, z4.d, z5.d}, p3/z, [x10, #0 mul vl]
```

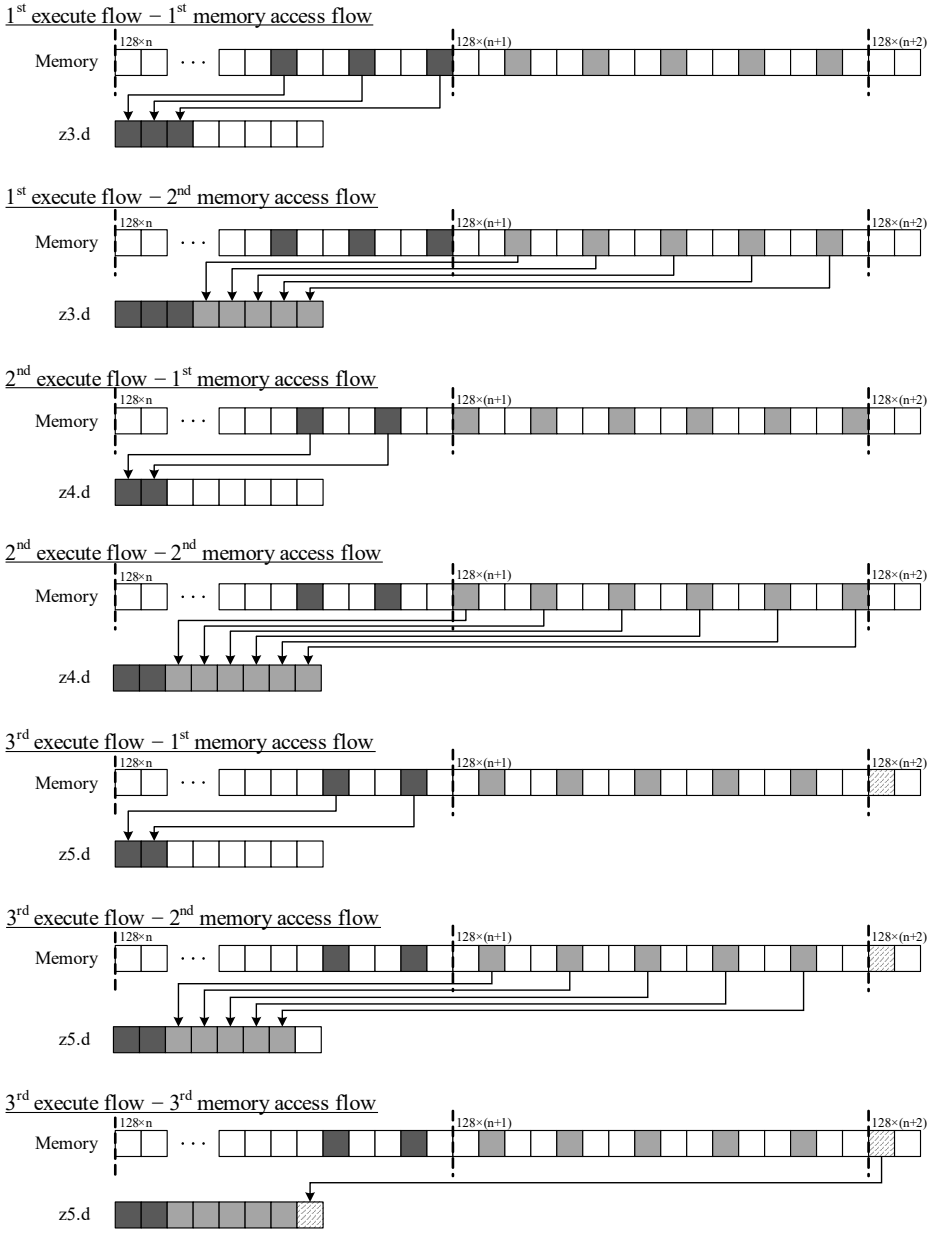


Figure 7-5 Illustration of Splitting LD3D (multiple structures) Instruction Flow

7.8.2. Gather Load/Scatter Store

Each SVE gather load instruction (called a gather instruction, below) and scatter store instruction (called a scatter instruction, below) performs memory access to multiple discrete addresses. Since one of the source operands for effective addresses is in a vector register, effective addresses are calculated in a floating-point operation pipeline. Therefore, the hardware behavior differs from the behavior for integer load/store instructions and SVE contiguous load/store instructions.

Decode

Unlike normal load/store instructions, gather/scatter instructions have multiple FP/SP entries allocated to one µOP instruction. This is because each element of processing by the load/store unit is independent as memory access by each element is independent. Note that SP entries are allocated to gather instructions although they are load instructions. Due to such particular FP/SP allocation, one restriction is that the decoder can decode only one gather or scatter instruction in the same cycle. This is a stronger restriction than in sequential decode in that other instructions cannot also be decoded. Table 7-7 shows the correspondence between the number of µOPs and the number of allocated FP/SP entries of each instruction that belongs to a group of gather/scatter instructions.

Table 7-7 Number of µOP Instructions and Number of Allocated FP/SP Entries for Each Gather/Scatter Instruction

	Number of µOP Instructions	FP	SP
LD1[BHW] (Gather) - S	1	8	1
LD1[BHWD] (Gather) - D	1	4	1
ST1[BHW] (Scatter) - S	8	16	16
ST1[BHWD] (Scatter) - D	4	8	8

For a gather instruction, one architecture instruction is decoded into one µOP instruction, but multiple FP/SP entries are allocated as shown in Table 7-7. Gather µOP instructions are dispatched to RSE0 connected to the FLA pipeline, which calculates effective addresses. Figure 7-6 is a diagram of the requests of a gather instruction.

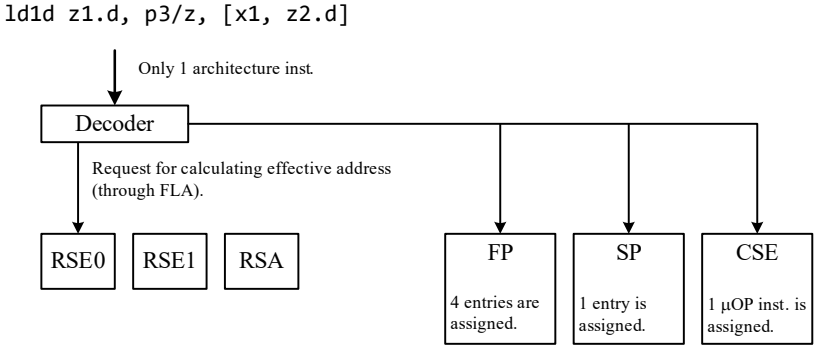


Figure 7-6 Requests of Gather Instruction

For a scatter instruction, one architecture instruction is decoded into four or eight µOP instructions depending on the number of elements. As many FP/SP entries as the number of elements are allocated. Unlike gather instructions, requests for calculating multiple effective addresses for scatter instructions are dispatched to RSE0. In addition, requests for store data transfer from a vector register to the SP are made to RSE0. Figure 7-7 is a diagram of the requests of a scatter instruction

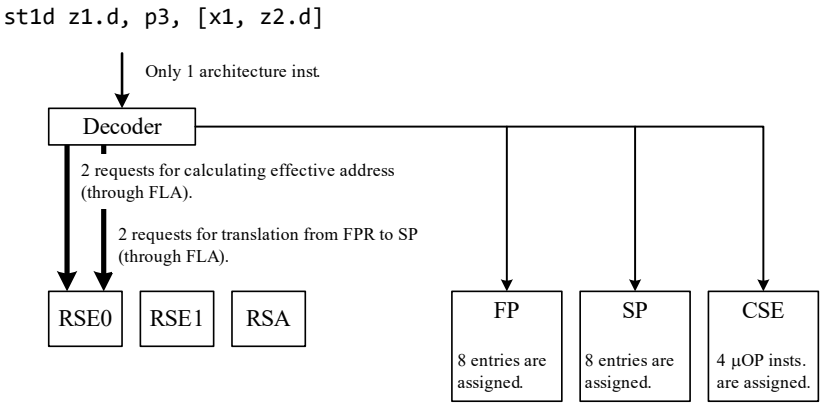


Figure 7-7 Requests of Scatter Instruction

When addressing is "vector plus immediate", an immediate value is passed directly to FLA from decoder. Therefore, requests for base operand transfer from the general-purpose register are omitted.

Effective Address Calculation

Figure 7-8 shows an outline of effective address calculation for a gather/scatter instruction. Unlike normal load/store instructions, a vector functional unit performs the calculation for effective address generation. However, only the FLA pipeline can calculate effective addresses. When addressing is "scalar plus vector," base operands must be transferred from the general-purpose register to the FLA functional unit. They are transferred from the general-purpose register via the EXA pipeline. The effective addresses calculated by the FLA functional unit are divided into and temporarily stored on the FP and SP. At this time, the issuing of other instructions to the EAGA/EAGB pipeline is limited due to a conflict at the write port of the FP.

For gather instructions, the effective addresses are divided into and stored on the FP and SP. For scatter instructions, the effective address of each element is individually stored in an FP/SP entry. In addition, scatter instructions require a transfer of store data from the vector register. This request is also processed using the FLA pipeline.

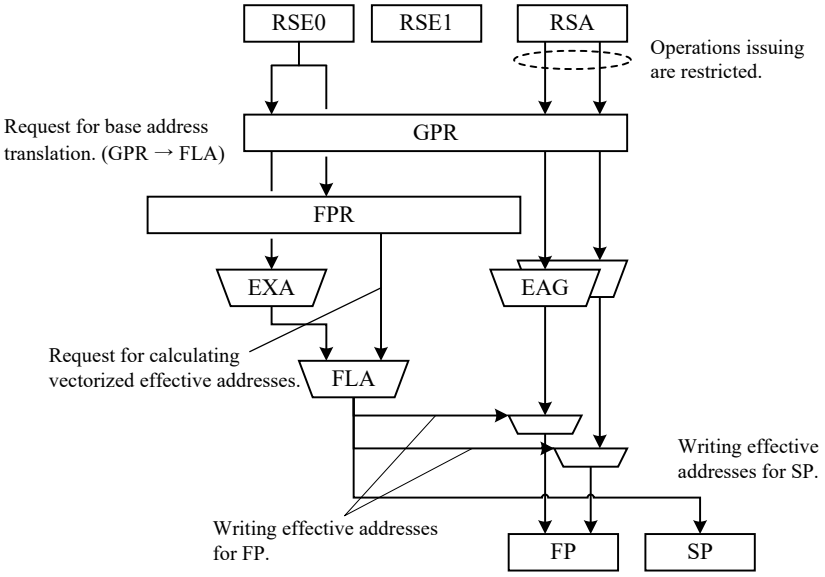


Figure 7-8 Effective Address Generation for Gather Instruction

Memory Access

Each element of a gather/scatter instruction points to an individual memory address. Therefore, each element is basically handled as individual memory access. Particularly, since scatter instructions have a restriction on ordering, each element is split and processed as a completely individual store flow. That is, if a scatter instruction has 8 elements, they are processed as 8 individual store flows; so 16 elements are 16 individual store flows. Individual flows and operations are the same as those in the basic processing of a normal store instruction.

On the other hand, a gather instruction is split in two stages from the perspective of reducing the number of flows. First, vector elements are divided into pairs of two elements from the head of its elements. If the address space for access by a pair does not fit into a 128-byte space within the same 128-byte boundary, the paired two elements are further divided. Conversely, if the address space fits into the same space, they are not split. Figure 7-9 shows the transition of memory access by a gather instruction. If a flow is split, as with the multiple structures instructions, executing the following flow must wait to finish the preceding flow and there are at least 5 cycles penalty with executing the following flow.

```
ld1d z2.d, p3/z, [x0, z1.d]
```

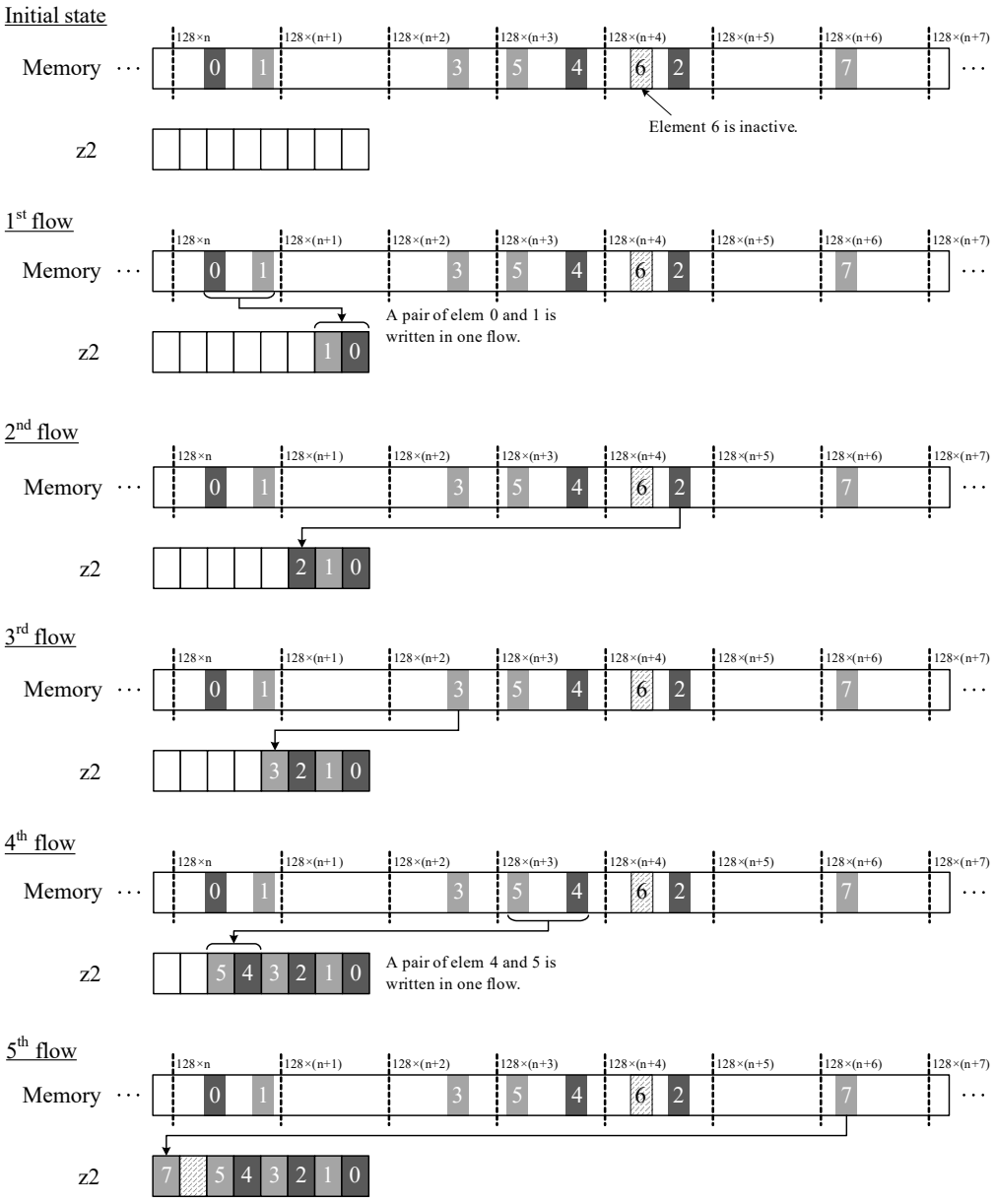


Figure 7-9 Summary of Elements for Gather Instruction

If both the elements in each of these two pairs are inactive, the memory access flow itself is deleted. The A64FX refers to the function for executing a pair of two elements of gather instructions as a combined gather function.

8. Memory Management Unit

8.1. Translation Lookaside Buffer

Table 8-1 shows the translation lookaside buffer (TLB) configuration of the A64FX. The TLB consists of two parts: instruction TLB and data TLB. Each of them has into two levels. L1-TLB has a full associative structure and adopts the FIFO method as a replacement algorithm. L2-TLB has a 4-way set associative structure and adopts the LRU method as a replacement algorithm.

The TLB of the A64FX supports the contiguous bit. The pages that are set with the contiguous bit store translation information in one entry.

Table 8-1 TLB Specifications

		For Instruction	For Data
L1	Association method	Full associative	Full associative
	Number of entries	16 entries	16 entries
	Replacement algorithm	FIFO	FIFO
L2	Association method	4-way set associative	4-way set associative
	Number of entries	1,024 entries	1,024 entries
	Replacement algorithm	LRU	LRU

8.2. Translation Table Cache

The translation table has a multi-level tree structure and the table walker requires multiple memory access times to obtain block/page descriptors. For the purpose of reducing these memory access times, a translation table cache is implemented for temporarily storing table descriptors. The translation table cache is like the TLB in that its purpose is to reduce processing time. However, there is one difference between them. The purpose of the TLB is to suppress the occurrence of a table walk itself, whereas the purpose of the translation table cache is to reduce the latency caused by memory access during a table walk.

As shown in Table 8-2, the translation table cache is a buffer that has a full associative structure, the number of entries is 16, and each entry holds a table descriptor. Table descriptors stored in the table cache are only for Stage-1 of a two-stage translation, and those for Stage-2 are not stored in the table.

Table 8-2 Table Cache Specifications

Translation table cache	Association method	Full associative
	Number of entries	16 entries
	Replacement algorithm	LRU

9. Cache Architecture

The A64FX has on-chip, two-level caches. The L1 caches are implemented in units of processor cores. There are two types of L1 caches: one for instructions, and the other for data. The L2 caches are implemented in units of CMGs. The L2 caches are shared by instructions and data. Coherence between caches is guaranteed by hardware.

9.1. Overview

As shown in Figure 9-1, the L2 caches and memory levels compose four CMGs. The ccNUMA (cache coherent NUMA) architecture is adopted between the CMGs. A memory unit is connected to only the L2 cache in a CMG. The physical address space is divided into the CMGs. Read/Write requests from the L2 cache are sent to the memory unit via the MAC (Memory Access Controller). Located between the L2 cache and MAC, a buffer called the move in buffer (MIB) manages in-flight requests to the MAC.

As shown in Figure 9-1, the CMGs are connected in a ring structure at the L2 cache level. Coherence between the L2 caches is guaranteed by hardware. The L2 caches are interconnected by two-way ring bus..

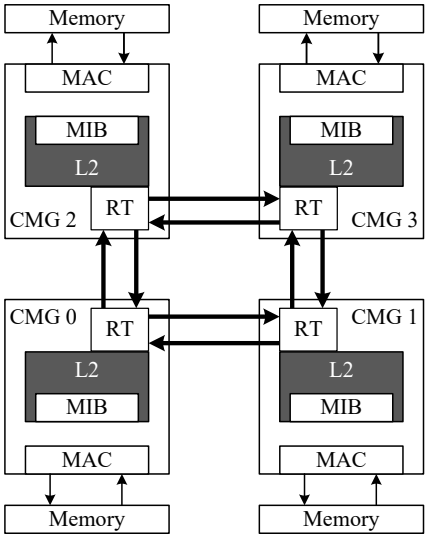


Figure 9-1 L2 Caches and Memory Levels

As shown in Figure 9-2, the L1I and L1D caches implemented in each processor core are connected to the L2 cache in a CMG on a one-to-one basis. The L1I and L1D caches in a CMG share the L2 cache in the same CMG, and the L2 cache contains the data in the L1I and L1D caches. The L1D cache and the L2 cache are connected by a two-tiered bus. In addition to a MIB, a MOB (Move Out Buffer) is located between the L1D cache and the L2 cache. L1D caches are in a structure that asynchronously manages move-in and move-out requests, and they have separate queues implemented. Table 9-1 summarizes bus throughput. Note that bus implemented units depend on the destination for itself.

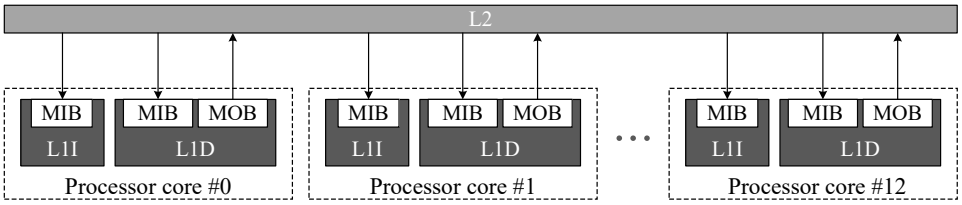


Figure 9-2 Connection Between L1 and L2 Caches

Table 9-1 Bus Throughput

	Direction	Bus Throughput
L1D	L2 to L1D	64 bytes / cycle (per Core)
	L1D to L2	32 bytes / cycle (per Core)
L2	L2 to L1D	512 bytes / cycle (per CMG)
	L1D to L2	256 bytes / cycle (per CMG)
L2	L2 to L2	64 bytes / cycle (per Ring)
L2	Memory to L2	128 bytes / cycle (per CMG)
	L2 to Memory	64 bytes / cycle (per CMG)

9.2. Cache Specifications

9.2.1. L1 Cache

Table 9-2 shows the specifications of the L1 cache. The access latency of the L1D cache varies in a range from 5 to 11 cycles, depending on the type of instruction. The L1D cache accepts two loads or one store at a time.

Table 9-2 L1 Cache Specifications

		For Instruction	For Data
L1 cache	Association method	4-way set associative	4-way set associative
	Capacity	64 KiB	64 KiB
	Hit latency (load-to-use)	4 cycles	5 cycles(integer)
			8 cycles (SIMD&FP / SVE in short mode)
			11 cycles (SIMD&FP / SVE in long mode)
	Line size	256 bytes	256 bytes
	Write method	---	Writeback
	Index tag	Virtual index and physical tag (VIPT)	Virtual index and physical tag (VIPT)
	Index formula	index_A = (A mod 16,384) / 256	index_A = (A mod 16,384) / 256
	Protocol	SI state	MESI state

Depending on the selection of the page size, a synonym may arise with the L1 cache. Since the L1 cache has a capacity of 64 KiB and is 4-way set associative, its indexes use a 16 KiB space. If the selected page is a 4-KiB page, a synonym may occur at bits[13:12] of the address. The A64FX is designed to prevent synonyms by using hardware.

9.2.2. L2 Cache

Table 9-3 shows the specifications of the L2 cache. The access latency of the L2 cache varies depending on the positional relationship between the processor core and the cache. The L2 cache hashes the indexes to mitigate index conflicts between processes. The cache has a two-bank configuration, and physical addresses are interleaved at bit[8].

Table 9-3 L2 Cache Specifications

		For instruction and data (by shared)
L2 cache (shared by instruction & data)	Association method	16-way set associative
	Capacity	8 MiB
	Hit latency (load-to-use)	46 to 56 cycles
	Line size	256 bytes
	Write method	Writeback
	Index and tag	Physical index and physical tag (PIPT)
	Index formula	index <10:0> = ((PA<36:34> xor PA<32:30> xor PA<31:29> xor PA<27:25> xor PA<23:21>) << 8) xor PA<18:8>
	Protocol	MESI state

9.3. Cache Coherence Protocol

In the A64FX, coherence between caches is guaranteed by hardware. A common MESI protocol is adopted as the protocol for coherence. Table 9-4 shows each state of the MESI protocol and major possible causes of the state.

Table 9-4 Details of MESI Protocol

Condition		State	Possible Cause of State
M	Modified	Data has been modified from main memory values (Dirty). Other caches at the same level do not have the data.	Data filling due to a store demand request. Stored in a cache line in the E/S state.
E	Exclusive	Data matches main memory values (Clean). Other caches at the same level do not have the data.	Data filling due to a load demand request while other caches do not have the data. Data filling due to prefetch access with a predefined type attribute while other caches do not have the data.
S	Shared	Data matches main memory values (Clean). Other caches at the same level also have the data.	Load demand request in the E state, or data fill request due to prefetch access with the Read attribute.
I	Invalid	A cache line is invalid.	Other caches request data when the data in the E/M state. Data writeback.

9.4. Move-In/Move-Out

Data fill and writeback to a cache level are performed on caches by operations called "move-in" and "move-out." This section summarizes and defines move-in and move-out operations.

Move-In

This operation writes data and tags, updates the state, and confirms the coherence of the data and the consistency of memory ordering at the cache level.

Move-Out

This operation reads data, disables the tag state, and confirms the coherence of the data and the consistency of memory ordering at the cache level.

Data fill and writeback are performed by combining the move-in and move-out operations. Basic cache miss processing at a cache level is as follows.

1. Receive a request from a higher level, and check whether the own cache level has data for the relevant address.
2. Upon confirming a cache miss, register the received request in the MIB, and communicate at the lower cache and memory level.
3. If the own cache level has no vacant lines for data fill, move out oldest data.
4. When there is a response with data from the lower cache and memory level, move in the data to the own cache level.
5. Read the moved-in data, and respond with the data to the higher level that is the source of the request.

Both the L1 and L2 caches can handle the processing of multiple cache misses in flight. The resources used to manage move-in and move-out in flight are the move-in buffer (MIB) and move-out buffer (MOB), respectively. Table 9-5 shows the quantity of resources at each cache level.

Table 9-5 Quantity of Queue Resources at Each Cache Level

	Queue Type	Number of Entries
L1I cache	MIB	3 entries / core
	MOB	---
L1D cache	MIB	12 entries / core
	MOB	4 entries / core
L2 cache	MIB	256 entries / CMG
	Store Lock Register	244 entries / CMG

The L2 cache has no MOB because move-out data is sent directly to the MAC. However, the L2 cache is equipped with the store lock register, which indicates when a move-out operation is in progress, to guarantee the consistency of write and read. The number of entries in this register determines an upper limit on the number of in-flights for writeback.

9.5. Move-In Bypass

As described in the previous section, a response with the data for which a cache miss occurred can be made to a higher-level cache after move-in is completed. However, waiting for the data response until the completion of move-in causes extra time taken for memory access. To reduce the time, the A64FX is equipped with the move-in bypass function, which makes a response to a higher cache level without waiting for the completion of move-in. Note that the L1D cache doesn’t have this function.

9.6. Zero Fill

The ARMv8 instruction set defines a DC ZVA instruction for the cache maintenance. This instruction writes zero data to the block that contains the virtual address specified by the instruction. In the A64FX, the DC ZVA instruction writes zero data to the L2 cache level. The block size indicated by DCZID_EL0 in the system register is the same as the cache line size. As shown in Figure 9-3, when the processor core executes the DC ZVA instruction, a zero-fill request is sent to the L2 cache via the load/store unit. When receiving the DC ZVA request, the L2 cache secures the cache line corresponding to the specified address and writes zero data.

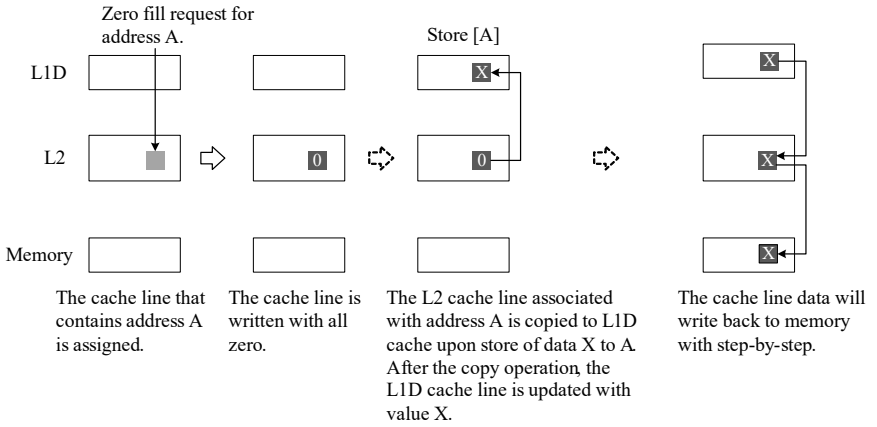


Figure 9-3 Basic Zero Fill Process

If the L2 cache does not have the data of the specified address, a cache miss does not occur, and no data fill operation from memory is performed. When the L1D cache has the data of the specified address, zero data is written after data in the L1D cache is written back to the L2 cache, as shown in Figure 9-4.

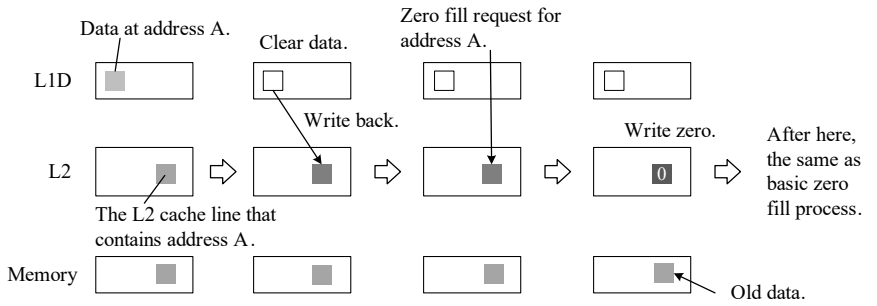


Figure 9-4 Zero Fill Process When L1D Cache Contains Data

As shown above, unlike a common store instruction, the DC ZVA instruction omits the data fill operation from memory to the L2 cache. This reduces memory bandwidth consumption at the time of writing to the memory space and can improve the effective memory bandwidth.

10. Memory Access Controller

10.1. Overview

The memory access controller (MAC) is a unit that writes to and reads from the main memory. The second-generation high bandwidth memory (HBM2) is adopted as the main memory. The MAC is implemented in each CMG. Each MAC has a P2P connection with the HBM2 chip.

Table 10-1 shows the specifications of the HBM2 supported by the MAC of the A64FX.

Table 10-1 Specifications of HBM2 Supported by A64FX

	Specification
Memory standard	HBM Gen2
Memory capacity	8 GiB (8Gib x8 stacks) / 1MAC
Data rate	2 Gbps

To achieve the maximum throughput while complying with HBM2 standards, the MAC has a scheduler for access order control. Table 10-2 shows the quantity of scheduler resources.

Table 10-2 Quantity of Scheduler Resources for HBM2

	Quantity of Resources
Scheduler queue size	244 entries / MAC

10.2. Performance

Table 10-3 shows the basic memory access performance of the A64FX. Note that the performance values represent performance per CMG.

Table 10-3 A64FX Memory Access Performance

		Memory Access Performance
Local memory latency (load-to-use)	Shortest core	135.5 ns (@ CPU 2GHz)
	Longest core	144.5 ns (@ CPU 2GHz)
Read throughput	Peak	256 GB/s (per MAC) (@ CPU 2GHz)
Write throughput	Peak	128 GB/s (per MAC) (@ CPU 2GHz)

11. Data Prefetch

Prefetch refers to an operation that reads in advance the data predicted to be used in a short time into a cache to improve performance. The two types of prefetch are software prefetch and hardware prefetch. In software prefetch, prefetch access is explicitly performed with a dedicated instruction. In hardware prefetch, hardware automatically predicts addresses and reads data pointed in its addresses. The A64FX supports both prefetch methods and is equipped with a hardware prefetch assist mechanism for finer control of prefetch operations.

11.1. Overview

This section describes hardware behavior for prefetch. The definitions of the following terms are given so that the descriptions are easy to understand.

Software prefetch

Explicit prefetch instruction based on an architecture instruction

Hardware prefetch

Prefetch instruction based on the address prediction mechanism of hardware

Demand access

Concept of memory access with data transferred between a register and a memory space, such as for a load/access instruction

Prefetch access

Concept of memory access generated for a memory space by a prefetch instruction and hardware prefetch. The information contained in prefetch access indicates the cache level at which to perform data fill, the access type, and address reliableness.

Demand flow

Operation-flow that involves data exchange between register, cache, or memory levels. For example, the LD flow of a load instruction is a demand flow distinct to a load/store pipeline because the LD flow writes data to a register. Likewise, an operation-flow of the L2 cache pipeline that must respond with data to the L1 cache is a demand flow.

Prefetch flow

Operation-flow that does not require data exchange between register, cache, or memory levels. This contrasts with a demand flow. For example, the operation-flow of a prefetch instruction does not need to write data to a register. From the perspective of the load/store pipeline, it is a prefetch flow. Likewise, from the perspective of the L2 cache pipeline, an operation-flow that does not need to respond to the L1 cache is a prefetch flow.

Figure 11-1 shows demand access, prefetch access, and the relationship of operation-flows that they generate.

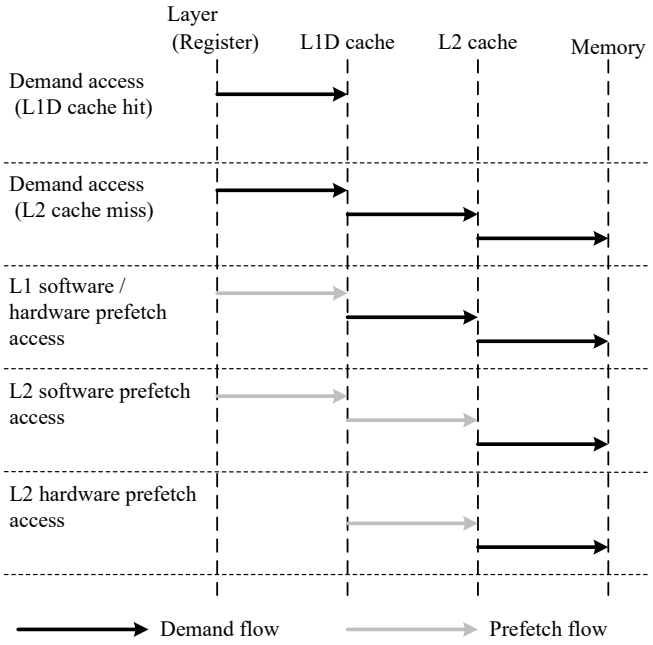


Figure 11-1 Operation-Flows for Demand Access and Prefetch Access

Every access by a load/store instruction is demand access, and all operation-flows for processing that access are demand flows. This is because the load/store instruction must eventually read data from or write data to a register, and the L2 cache and MAC must respond with data.

On the other hand, operation-flows for prefetch access processing vary in type at each cache level. For example, suppose that software prefetch causes prefetch access to the L1D cache. From the perspective of the load/store pipeline, that access is a prefetch flow. However, from the perspective of the L2 cache pipeline, it is a demand flow. That is because of the need to respond with data to the L1D cache.

In addition, software prefetch and hardware prefetch are different in that they require different operation-flows in the load/store pipeline. Software prefetch is an instruction, so its operation-flow goes through the execution and load/store pipelines, like the LD flow of a load instruction. This means that the flow consumes pipeline resources. On the other hand, prefetch access by hardware prefetch requires no operation-flows for the execution and load/store pipelines. This is because it submits operation-flows directly to the L2 cache pipeline. However, it requires the MI and MO flows for data fill and writeback, respectively.

In prefetch, there is a concept called "distance" between prefetch access and demand one.. The purpose of prefetch is to hide the latency of demand access. However, to achieve this purpose, it is necessary to prefetch data at the same address as targeted by demand access, at a preceding in the time axis direction. For example, to hide a demand access latency of N cycles, it is necessary to prefetch data more than N cycles before demand access generation. This time difference is the distance of prefetch. Particularly if memory access is continuous access in address, distance can be replaced with address space direction. For example, suppose that demand access is at A, and prefetch data is at address A+P. Then, if the demand access to A+P results in a cache hit when the access reaches A+P, latency N of the demand access can be said to be successfully hidden. In such cases, address difference P is the distance. This guide treats distance in the address space direction as prefetch distance.

11.2. Prefetch Access Type

One piece of information added to prefetch access is type. Type is information that indicates whether the prefetched data is for load or store. There are two types, Read and Write. The hardware uses type information to determine the cache state at the data fill time.

11.3. Prefetch Access Reliableness

One piece of information added to prefetch access is reliableness. Reliableness is an indicator for determining priority in processing generated prefetch requests. There are two types of reliableness, Strong and Weak. In the A64FX, reliableness can be set individually for each prefetch access by either using

tagged addresses and the system register for software prefetch or using the system register for hardware prefetch. The set reliableness is also applied to prefetch flows.

Prefetch Flow with Strong Attribute

Hardware tries to complete prefetch access as correctly as possible. For example, when resources are insufficient to convey a generated prefetch flow to the memory level, hardware waits until resources are available. In this situation, the flow is not deleted, and the prefetch access is executed to the end even if the following load/store instruction may be affected. However, if a TLB miss or page fault occurs in software prefetch, the prefetch request is deleted at that point in time. In this case with software prefetch, the prefetch instruction that is the source of the prefetch access is handled as a NOP instruction. In the case of hardware prefetch, prefetch access is stopped and the PFQ is cleared at the point in time when the TLB miss occurred. Since prefetch access with the Strong attribute may place a heavy burden on load/store pipeline, we recommend using the attribute only when prefetched data will absolutely be used. For details on the behavior of prefetch instructions, see the *A64FX Specification*.

Prefetch Flow with Weak Attribute

If plenty of resources are available, hardware correctly completes prefetch access. If not, the generated prefetch request is deleted. Basically, the processing of a demand flow or prefetch flow with the Strong attribute takes priority.

11.4. Software Prefetch

Software prefetch explicitly performs prefetch access based on a prefetch instruction. The operand part of a prefetch instruction can control the prefetch address required for prefetch access, the cache level that is the data fill destination, and the cache state. The A64FX has the HPC tagged address override function, which is a proprietary extension for HPC that can control hardware behavior with tagged addresses.

11.4.1. Prefetch Instructions

Prefetch instructions can be roughly divided into three types: ARMv8 prefetch instructions, SVE contiguous instructions, and SVE gather prefetch instructions. The following provides the definition and characteristics of each type:

ARMv8 prefetch instruction

Instruction to perform prefetch access to the prefetch address specified in an operand. Hardware performs data fill from memory in units of cache lines containing the specified address. Note that the phenomenon of cache line cross does not occur because the data sizes of ARMv8 prefetch instructions are considered as the byte type.

SVE contiguous prefetch instruction

Instruction to perform prefetch access in a range from the address specified in an operand to the address obtained by adding an SVE vector data length. Hardware performs data fill from memory in units of cache lines containing addresses from the first to last addresses in the range. If the first and last addresses for prefetch belong to different cache lines, hardware fills both memory blocks.

SVE gather prefetch instruction

Instruction that supports the same addressing mode as discrete access instructions (gather/scatter) do. A single instruction can perform prefetch access to multiple addresses. The basic behavior for individual addresses is the same as with ARMv8 prefetch instructions.

Table 11-1 shows the classification and mnemonic correspondence of prefetch instructions.

Table 11-1 Classifications and Mnemonics of Prefetch Instructions

Classification	Mnemonic	Description
ARMv8 prefetch instruction	PRFM (immediate)	Prefetch instructions that support consecutive load/store (without consideration of line cross)
	PRFM (literal)	
	PRFM (register)	
	PRFM (unscaled offset)	
SVE contiguous prefetch instruction	PRF[BHWD] (scalar plus immediate)	Prefetch instructions that support consecutive load/store (with consideration of line cross)
	PRF[BHWD] (scalar plus scalar)	
SVE gather prefetch instruction	PRF[BHWD] (scalar plus vector)	Prefetch instructions that support discrete access instructions (gather/scatter)
	PRF[BHWD] (vector plus immediate)	

11.4.2. Prefetch Instruction Attribute

The first operand of a prefetch instruction specifies prefetch options. There are three options: Type, Target, and Policy. The cache level that is the data fill destination and the cache state can be controlled by combining Type and Target as shown in Table 11-2. No Policy setting affects hardware behavior.

Table 11-2 Correspondence Between Prefetch Instruction Options, Cache Levels, and States

		Target		
		L1	**L2**	**L3**
Type	**PLI**	NOP	NOP	NOP
	PLD	L1D / S or E	L2 / S or E	NOP
	PST	L1D / E	L2 / E	NOP

In addition, the reliableness of software prefetch can be controlled by using the pf_func[0] bits in the tagged address part of a prefetch instruction. Table 11-3 shows the correspondence between the bit field and reliableness. For details on bit fields, see the *A64FX Specification*.

Table 11-3 Correspondence Between pf_func[0] Bit and Software Prefetch Reliableness

pf_func[0]	Software Prefetch Reliableness
0	Strong
1	Weak

11.5. Hardware Prefetch

The A64FX has functions using hardware to predict the addresses likely to be accessed in a short time and to perform prefetch access. These functions are collectively called hardware prefetch. Hardware prefetch by the A64FX can predict addresses for continuous access streams. There are two hardware prefetch modes: stream detect mode and prefetch injection mode.

11.5.1. Prefetch Resource

The hardware has a resource called PFQ (Pre-Fetch Queue) for address prediction and prefetch access. The PFQ is located inside each processor core and has 16 entries per processor core. The PFQ stores a predicted address, prefetch distance, and address offset for prediction.

The predicted address is an address likely to be demand-accessed in the future. When the predicted address prepared in advance by hardware matches the actual demand-access address, the PFQ performs prefetch access to the address obtained by adding the prefetch distance to the predicted address. Then, the PFQ further adds the address offset to update the predicted address. This operation is repeated to continue prefetch access.

11.5.2. Behavior of Stream Detect Mode

This section describes the behavior of stream detect mode, which is one of the two hardware prefetch modes. In stream detect mode, prefetch access automatically detects a continuous access stream. Figure 11-2 is a diagram of hardware prefetch behavior in stream detect mode.

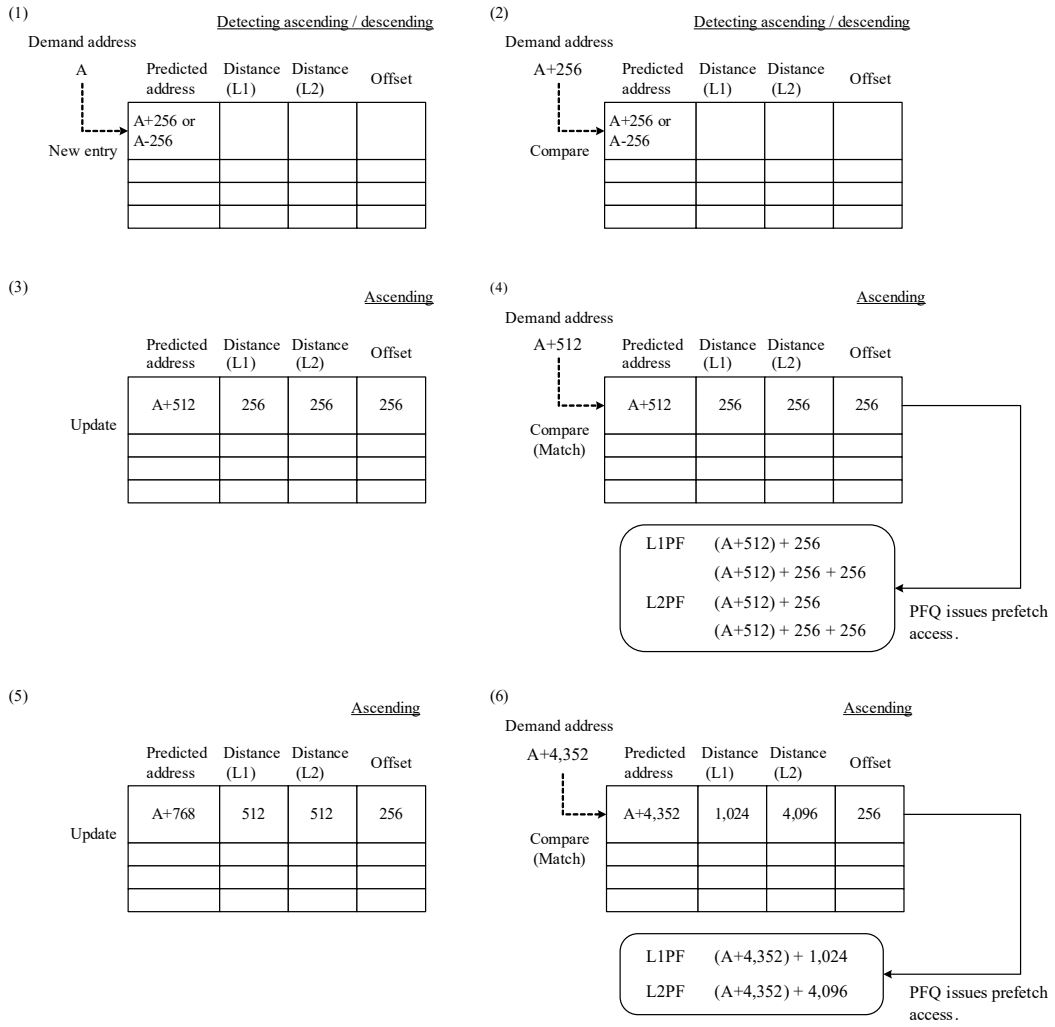


Figure 11-2 Hardware Prefetch Behavior in Stream Detect Mode

Stream Detection and Registration with PFQ

The PFQ monitors access for L1D cache misses in demand access. If a cache miss occurs, the PFQ generates a predicted address based on the demand-access address and registers it itself to try to determine whether the stream of the demand access is proceeding in ascending or descending order (Figure 11-2 (1)). At the next demand access time, the PFQ compares the demand-access address to the predicted address that

it registered earlier itself to determine the direction of the stream (Figure 11-2 (2)). Upon determining the direction of the stream, the PFQ updates the predicted address, registers a new prefetch distance and address offset, and then starts the following demand access. The prefetch distance and address offset are registered as positive values for a stream in ascending order or as negative values for a stream in descending order. The absolute value for both the prefetch distance and offset registered first is 256 bytes (Figure 11-2 (3)).

Prefetch Access Generation

Upon detecting demand access matching the predicted address, the PFQ issues prefetch access (Figure 11-2 (4)). After giving an instruction to perform prefetch access, the PFQ adds the offset to the predicted address to update the predicted address (Figure 11-2 (5)). For a certain period from the start of the following demand access, the PFQ gives an instruction to perform L1 and L2 prefetch access of two cache lines. In addition, the PFQ extends the prefetch distance by adding 256 bytes to the prefetch distance every time that prefetch access is performed.

Steady Prefetch Access Generation

After the same PFQ repeatedly gives an instruction to perform prefetch access, it eventually reaches the maximum value of the prefetch distance from L1 to L2 in a phased manner. When the prefetch distance reaches the maximum value, the PFQ changes its prefetch access instruction to access one cache line. At the same time, it stops extending the prefetch distance (Figure 11-2 (6)). Afterward, this condition continues if demand access matching the predicted address continues.

In stream detect mode on the A64FX, demand and prefetch access addresses are rounded off in units of cache lines. That is, the lower 7 bits of the access addresses are ignored. This ensures that, even when stream access is not completely continuous, prefetch access can be issued if the stream access is continuous in terms of cache lines.

11.5.3. Behavior of Prefetch Injection Mode

The other hardware prefetch mode is prefetch injection mode. In prefetch injection mode, the characteristics of access by load/store instructions are set with a register for prefetch control, and they are used for hardware prefetch. The prefetch injection mode is further divided into PFQ_ALLOCATE and PFQ_NOALLOCATE modes.

PFQ_ALLOCATE Mode

The behavior in this mode is basically the same as in stream detect mode. Demand access is monitored for L1D cache misses. If a cache miss occurs, a predicted address, prefetch distance, and offset are registered with the PFQ. However, this mode differs from stream detect mode in that the predicted address, prefetch distance, and offset are calculated from set values in the system register. In PFQ_ALLOCATE mode, hardware prefetch follows the two steps below.

1. Stream detection

   If demand access of the target stream results in an L1D cache miss, the value obtained by adding an offset to the demand-access address is used as the predicted address. Set values in the system register are used as the prefetch distance and offset.
2. Prefetch access generation

   When the predicted address matches the demand-access address, a prefetch access instruction is given for the address obtained by adding the prefetch distance to the matching address. Further, the offset is added to the predicted address to update it. However, the prefetch distance remains at the initial value and is not extended.

PFQ_NOALLOCATE Mode

In this mode, the PFQ does not monitor access for L1D cache misses, but it gives an instruction to perform prefetch access at the point in time when demand access of the target stream occurs. The prefetch access is performed at the address obtained by adding the prefetch distance to the demand-access address. Since prefetch access is generated unconditionally in this mode, prefetch access occurs even when the demand-access address results in a cache hit. On the other hand, the PFQ does not perform monitoring, which has the benefit of no consumption of the PFQ.

11.5.4. Hardware Prefetch Assist Mechanism

To improve the convenience of hardware prefetch, the A64FX has the hardware prefetch assist mechanism as an interface for controlling the behavior of hardware prefetch.

In Stream Detect Mode

In stream detect mode, prefetch behavior can be controlled in the following ways by using tagged addresses and the system register:

- Disabling prefetch

  A tagged address can be used to specify whether to set each instruction as a PFQ monitoring target. No prefetch access is generated from instructions that are not subject to PFQ monitoring.
- Specifying a cache level attribute for prefetch access

  An instruction can be given to perform prefetch access with a cache level attribute specified using a tagged address. The cache level attribute can be selected from three choices: L1D cache, L2 cache, and both.
- Specifying a reliableness attribute for prefetch access

  A reliableness attribute can be specified for a prefetch request generated by the PFQ via the system register. The reliableness attribute can be selected between two choices: Strong and Weak.
- Specifying a maximum prefetch distance for the PFQ

  A maximum value can be specified for the prefetch distance of the PFQ via the system register.

In Prefetch Injection Mode

In prefetch injection mode, prefetch behavior can be controlled in the following ways in addition to the functions in stream detect mode:

- Assigning stream numbers

  A tagged address can be used to assign a stream number to each instruction. Basically, functions in stream detect mode can be specified in units of streams.
- Specifying a prefetch distance for each stream

  The system register can be used to specify a prefetch distance for each stream number.
- Specifying an offset for each stream

  The system register can be used to specify an offset for each stream number.

11.5.5. Consideration of Cache Hierarchy

One piece of information added to prefetch access is the cache level attribute. The cache level attribute indicates the cache level of the prefetch destination. Higher-speed programs require prefetch access with an appropriate cache level attribute. Unnecessary prefetch access leads to wasteful consumption of hardware resources.

In hardware prefetch, hardware automatically determines the optimal cache level attribute. Basically, the PFQ gives instructions to perform both prefetch access with the L1D cache attribute and prefetch access with the L2 cache attribute. If the prefetch access with the L2 cache attribute repeatedly results in a cache hit in the L2 cache, the PFQ stops the prefetch access with the L2 cache attribute. However, if the prefetch access with the L1D cache attribute results in an L2 cache miss, the PFQ resumes the prefetch access with the L2 cache attribute.

11.6. Usage Example of Prefetch Injection Mode

One of the features of prefetch injection mode is that it enables offsets to be specified by software. This function enables prefetch access generation for stride access that exceeds the cache line size. Figure 11-3 shows a sample program. Table 11-4 shows a control register configuration example. The *A64FX Specification* describes the tagged addresses and control register specifications.

Sample code

```
int  i;
double  y[N], x[N*64];
assert (N > 0);
for (i = 0; i < N; i ++) {
      y[i] = x[i*64];          /* accesses the array x with stride of 512 bytes width. */
}
```

Sample assembly code

```
      mov       x0,#N
      adr       x1,y                    // sets the address of array y.
      adr       x2,x                    // sets the address of array x.
      orr       x2,x2,#(8<<60)          // merges base address and tagged
                                        address which assigns the stream to
                                        control#0 register.
loop:
      ldr       d0,[x2]
      str       d0,[x1]
      add       x2,x2,#512
      add       x1,x1,#8
      subs      x0,x0,#1
      b.gt      x0,loop
```

Figure 11-3 Usage Example of Prefetch Injection Mode

Table 11-4 Control Register Configuration Example

System Register	Bit Field	Set Value	Description
IMP_PF_INJECTION_CTRL0_EL0	V	1	Enables the control register.
	L1W	0	Sets the L1 prefetch attribute to Strong.
	L2W	0	Sets the L2 prefetch attribute to Strong.
	A	1	Sets PFQ_ALLOCATE mode.
	T	0	Sets the prefetch attribute to PLD.
	SWW	0	This is an instruction for software prefetch. It does not matter whether the value is 0 or 1 in this example.
	PFQ_OFFSET	512	Sets the same value as the stride width.
IMP_PF_INJECTION_DISTANCE0_EL0	L1PF_DISTANCE	1,024	Sets the L1 prefetch distance.
	L2PF_DISTANCE	10,240	Sets the L2 prefetch distance.

The reliableness of prefetch access, PFQ_ALLOCATE/PFQ_NOALLOCATE mode, and prefetch distances must be determined according to program characteristics.

12. Sector Cache

12.1. Overview

The sector cache is a mechanism that partitions the area of a cache and can select which partition to use in units of instructions or processor cores. The purpose of this mechanism is to provide software with a method of controlling the use of the cache with finer granularity. Each partitioned area in the A64FX is called "sector." A cache can be partitioned into sectors with any capacity in units of cache ways via the system register. This mechanism is implemented in the L1D and L2 caches. Shown in Figure 12-1, the L1D cache has four sectors that can be specified in units of instructions. The L2 cache also has four sectors, but it has a hierarchical structure consisting of two sectors that can be specified in units of processor cores and two sectors that can be specified in units of instructions. The four sectors of the L1D cache are mapped to two sectors of the same sector group in the L2 cache. Within each CMG, the area of a sector is closed. Tagged addresses are used to specify sectors in units of instructions. The system register is used in units of processor cores. For details on tagged addresses and the system register, see the *A64FX Specification*.

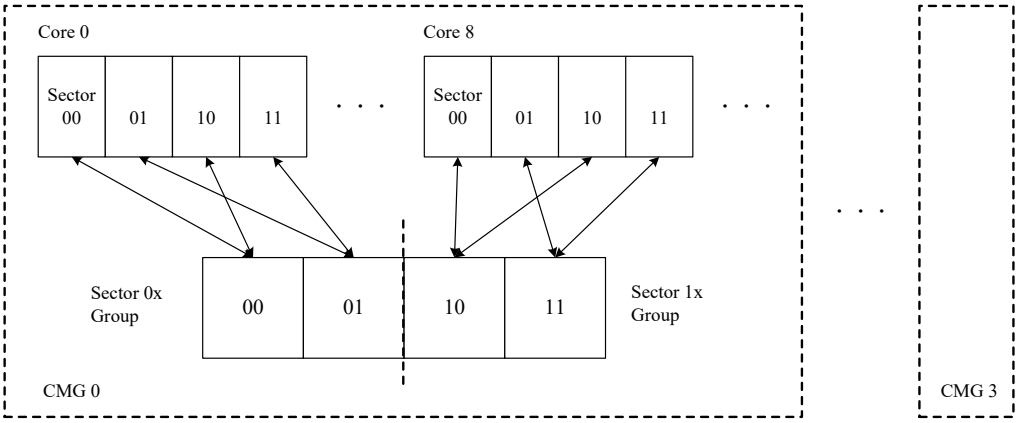
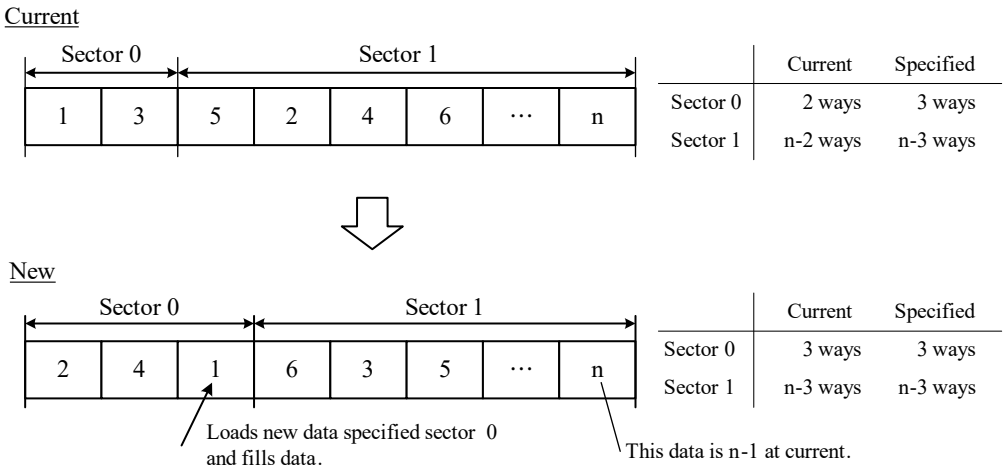


Figure 12-1 L1D/L2 Sector Cache

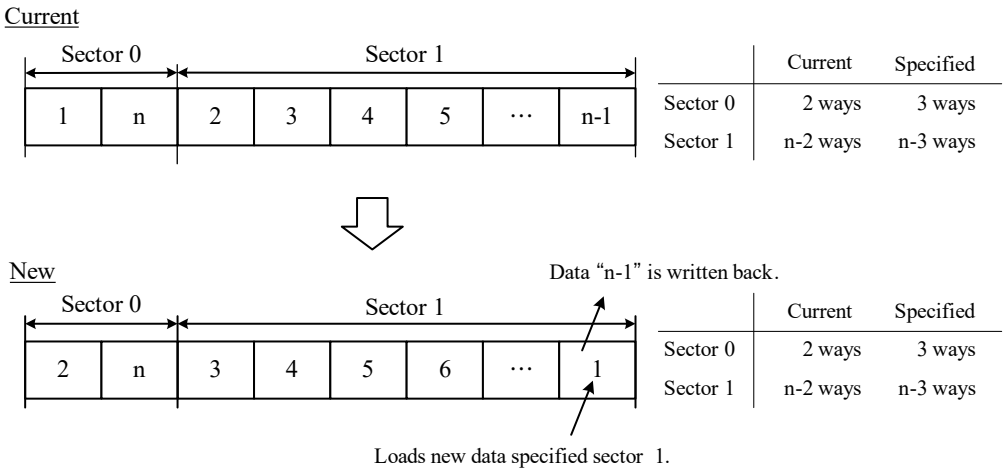
12.2. Sector Cache Behavior

Each sector capacity of a cache is specified by the maximum number of ways allocated to the sector. The sector capacity can be dynamically changed during program execution. When the sector capacity changes, hardware adjusts the capacity at data fill on individual cache lines in order to gradually bring the sector capacity closer to the specified capacity. Figure 12-2 and Figure 12-3 show examples. Figure 12-2 shows an example where sector 0 usage is lower than the required sector capacity. Data in sector 1 is written back to adjust the capacity at the execution time of a load instruction with sector 0 specified. In contrast, Figure 12-3 shows an example where sector 1 usage is higher and a load instruction with sector 1 specified is executed. In this case, data in sector 1 is written back so that sector 0 usage does not decrease.



(*) Numbers show LRU

Figure 12-2 Example of Sector Cache Capacity Adjustment (1)



(*) Numbers show LRU

Figure 12-3 Example of Sector Cache Capacity Adjustment (2)

13. Hardware Barrier

The hardware barrier is a mechanism that supports synchronization between software processes or threads through hardware. Shown in Figure 13-1, each CMG has dedicated system registers, and synchronization is processed through their registers. Since the system register is implemented in the L2 cache, the response time for register access in synchronization processing is nearly equal to that for an L2 cache hit. In addition, since the system register itself works as a barrier variable so that the atomicity of register operations is guaranteed by hardware, mutual exclusion for variable operations is not necessary. The aim of these features is simplification of programs and higher-speed synchronization processing.

Synchronization processing across CMGs is not supported since hardware barrier resources are implemented per CMG. Figure 13-2 shows sample code for synchronization processing. For details on hardware barrier specifications, see the *A64FX Specification*.

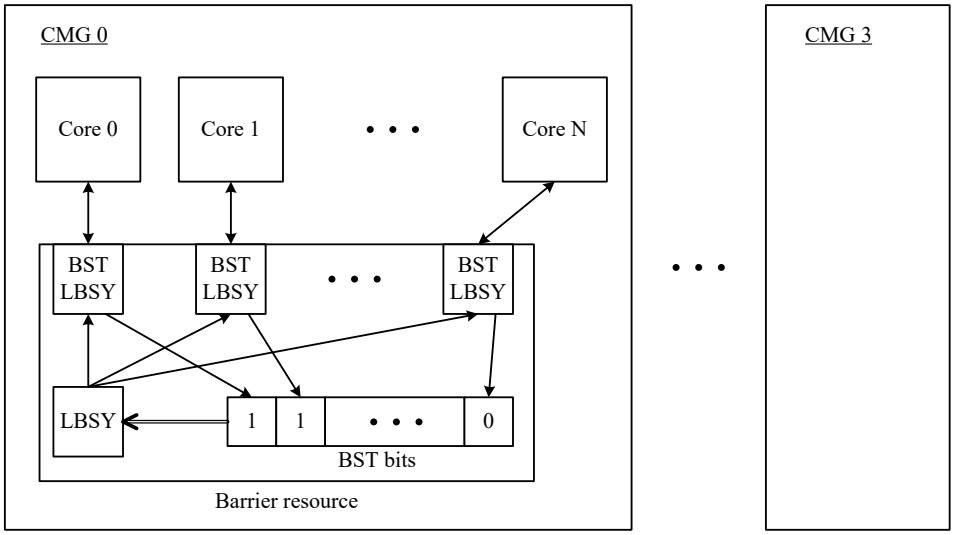


Figure 13-1 Hardware Barrier Resources

```
#define     BARRIER_LBSY_SYNC_W1_EL0       S3_3_C15_C15_1
#define     BARRIER_BST_SYNC_W1_EL0        S3_3_C15_C15_1

mrs_s       x2, BARRIER_LBSY_SYNC_W1_EL0         // Load LBSY to x2
eor         x1, x2, #0x1
msr_s       BARRIER_BST_SYNC_W1_EL0, x1          // Write ~LBSY to BST
sevl                                             // EVENT register clear
wfe
loop:
   wfe                                           // Sleep
   mrs_s    x2, BARRIER_LBSY_SYNC_W1_EL0
   cmp      x2, x1                               // Compare x2 (= LBSY) and x1 (= ~LBSY)
   b.ne     loop                                 // Reload LBSY to x2 if LBSY != ~LBSY
```

Figure 13-2 Sample Code for Synchronization Processing

14. Performance Monitor Events

The processor is equipped with the performance monitoring unit (PMU) to monitor dynamic program behavior. In addition to the events defined in the *ARMv8 Manual* and the *SVE Manual*, the A64FX has events unique to it. These events are designed to enable calculation of indicators that assist software performance analysis by not only using each event with its direct meaning but also combining multiple events. This chapter describes how to create these indicators. For details on the specifications of each event, see the *A64FX PMU Events*.

14.1. Instruction Mix

Table 14-1 summarizes events for calculating a dynamic instruction frequency distribution at the program execution time. All the events for Instruction Mix are designed to count architecture instruction commits. A group of instructions counted with these events has an inclusion relationship. The relationship means that the frequency of a group of instructions for which events are not defined can be calculated by combining those events. The inclusion relationship of events is represented by the levels of indentations in the event name column in Table 14-1. The rows that display "Other" do not refer to actual events but to items calculated by combining events. The formulas are shown in Table 14-2. Note that the SVE_MATH_SPEC event is not included in the INST_SPEC event.

Table 14-1 Performance Events for Instruction Mix

Event for Instruction Mix	Target of Counting by Event
INST_SPEC	All instructions
FP_SPEC	All instructions categorized floating-point operation
FP_FMA_SPEC	All FMA operation instructions
FP_RECPE_SPEC	Reciprocal approximation instructions and reciprocal square root approximation instructions
Other (Basic FP operations)	General floating-point operation instructions
FP_CVT_SPEC	Floating-point precision conversion instructions (including conversion between general-purpose register values)
FP_MV_SPEC	Transfer instructions using floating-point register (including general-purpose register)
ASE_SVE_INT_SPEC	Integer operation instruction using floating-point register
PRD_SPEC	Integer operation instructions using predicate register
LD_SPEC	All load instructions
BASE_LD_REG_SPEC	Load instructions to general-purpose register
ASE_SVE_LD_SPEC	Load instructions to floating-point register
FP_LD_SPEC	Scalar load instructions to floating-point register
Other (All vector load)	Vector load instructions to floating-point register
SVE_LDR_REG_SPEC	All SVE LDR instructions
SVE_LDR_PREG_SPEC	SVE LDR (predicate) instructions
Other (LDR vector)	SVE LDR (vector) instructions

Event for Instruction Mix	Target of Counting by Event
BC_LD_SPEC	Replicate and broadcast load instructions to floating-point register (LD1R)
ASE_SVE_LD_MULTI_SPEC	Multiple structure load instructions to floating-point register (LD[234]*)
SVE_LD_GATHER_SPEC	SVE gather load instructions
SVE_LDFF_SPEC	SVE first-fault and non-fault load instructions
Other (Basic vector load)	General vector load instructions to floating-point register
ST_SPEC	All store instructions
BASE_ST_REG_SPEC	Store instructions from general-purpose register
ASE_SVE_ST_SPEC	Store instructions from floating-point register
FP_ST_SPEC	Scalar store instructions from floating-point register
Other (All vector store)	Vector store instructions from floating-point register
SVE_STR_REG_SPEC	All SVE STR instructions
SVE_STR_PREG_SPEC	SVE STR (predicate) instructions
Other (STR vector)	SVE STR (vector) instructions
ASE_SVE_ST_MULTI_SPEC	Multiple structure store instructions from floating-point register (ST[234]*)
SVE_ST_SCATTER_SPEC	SVE scatter store instructions
Other (Basic vector store)	General vector store instructions from floating-point register
PRF_SPEC	All prefetch instructions
SVE_PRF_GATHER_SPEC	SVE gather prefetch instructions
SVE_PRF_CONTIG_SPEC	SVE contiguous prefetch instructions
Other (Prefetch in Base Instructions)	Base instruction prefetch
DCZVA_SPEC	DC ZVA instructions
BR_PRED	All branch instructions
CRYPTO_SPEC	All encryption instructions
SVE_MOVPRFX_SPEC	SVE MOVPRFX instructions
Other (Base instruction excluding load/store)	Instructions that belong to Base Instructions excluding load/store instructions
Event Independent of Inclusion Relationship	
SVE_MATH_SPEC	SVE mathematical function auxiliary instructions

Table 14-2 Formulas for Other (Instruction Mix)

Item	Formula
Basic FP operations	FP_SPEC – (FP_FMA_SPEC – FP_RECPE_SPEC)
All vector loads	ASE_SVE_LD_SPEC – FP_LD_SPEC
LDR vector	SVE_LDR_REG_SPEC – SVE_LDR_PREG_SPEC
Basic vector loads	ASE_SVE_LD_SPEC – (FP_LD_SPEC + SVE_LDR_REG_SEPC + BC_LD_SPEC + ASE_SVE_LD_MULTI_SPEC + SVE_LD_GATHER_SPEC + SVE_LDFF_SPEC)
All vector stores	ASE_SVE_ST_SPEC – FP_ST_SPEC
STR vector	SVE_STR_REG_SPEC – SVE_STR_PREG_SPEC
Basic vector stores	ASE_SVE_ST_SPEC – (FP_ST_SPEC + SVE_STR_REG_SPEC + ASE_SVE_ST_MULTI_SPEC + SVE_ST_SCATTER_SPEC)
Prefetch in base instruction	PRF_SPEC – (SVE_PRF_GATHER_SPEC + SVE_PRF_CONTIG_SPEC)
Base insts. excluding load/store	INST_SPEC – (FP_SPEC + FP_CVT_SPEC + FP_MV_PSEC + ASE_SVE_INT_SPEC + PRD_SPEC + LD_SPEC + ST_SPEC + PRF_SPEC + DCZVA_SPEC + BR_PRED + CRYPTO_SPEC + SVE_MOVPRFX_SPEC)

14.2. FLOPS

Table 14-3 summarizes events for calculating floating operations per second (FLOPS). These events count the number of floating-point operations that are committed. Among these events, the ones related to SVE instructions only count the number of operations in units of 128 bits, so they are not affected by vector length at the program execution. Therefore, the correct number of operations must be calculated with consideration of vector length at the execution. In addition, operation of FMA instruction are counted as two operations per element.

Since the PMU is a resource in units of PEs, the number of operations measured by the events is in units of PEs. Therefore, calculations of the total number of operations at the parallel execution must consider the job model, etc. Since FLOPS is the number of operations per unit time, external parameters (processor operation frequency during program execution and program execution time) are separately required. To calculate a highly precise execution time, we recommend using the CPU_CYCLES event obtained at the same time.

Table 14-3 Performance Events for FLOPS

Performance Event	Description of Event
FP_SCALE_OPS_SPEC	Number of operations per 128 bits considering the number of elements in each SVE instruction
FP_FIXED_OPS_SPEC	Number of operations considering the number of elements in SIMD&FP instructions
FP_HP_SCALE_OPS_SPEC	Number of half-precision operations only, taken out of FP_SCALE_OPS_SPEC
FP_HP_FIXED_OPS_SPEC	Number of half-precision operations only, taken out of FP_FIXED_OPS_SPEC
FP_SP_SCALE_OPS_SPEC	Number of single-precision operations only, taken out of FP_SCALE_OPS_SPEC
FP_SP_FIXED_OPS_SPEC	Number of single-precision operations only, taken out of FP_FIXED_OPS_SPEC
FP_DP_SCALE_OPS_SPEC	Number of double-precision operations only, taken out of FP_SCALE_OPS_SPE
FP_DP_FIXED_OPS_SPEC	Number of double-precision operations only, taken out of FP_FIXED_OPS_SPEC

14.3. Hardware Resource Monitor

Table 14-4 summarizes events for monitoring the behavior of basic processor resources. These events count dynamic hardware behaviors, such as cache misses and branch misprediction , at the program execution.

Table 14-4 Performance Events for Hardware Resource Monitoring

Performance Event	Description of Event
BR_MIS_PRED	Number of times of pipeline flush due to a branch misprediction
L1I_CACHE_REFILL	Number of L1I cache misses
L1D_CACHE_REFILL	Number of L1D cache misses
L1D_CACHE_REFILL_DM	Number of L1D cache misses attributable to demand access
L1D_CACHE_REFILL_PRF	Number of L1D cache misses attributable to prefetch access
L1D_CACHE_REFILL_HWPRF	Number of L1D cache misses attributable to hardware prefetch access
L1D_CACHE_WB	Number of times of writeback from the L1D cache
L1_MISS_WAIT	Amalgamated value of the number of in-flights per cycle in L1D cache miss processing (i.e., value obtained by integrating the number of used MIBs of the L1D cache for each cycle)
L2D_CACHE_REFILL	Number of L2 cache misses
L2D_CACHE_REFILL_DM	Number of L2 cache misses attributable to the demand flow
L2D_CACHE_REFILL_PRF	Number of L2 cache misses attributable to the prefetch flow
L2D_CACHE_REFILL_HWPRF	Number of L2 cache misses attributable to hardware prefetch in the prefetch flow
L2D_CACHE_WB	Number of times of writeback from the L2 cache
L2_MISS_WAIT	Amalgamated value of the number of in-flights per cycle in L2 cache miss processing (i.e., value obtained by integrating the number of used MIBs of the L2 cache for each cycle)
L1I_TLB_REFILL	Number of L1-ITLB misses
L1D_TLB_REFILL	Number of L1-DTLB misses
L2I_TLB_REFILL	Number of L2-ITLB misses
L2D_TLB_REFILL	Number of L2-DTLB misses
EFFECTIVE_INST_SPEC	Number of committed architecture instructions excluding MOVPRFX instructions
BR_PRED	Number of committed branch instructions
CPU_CYCLES	Number of PE cycles

The events shown in Table 14-4 can be used to calculate indicators of hardware performance at the program execution. Table 14-5 summarizes the indicators.

Table 14-5 Method to Calculate Hardware Performance Indicators at Program Execution

Indicator	Formula
Cycles per instruction (CPI)	CPU_CYCLES / EFFECTIVE_INST_SPEC
Branch misprediction rate	BR_MIS_PRED / EFFECTIVE_INST_SPEC
L1I cache miss rate	L1I_CACHE_REFILL / EFFECTIVE_INST_SPEC
L1D cache miss rate	L1D_CACHE_REFILL / EFFECTIVE_INST_SPEC
L1D cache miss rate attributable to demand access	L1D_CACHE_REFILL_DM / EFFECTIVE_INST_SPEC
L1D cache miss rate attributable to prefetch access	L1D_CACHE_REFILL_PRF / EFFECTIVE_INST_SPEC
L1D cache miss rate attributable to prefetch access generated by hardware prefetch	L1D_CACHE_REFILL_HWPRF / EFFECTIVE_INST_SPEC
L1D cache miss rate attributable to software prefetch access	(L1D_CACHE_REFILL_PRF - L1D_CACHE_REFILL_HWPRF) / EFFECTIVE_INST_SPEC
L2 cache miss rate	L2D_CACHE_REFILL / EFFECTIVE_INST_SPEC
L2 cache miss rate attributable to demand flow	L2D_CACHE_REFILL_DM / EFFECTIVE_INST_SPEC
L2 cache miss rate attributable to prefetch flow	L2D_CACHE_REFILL_PRF / EFFECTIVE_INST_SPEC
L2 cache miss rate attributable to prefetch flow generated by hardware prefetch	L2D_CACHE_REFILL_HWPRF / EFFECTIVE_INST_SPEC
L2 cache miss rate attributable to prefetch flow generated by software prefetch	(L2D_CACHE_REFILL_PRF - L2D_CACHE_REFILL_HWPRF) / EFFECTIVE_INST_SPEC
Average latency of L1D cache miss processing	L1_MISS_WAIT / L1D_CACHE_REFILL
Average latency of L2 cache miss processing	L2_MISS_WAIT / L2D_CACHE_REFILL
Average number of outstanding misses in L1D cache miss processing	L1_MISS_WAIT / CPU_CYCLES
Average number of outstanding misses in L2 cache miss processing	L2_MISS_WAIT / CPU_CYCLES
L1-ITLB miss rate	L1I_TLB_REFILL / EFFECTIVE_INST_SPEC
L1-DTLB miss rate	L1D_TLB_REFILL / EFFECTIVE_INST_SPEC
L2-ITLB miss rate	L2I_TLB_REFILL / EFFECTIVE_INST_SPEC
L2-DTLB miss rate	L2D_TLB_REFILL / EFFECTIVE_INST_SPEC
Bidirectional effective bandwidth between L1D cache and L2 cache	(L1D_CACHE_REFILL + L1D_CACHE_WB) * 256 * processor frequency / CPU_CYCLES
Bidirectional effective bandwidth between L2 cache and memory	(L2D_CACHE_REFILL + L2D_CACHE_WB) * 256 * processor frequency / CPU_CYCLES

14.4. Cycle Accounting

One of the processor performance indicators is cycles per instruction (CPI), which represents the average CPU cycles spent by the processor to execute one instruction. Here, CPI can be considered as the accumulated processing time of various operation-flows for instruction execution; for example, time for operations and memory access. The expression of CPI as an accumulation of such individual processing times is called "Cycle Accounting." The A64FX has events for Cycle Accounting. Table 14-6 summarizes the events. Like events for Instruction Mix, there is an inclusion relationship between objects measured. "Other" indicates an item that can be calculated by combining events. The formulas are shown in Table 14-7.

Table 14-6 Performance Events for Cycle Accounting

Events for Cycle Accounting	Target of Counting by Event
CPU_CYCLES	CPU clock cycles
0INST_COMMIT	Cycles during which number of instruction commits is 0
LD_COMP_WAIT	Cycles during which oldest instruction in CSE cannot be committed due to wait for completion of memory access
LD_COMP_WAIT_EX	Cycles caused by instructions belonging to Base Instructions, taken out of LD_COMP_WAIT
LD_COMP_WAIT_L2_MISS	Cycles during L2 cache miss, taken out of LD_COMP_WAIT
LD_COMP_WAIT_L2_MISS_EX	Cycles caused by instructions belonging to Base Instructions, taken out of LD_COMP_WAIT_L2_MISS
Other (ld_comp_wait_l2_miss_fl)	Cycles caused by instructions belonging to SIMD&FP or SVE instruction, taken out of LD_COMP_WAIT_L2_MISS
LD_COMP_WAIT_L1_MISS	Cycles during L1D cache miss and L2 cache hit, taken out of LD_CIMP_WAIT (Strictly speaking, this includes the cycles until the L2 cache miss is determined at the L2 cache miss time.)
LD_COMP_WAIT_L1_MISS_EX	Cycles caused by instructions belonging to Base Instructions, taken out of LD_COMP_WAIT_L1_MISS
Other (ld_comp_wait_l1_miss_fl)	Cycles caused by instructions belonging to SIMD&FP or SVE instruction, taken out of LD_COMP_WAIT_L1_MISS
LD_COMP_WAIT_PFP_BUSY	Cycles during which memory access instructions cannot be committed due to insufficient resources for L2 cache prefetch processing, taken out of LD_COMP_WAIT (This represents an event where prefetch flow cannot be processed and the flow generator instruction cannot be committed.)
LD_COMP_WAIT_PFP_BUSY_EX	Cycles caused by instructions belonging to Base Instructions, taken out of LD_COMP_WAIT_PFP_BUSY
LD_COMP_WAIT_PFP_BUSY_SWPF	Cycles caused by software prefetch instruction, taken out of LD_COMP_WAIT_PFP_BUSY
Other (ld_comp_wait_pfp_busy_fl)	Cycles caused by SIMD&FP or SVE instruction, taken out of LD_COMP_WAIT_PFP_BUSY
Other (ld_comp_wait_l1_hit)	Cycles during L1D cache hit, taken out of LD_COMP_WAIT (Strictly speaking, this includes the cycles until the L1D cache miss is determined at the L1D cache miss time.)
Other (ld_comp_wait_l1_hit_ex)	Cycles caused by instructions belonging to Base Instructions, taken out of ld_comp_wait_l1_hit
Other (ld_comp_wait_l1_hit_fl)	Cycles caused by instructions that belong to SIMD&FP or SVE instructions, taken out of ld_comp_wait_l1_hit
EU_COMP_WAIT	Cycles during which oldest instruction in CSE cannot be committed due to wait for completion of operations
FL_COMP_WAIT	Cycles caused by instructions belonging to SIMD&FP or SVE instruction, taken out of EU_COMP_WAIT
Other (ex_comp_wait)	Cycles caused by instructions belonging to Base Instructions, taken out of EU_COMP_WAIT
BR_COMP_WAIT	Cycles during which oldest instruction in CSE cannot be committed because it is branch direction and waiting for branch direction determination

Events for Cycle Accounting	Target of Counting by Event
ROB_EMPTY	Cycles during which instructions cannot be committed because CSE is empty (The instruction does not exist after the decode stage.)
ROB_EMPTY_STQ_BUSY	Cycles during which instructions cannot be committed because CSE is empty and Virtual SP is full (Since Virtual SP is full, decoding is currently stopped.)
WFE_WFI_CYCLE	Cycles during which behavior of PE is stopped by WFE instruction or WFI (This appears as a synchronization wait time in a parallel program.)
Other (rob_empty_not_stq_busy)	Cycles during which instructions cannot be committed because CSE is empty due to other causes (This mainly appears as the instruction fetch time.)
UOP_ONLY_COMMIT	Cycles during which only μOP instructions are committed (For an architecture instruction that is decoded into 2 or more μOP instructions, commit of the last μOP instruction means commit of the architecture instruction. This means there is a condition where only μOP instructions are committed.)
SINGLE_MOVPRFX_COMMIT	Cycles during which only unpacked MOVPRFX instructions are committed
Other (0inst_commit_other)	Cycles during which instructions cannot be committed due to other causes
1INST_COMMIT	Cycles during which number of instruction commits is 1
2INST_COMMIT	Cycles during which number of instruction commits is 2
3INST_COMMIT	Cycles during which number of instruction commits is 3
4INST_COMMIT	Cycles during which number of instruction commits is 4
Event Independent of Inclusion Relation	
LD_COMP_WAIT_EX	Cycles caused by instructions belonging to Base Instructions, taken out of LD_COMP_WAIT

Table 14-7 Formulas for Other (Cycle Accounting)

Item	Formula
ld_comp_wait_l2_miss_fl	LD_COMP_WAIT_L2_MISS － LD_COMP_WAIT_L2_MISS_EX
ld_comp_wait_l1_miss_fl	LD_COMP_WAIT_L1_MISS － LD_COMP_WAIT_L1_MISS_EX
ld_comp_wait_pfp_busy_fl	LD_COMP_WAIT_PFP_BUSY – (LD_COMP_WAIT_PFP_BUSY_EX + LD_COMP_WAIT_PFP_BUSY_SWPF)
ld_comp_wait_l1_hit	LD_COMP_WAIT – (LD_COMP_WAIT_L2_MISS + LD_COMP_WAIT_L1_MISS + LD_COMP_WAIT_PFP_BUSY)
ld_comp_wait_l1_hit_ex	LD_COMP_WAIT_EX －(LD_COMP_WAIT_L2_MISS_EX + LD_COMP_WAIT_L1_MISS_EX + LD_COMP_WAIT_PFP_BUSY_EX)
ld_comp_wait_l1_hit_fl	ld_comp_wait_l1_hit - ld_comp_wait_l1_hit_ex
ex_comp_wait	EU_COMP_WAIT - FL_COMP_WAIT
rob_empty_not_stq_busy	ROB_EMPTY – (ROB_EMPTY_STQ_BUSY + WFE_WFI_CYCLE)
0inst_commit_other	0INST_COMMIT – (UOP_ONLY_COMMIT + SINGLE_MOVPRFX_COMMIT + LD_COMP_WAIT + EU_COMP_WAIT + BR_COMP_WAIT + ROB_EMPTY)

15. List of Resources

This chapter lists and summarizes A64FX hardware resources.

Table 15-1 Out-of-Order Resources

Resource	Quantity of Resource		
Commit stack entry (CSE)	128 entries		
Group ID (GID)	32 entries		
General-purpose physical register (GPR)	96 entries	Architecture register	32 entries
		Renaming register	64 entries
Floating-point physical register (FPR)	128 entries	Architecture register	32 entries
		Renaming register	96 entries
Predicate physical register (PPR)	48 entries	Architecture register	16 entries
		Renaming register	32 entries
Reservation station for EAG (RSA)	10 entries x 2 (split)		
Reservation station for EXE (RSE) (shared by Integer, SIMD&FP, SVE)	20 entries x 2 (split)		
Reservation station for branch (RSBR)	19 entries		
Temporary operand register (TOR)	3 entries		
Fetch port (FP)	Virtual	160 entries	
	Real	40 entries	
Store port (SP)	Virtual	192 entries	
	Real	24 entries	
Write buffer (WB)	8 entries		

Table 15-2 Resources for Branch Misprediction Mechanism

Resource	Quantity of Resource
Instruction Buffer (IBUFF)	6 entries
Small Taken Chain Predictor (S-TCP)	4 entries
Loop Prediction Table (LPT)	8 entries
Branch Weight Table (BWT)	2,048 entries
Branch Target Buffer (BTB)	2,048 entries (4-way set associative)
Return Address Stack (RAS)	8 entries

Table 15-3 Resources for Memory Management Unit

Resource	Quantity of Resource
L1-ITLB	16 entries (full associative)
L1-DTLB	16 entries (full associative)
L2-ITLB	1,024 entries (4-way set associative)
L2-DTLB	1,024 entries (4-way set associative)
Translation Table Cache	16 entries (full associative)

Table 15-4 Resources for L1/L2 Cache

Resource	Quantity of Resource
L1I cache	64 KiB (4-way set associative)
L1D cache	64 KiB (4-way set associative)
L2 unified cache	8 MiB (16-way set associative)
L1I MIB	3 entries/core
L1D MIB	12 entries/core
L1D MOB	4 entries/core
L2 MIB	256 entries/CMG
L2 Store lock register	244 entries/CMG

16. List of Instruction Attribute and Latency

This chapter provides lists of latencies of all instructions supported by A64FX processor. Table 16-1, Table 16-2 and Table 16-3 represent ARMv8, ARMv8 SIMD&FP, and SVE instructions, respectively.
Each column of the tables is described below.

- Instruction, Alias
  This column shows an instruction. Alias instructions are shown as a subset of the source instructions.
- Control Option
  This column shows conditions for different hardware behavior within the same instruction. Basically, conditions are expressed in assembler syntax.
  The register size of destination operands is used to distinguish a variant.
  If variant cannot be distinguished by destination operands, source operands are used.
- VL
  This column shows the vector length if the vector length has an impact a control option.
- Number of µOP
  This column shows the number of µOP instructions into which an instruction is split at the decode time. For detail on µOP instructions, see Section 4.1.
- Sequential Decode
  This column shows a mark if the instruction is subjected to sequential decode. For detail on sequential decode, see Section 4.1.
- Pre-Sync, Post-Sync
  These columns show a mark if the instruction is subjected to pre-sync or post-sync control. For detail on sync controls, see Section 4.7.
- Pack
  This column shows a mark if the instruction is possible to be packed when modified by MOVPRFX instruction. For detail on packing, see Section 4.3.
- Extra µOP
  This column shows a mark to the instruction that is subjected to be added a µOP instruction if the MOVPRFX modification with merging predication is performed. For detail on merging predication, see Section 6.5.1.
- Blocking
  This column shows a mark if the instruction is subjected to be blocking control at the execution stage.
  The letter "P" means pipeline blocking and the letter "E" means operation blocking. For detail on blocking control, see Section 6.3.
- Latency
  This column shows the execution latency of the instruction. Basically, the latency is expressed in units of µOP instructions.
  For load instructions, this list describes those latency only for L1D cache hit case.
  Since operation-flow splitting in the load/store stage depends on instructions' access properties such as address alignment, data length, etc., this list describes only the first flow required for the processing.
  Notation rules are described below:
  - A forward slash ("/") is the separator between µOP instructions.
  - If grouping instructions that have dependencies, the upper left superscript indicates the relative position of the source µOP instruction.
    For example, "1/2/3/[1,2]4" means that input to the last µOP instruction is output from the second and third µOP instructions.
  - The "( )" format indicates a grouping. Moreover, the "( ) x N" format indicates expansion of a group, and it means that the group enclosed in the round brackets is expanded N times.
    For example, "(1 / 2) x 3" is equal to "1 / 2 / 1 / 2 / 1 / 2", and "1 / (2 / [1]4) x 2" is equal to "1 / 2 / [1]4 / 2 / [1]4".
  - If the instructions have dependency, the position notation is also grouped.
    For example, "1 / [1]2 / [2]2 / [3]2" means that second, third and last µOP instructions are dependent respectively on the first.
    At this time, the superscript characters are also grouped : "1 / [1/2/3](2) x 3".

  In addition, some µOP instructions may be split into multiple operation-flows. For details on multiple operations, see Section 4.2. The notation when multiple operation-flows are combined are shown below:
  - If each operation-flow into which an instruction is split at the decode stage has no dependency and can be executed independently in multiple pipelines: Write these flows delimited by the comma (",") in a row.
  - If each operation-flow into which an instruction is split at the decode stage has dependency but are executed in multiple pipelines: Write these flows by connecting them with semicolon (";").
  - If each operation-flow into which an instruction is split at the execution stage has dependency and are executed sequentially in one pipeline: Write these flows by connecting them with plus sign ("+").
  - "Pipe()" is notation for gather load / scatter store and multiple structures load / store.
    Pipe(L, N) indicates that the flow of the latency L is issued N times continuously in each cycle. Each operation-flow is executed in pipeline because there is no dependency between each other.

  Since the dependencies between operation-flows are only before and after, the position of the dependency is not shown.
- Pipeline
  This column shows the pipeline that executes operation-flow. For details on execution pipelines, see Section 6.2.
  The notation is based on that of latency and include the additional rules shown below:
  - Wildcard ("*") and logical add ("|") indicate cases where multiple pipelines can execute an operation-flow.
    For example, "EX* | EAG*" indicates that either EX (integer operation) or EAG (address calculation) pipeline can execute the flow.
    Furthermore, "(EXA+EXA) | (EXB+EXB)" means that there is dependency between the first and second operation-flow, and that both are executed in either EXA or EXB pipeline.
  - A bypass penalty is statically applied to some combinations of operation-flow.
    In such cases, insert "+NULL+" between the pipelines where the bypass occurs, and indicating the latency of the bypass penalty at the same position.
    For example, the latency notation of "1+3+6" and the pipeline notation of "EXA+NULL+FLA" indicate that the bypass penalty is 3 cycles at the center.

- "Pipe()", like that of latency, is a particular notation for gather load, scatter store, multiple structures load / store.
  Pipe(P, N) indicates that the flow is issued N times for pipeline P.
  However, in the case of gather load or scatter store, since one operation-flow uses both EAGA and EAGB pipelines, the pipeline used is denoted by "EAGA & EAGB".

- Number of FP
  This column shows the number of fetch ports to allocate to the load/store instruction. For details on fetch port, see Section 7.3.
- Number of SP
  This column shows the number of store ports to allocate to the load/store instruction. For details on store port, see Section 7.3.
- FLOPS
  This column shows the number of floating-point operations per element that can be counted by using their performance events of instructions.
  If this field is blank, it is treated as 0 FLOPS.
  For details on FLOPS calculation with performance events, see Section 14.2.

16.1. ARMv8 Base Instructions

Table 16-1 Instruction Attributes/Latency (ARMv8)

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
ADC			1					1	EX*		
ADCS			1					1	EX*		

ADD (extended register)		<amount> = 0 && (
		If sf = 0 Then
		  <extend> = {LSL|UXTW|UXTX|SXTW|SXTX}
		Else
		  <extend> = {UXTX|SXTX}
		)
			1					1	EX* | EAG*		

			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
ADD (immediate)	MOV (to/from SP)		1					1	EX* | EAG*		
			1					1	EX* | EAG*		
ADD (shifted register)		<amount> = 0	1					1	EX* | EAG*		
		<amount> = [1-4] && <shift>=LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ADDS (extended register)	CMN (extended register)	<amount> = 0	1					1	EX*		
			1				P	1+1	(EXA + EXA) | (EXB + EXB)		

		<amount>=0 && (
		If sf = 0 Then
		  <extend>= {LSL|UXTW|UXTX|SXTW|SXTX}
		Else
		  <extend> = {UXTX|SXTX}
		)
			1					1	EX*		

			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
ADDS (immediate)	CMN (immediate)		1					1	EX*		
			1					1	EX*		
ADDS (shifted register)	CMN (shifted register)	<amount> = 0	1					1	EX*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
		<amount> = 0	1					1	EX*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ADR			1					1	EAGB		
ADRP			1					1	EAGB		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
AND (immediate)			1					1	EX* | EAG*		
AND (shifted register)		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ANDS (immediate)	TST (immediate)		1					1	EX*		
			1					1	EX*		
ANDS (shifted register)	TST (shifted register)	<amount> = 0	1					1	EX*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
		<amount> = 0	1					1	EX*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ASRV	ASR (register)		1					2	EX*		
B.cond			1					NA	BR		
B			1					NA	BR		
BFM	BFC		4	✓				2 / [1]1 / 1 / [1,2]1	EX* / EX* / EX* / EX*		
	BFI		4	✓				2 / [1]1 / 1 / [1,2]1	EX* / EX* / EX* / EX*		
	BFXIL		4	✓				2 / [1]1 / 1 / [1,2]1	EX* / EX* / EX* / EX*		
			4	✓				2 / [1]1 / 1 / [1,2]1	EX* / EX* / EX* / EX*		
BIC (shifted register)		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
BICS (shifted register)		<amount> = 0	1					1	EX*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
BL			1					1	EAGB, BR		
BLR			1					1, NA, NA	EAGB, EXA, BR		
BR			1					1, NA	EXA, BR		
BRK			2	✓	✓	✓		NA / NA	/		
CAS{|A|AL|L}			3	✓				1 / 5;1 / [2]1	EAG* / EAGA; EXA / EXA	1	1
CAS{|A|AL|L}B			3	✓				1 / 5;1 / [2]1	EAG* / EAGA; EXA / EXA	1	1
CAS{|A|AL|L}H			3	✓				1 / 5;1 / [2]1	EAG* / EAGA; EXA / EXA	1	1
CASP{|A|AL|L}			7	✓				1 / 5;[1]1 / 1 / [2]1 / 5;[4]1 / 1 / [2]1	EAG* / EAGA; EXA / EXA / EAG* / EAGA; EXA / EXA / EAG*	2	2
CBNZ			1					1	EX*		
CBZ			1					1	EX*		
CCMN (immediate)			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
CCMN (register)			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
CCMP (immediate)			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
CCMP (register)			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
CLREX			2	✓				NA / NA	/ EAGA	1	
CLS			1					2	EX*		
CLZ			1					2	EX*		
CRC32B			1				E	10	EXB		
CRC32H			1				E	10	EXB		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
CRC32W			1				E	12	EXB		
CRC32X			1				E	20	EXB		
CRC32CB			1				E	10	EXB		
CRC32CH			1				E	10	EXB		
CRC32CW			1				E	12	EXB		
CRC32CX			1				E	20	EXB		
CSEL			1					1	EX*		
CSINC	CINC		1					1	EX*		
	CSET		1					1	EX*		
			1					1	EX*		
CSINV	CINV		1					1	EX*		
	CSETM		1					1	EX*		
			1					1	EX*		
CSNEG	CNEG		1					1	EX*		
			1					1	EX*		
DCPS1			2	✓	✓	✓		NA / NA	/		
DCPS2			2	✓	✓	✓		NA / NA	/		
DCPS3			2	✓	✓	✓		NA / NA	/		
DMB			2	✓				NA / NA	/ EAGA	1	
DRPS			2	✓	✓	✓		NA / NA	/		
DSB			2	✓				NA / NA	/ EAGA	1	
EON (shifted register)		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
EOR (immediate)			1					1	EX* | EAG*		
EOR (shifted register)		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ERET			2	✓	✓	✓		NA / NA	/		
EXTR	ROR (immediate)		1					2	EX*		
			3	✓				2 / 2 / [1,2]1	EX* / EX* / EX*		
HINT	NOP		1					NA			
	YIELD		6	✓	✓	✓		NA / NA / NA / NA / NA / NA	/ / / / /		
	WFE		2	✓	✓	✓		NA / NA	/		
	WFI		2	✓	✓	✓		NA / NA	/		
	SEV		2	✓	✓	✓		NA / NA	/		
	SEVL		2	✓	✓	✓		NA / NA	/		
	ESB		2	✓		✓		NA / NA	/ EAGA	1	
HLT			2	✓	✓	✓		NA / NA	/		
HVC			2	✓	✓	✓		NA / NA	/		
ISB			2	✓		✓		NA / NA	/ EAGA	1	

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
LDADD	STADD		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDA			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDAB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDAH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDAL			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDALB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDALH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDB	STADDB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDH	STADDH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDL	STADDL		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDLB	STADDLB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDADDLH	STADDLH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDAR			1					5	EAGA	1	
LDARB			1					5	EAGA	1	
LDARH			1					5	EAGA	1	
LDAXP			3	✓				1 / [1]5 / [2]5	EAG* / EAGA / EAGA	3	
LDAXR			1					5	EAGA	1	
LDAXRB			1					5	EAGA	1	
LDAXRH			1					5	EAGA	1	
LDCLR	STCLR		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRA			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRAB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRAH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRAL			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRALB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRALH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRB	STCLRB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRH	STCLRH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRL	STCLRL		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRLB	STCLRLB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDCLRLH	STCLRLH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEOR	STEOR		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORA			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORAB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORAH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORAL			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORALB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORALH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORB	STEORB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORH	STEORH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORL	STEORL		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORLB	STEORLB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDEORLH	STEORLH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDLAR			1					5	EAGA	1	
LDLARB			1					5	EAGA	1	
LDLARH			1					5	EAGA	1	
LDNP			2					5 / 5	EAG* / EAG*	2	
LDP		Post-index	3					5 / 5 / 1	EAG* / EAG* / EX*| EAG*	2	
		Pre-index	3					5 / 5 /1	EAG* / EAG* / EX*| EAG*	2	
		Signed offset	2					5 / 5	EAG*/EAG*	2	
LDPSW		Post-index	3					5 / 5 / 1	EAG* / EAG* / EX*| EAG*	2	
		Pre-index	3					5 / 5 / 1	EAG* / EAG* / EX*| EAG*	2	
		Signed offset	2					5 / 5	EAG*/EAG*	2	
LDR (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDR (literal)			1					5	EAGB	1	
LDR (register)			1					5	EAG*	1	

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
LDRB (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDRB (register)			1					5	EAG*	1	
LDRH (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDRH (register)			1					5	EAG*	1	
LDRSB (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDRSB (register)			1					5	EAG*	1	
LDRSH (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDRSH (register)			1					5	EAG*	1	
LDRSW (immediate)		Post-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Pre-index	2					5 / 1	EAG* / EX*| EAG*	1	
		Unsigned offset	1					5	EAG*	1	
LDRSW (literal)			1					5	EAGB	1	
LDRSW (register)			1					5	EAG*	1	
LDSET	STSET		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETA			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETAB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETAH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETAL			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETALB			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETALH			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETB	STSETB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETH	STSETH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETL	STSETL		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETLB	STSETLB		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSETLH	STSETLH		4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
			4	✓				1 / [1]5 / [1]1 / [1]NA	EAG* / EAGA / EXA / EXA	1	1
LDSMAX	STSMAX		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXA			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXAB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXAH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXAL			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXALB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXALH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXB	STSMAXB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXH	STSMAXH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXL	STSMAXL		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXLB	STSMAXLB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMAXLH	STSMAXLH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMIN	STSMIN		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINA			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINAB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINAH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINAL			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINALB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINALH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINB	STSMINB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINH	STSMINH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINL	STSMINL		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINLB	STSMINLB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDSMINLH	STSMINLH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDTR			1					5	EAG*	1	

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
LDTRB			1					5	EAG*	1	
LDTRH			1					5	EAG*	1	
LDTRSB			1					5	EAG*	1	
LDTRSH			1					5	EAG*	1	
LDTRSW			1					5	EAG*	1	
LDUMAX	STUMAX		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXA			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXAB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXAH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXAL			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXALB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXALH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXB	STUMAXB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXH	STUMAXH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXL	STUMAXL		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXLB	STUMAXLB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMAXLH	STUMAXLH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMIN	STUMIN		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINA			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINAB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINAH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINAL			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINALB			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINALH			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINB	STUMINB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINH	STUMINH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINL	STUMINL		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINLB	STUMINLB		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUMINLH	STUMINLH		4	✓				1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
			4	✓			/ / P /	1 / [1]5 / 1+[1]1 / [1]NA	EAG* / EAGA / EXA+EXA / EXA	1	1
LDUR			1					5	EAG*	1	
LDURB			1					5	EAG*	1	
LDURH			1					5	EAG*	1	
LDURSB			1					5	EAG*	1	
LDURSH			1					5	EAG*	1	
LDURSW			1					5	EAG*	1	
LDXP			3	✓				1 / [1]5 / [2]5	EAG* / EAGA / EAGA	3	
LDXR			1					5	EAGA	1	
LDXRB			1					5	EAGA	1	
LDXRH			1					5	EAGA	1	
LSLV	LSL (register)		1					2	EX*		
LSRV	LSR (register)		1					2	EX*		
MADD	MUL		1					5	EXA		
			2					5 / [1]1	EXA / EXA		
MOVK			1					1	EX* | EAG*		
MOVN	MOV (inverted wide immediate)		1					1	EX* | EAG*		
			1					1	EX* | EAG*		
MOVZ	MOV (wide immediate)		1					1	EX* | EAG*		
			1					1	EX* | EAG*		
MRS (*1)			2	✓	✓						
MSR (immediate) (*1)			2	✓		✓					
MSR (register) (*1)			2	✓		✓					
MSUB	MNEG		2					5 / [1]1	EXA / EXA		
			2					5 / [1]1	EXA / EXA		
ORN (shifted register)	MVN	<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
ORR (immediate)	MOV (bitmask immediate)		1					1	EX* | EAG*		
			1					1	EX* | EAG*		
ORR (shifted register)	MOV (register)	<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	EX* + EX*		
		<amount> = 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
PRFM (immediate)			1					NA	EAG*	1	
PRFM (literal)			1					NA	EAGB	1	
PRFM (register)			1					NA	EAG*	1	
PRFM (unscaled offset)			1					NA	EAG*	1	
RBIT			1					1	EX* | EAG*		
RET			1					1	EXA		
REV	REV64		1					1	EX* | EAG*		
REV16			1					1	EX* | EAG*		
REV32			1					1	EX* | EAG*		
RORV	ROR (register)		1					2	EX*		
SBC	NGC		1					1	EX*		
			1					1	EX*		
SBCS	NGCS		1					1	EX*		
			1					1	EX*		
SVC			2	✓	✓	✓		NA / NA	/		
SBFM	ASR (immediate)	<shift> = 0	1					1	EX*		
			1					2	EX*		
	SBFIZ		1				P	2+1	(EXA + EXA) | (EXB + EXB)		
	SBFX		1				P	2+1	(EXA + EXA) | (EXB + EXB)		
	SXTB		1					1	EX*		
	SXTH		1					1	EX*		
	SXTW		1					1	EX*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
SDIV		sf = 0	1				E	n (9-26)	EXB		
		sf = 1	1				E	n (9-42)	EXB		
SMADDL	SMULL		1					5	EXA		
			2					5 / [1]1	EXA / EXA		
SMC			2	✓	✓	✓		NA / NA	/		
SMSUBL	SMNEGL		2					5 / [1]1	EXA / EXA		
			2					5 / [1]1	EXA / EXA		
SMULH			1					5	EXA		
STLLR			1					NA, NA	EAG*, EXA	1	1
STLLRB			1					NA, NA	EAG*, EXA	1	1
STLLRH			1					NA, NA	EAG*, EXA	1	1
STLR			1					NA, NA	EAG*, EXA	1	1
STLRB			1					NA, NA	EAG*, EXA	1	1
STLRH			1					NA, NA	EAG*, EXA	1	1
STLXP			7	✓				1 / 8;[1]1 / 1 / [2]1 / 8;[4]1 / 1 / [2]1	EAG* / EAGA; EXA / EXA / EAG* / EAGA; EXA / EXA / EAG*	2	2
STLXR			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
STLXRB			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1
STLXRH			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1
STNP			2					NA, NA / NA, NA	EXA, EAG* / EXA, EAG*	2	2
STP		Post-index	3					NA, NA / NA, NA / 1	EAG*, EXA / EAG*, EXA / EX*| EAG*	2	2
		Pre-index	3					NA, NA / NA, NA / 1	EAG*, EXA / EAG*, EXA / EX*| EAG*	2	2
		Signed offset	2					NA, NA / NA, NA	EAG*, EXA / EAG*, EXA	2	2
STR (immediate)		Post-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Pre-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Unsigned offset	1					NA, NA	EAG*, EXA	1	1
STR (register)			1					NA, NA	EAG*, EXA	1	1
STRB (immediate)		Post-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Pre-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Unsigned offset	1					NA, NA	EAG*, EXA	1	1
STRB (register)			1					NA, NA	EAG*, EXA	1	1
STRH (immediate)		Post-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Pre-index	2					NA, NA / 1	EAG*, EXA / EX*| EAG*	1	1
		Unsigned offset	1					NA, NA	EAG*, EXA	1	1
STRH (register)			1					NA, NA	EAG*, EXA	1	1
STTR			1					NA, NA	EAG*, EXA	1	1
STTRB			1					NA, NA	EAG*, EXA	1	1
STTRH			1					NA, NA	EAG*, EXA	1	1
STUR			1					NA, NA	EAG*, EXA	1	1
STURB			1					NA, NA	EAG*, EXA	1	1
STURH			1					NA, NA	EAG*, EXA	1	1
STXP			7	✓				1 / 8;[1]1 / 1 / [2]1 / 8;[4]1 / 1 / [2]1	EAG* / EAGA; EXA / EXA / EAG* / EAGA; EXA / EXA / EAG*	2	2
STXR			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1
STXRB			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1
STXRH			3	✓				1 / 8;1 / [2]NA	EAG* / EAGA; EXA / EXA	1	1

SUB (extended register)		<amount> = 0 && (
		If sf = 0 Then
		  <extend> = {LSL|UXTW|UXTX|SXTW|SXTX}
		Else
		  <extend> = {UXTX|SXTX}
		)
			1					1	EX* | EAG*		

			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
SUB (immediate)			1					1	EX* | EAG*		
SUB (shifted register)	NEG (shifted register)	<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
		<amount> == 0	1					1	EX* | EAG*		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
		<amount> = 0	1					1	EX* | EAG*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
SUBS (extended register)	CMP (extended register)	<amount> = 0	1					1	EX*		
			1				P	1+1	(EXA + EXA) | (EXB + EXB)		

		<amount> = 0 && (
		If sf = 0 Then
		  <extend> = {LSL|UXTW|UXTX|SXTW|SXTX}
		Else
		  <extend> = {UXTX|SXTX}
		)
			1					1	EX*		

			1				P	1+1	(EXA + EXA) | (EXB + EXB)		
SUBS (immediate)	CMP (immediate)		1					1	EX*		
			1					1	EX*		
SUBS (shifted register)	CMP (shifted register)	<amount> = 0	1					1	EX*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
	NEGS	<amount> = 0	1					1	EX*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
		<amount> = 0	1					1	EX*		
		<amount> = [1-4] && <shift> = LSL	1				P	1+1	(EXA + EXA) | (EXB + EXB)		
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
SWP{|A|AL|L}			1					NA, NA	EAGA, EXA	1	1
SWP{|A|AL|L}B			1					NA, NA	EAGA, EXA	1	1
SWP{|A|AL|L}H			1					NA, NA	EAGA, EXA	1	1
SYS	AT		1					NA, NA	EAGA, EXA	1	1
	DC		1					NA, NA	EAGA, EXA	1	1
	IC		1					NA, NA	EAGA, EXA	1	1
	TLBI		1					NA, NA	EAGA, EXA	1	1
			1					NA, NA	EAGA, EXA	1	1
SYSL			2	✓				NA / NA	/		
TBNZ			1					1	EX*		
TBZ			1					1	EX*		
UBFM	LSL (immediate)	<shift> = [1-4]	1					1	EX*		
			1					2	EX*		
	LSR (immediate)	<shift> = 0	1					1	EX*		
			1					2	EX*		
	UBFIZ		1				P	2+1	(EXA + EXA) | (EXB + EXB)		
	UBFX		1				P	2+1	(EXA + EXA) | (EXB + EXB)		
	UXTB		1					1	EX*		
	UXTH		1					1	EX*		
		If sf = 1 Then immr == '000000' && imms == '011111'	1					1	EX*		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP
			1				P	2+1	(EXA + EXA) | (EXB + EXB)		
UDIV		sf = 0	1				E	n (9-25)	EXB		
		sf = 1	1				E	n (9-41)	EXB		
UMADDL	UMULL		1					5	EXA		
			2					5 / [1]1	EXA / EXA		
UMSUBL	UMNEGL		2					5 / [1]1	EXA / EXA		
			2					5 / [1]1	EXA / EXA		
UMULH			1					5	EXA		

(*1) MRS/MSR instructions are controlled differently depending on the accessed register type.

16.2. ARMv8 SIMD&FP Instructions

Table 16-2 Instruction Attributes/Latency (ARMv8 SIMD&FP)

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
ABS			1					4	FL*			
ADD (vector)			1					4	FL*			
ADDHN, ADDHN2			2	✓				4 / [1]6	FL* / FLB			
ADDP (scalar)			2	✓				6 / [1]4	FLA / FL*			
ADDP (vector)			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
ADDV			6	✓				4 / [1]4 / [1]6 / [1,2]4 / [1]4 / [1]4	FL* / FL* / FLA / FL* / FL* / FL*			
AESD			1				E	8	FLA			
AESE			1				E	8	FLA			
AESIMC			1				E	8	FLA			
AESMC			1				E	8	FLA			
AND (vector)			1					4	FL*			
BIC (vector, immediate)			1					4	FLA			
BIC (vector, register)			1					4	FL*			
BIF			1					1+4	FL* + FL*			
BIT			1					1+4	FL* + FL*			
BSL			1					1+4	FL* + FL*			
CLS (vector)			1					4	FLA			
CLZ (vector)			1					4	FLA			
CMEQ (register)			1					4	FL*			
CMEQ (zero)			1					4	FL*			
CMGE (register)			1					4	FL*			
CMGE (zero)			1					4	FL*			
CMGT (register)			1					4	FL*			
CMGT (zero)			1					4	FL*			
CMHI (register)			1					4	FL*			
CMHS (register)			1					4	FL*			
CMLE (zero)			1					4	FL*			
CMLT (zero)			1					4	FL*			
CMTST			1					4	FL*			
CNT			1					4	FLB			
DUP (element)	MOV (scalar)		1					6	FLA			
DUP (general)			1					1+3+6	EXA + NULL + FLA			
EOR (vector)			1					4	FL*			
EXT			1					6	FLA			
FABD			1					9	FL*			1
FABS (scalar)			1					4	FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FABS (vector)			1					4	FL*			
FACGE			1					4	FL*			
FACGT			1					4	FL*			
FADD (scalar)			1					9	FL*			1
FADD (vector)			1					9	FL*			1
FADDP (scalar)			2	✓				6 / [1]9	FLA / FL*			1
FADDP (vector)			3	✓				6 / 6 / [1,2]9	FLA / FLA / FL*			1
FCADD			2					6 / [1]9	FLA / FLB			1
FCCMP			1					4	FL*			
FCCMPE			1					4	FL*			
FCMEQ (register)			1					4	FL*			
FCMEQ (zero)			1					4	FL*			
FCMGE (register)			1					4	FL*			
FCMGE (zero)			1					4	FL*			
FCMGT (register)			1					4	FL*			
FCMGT (zero)			1					4	FL*			
FCMLA			3					6 / 6 / [1,2]9	FLA / FLA / FL*			2
FCMLA (by element)			3					6 / 6 / [1,2]9	FLA / FLA / FL*			2
FCMLE (zero)			1					4	FL*			
FCMLT (zero)			1					4	FL*			
FCMP			1					4	FL*			
FCMPE			1					4	FL*			
FCSEL			1					4	FL*			
FCVT			1					9	FL*			
FCVTAS (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTAS (vector)			1					9	FL*			
FCVTAU (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTAU (vector)			1					9	FL*			
FCVTL, FCVTL2		<Ta> = 4S	2					6 / [1]9	FLB / FL*			
		<Ta> = 2D	1					6	FLB			
FCVTMS (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTMS (vector)			1					9	FL*			
FCVTMU (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTMU (vector)			1					9	FL*			
FCVTN, FCVTN2			2					9 / [1]6	FL* / FLA			
FCVTNS (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTNS (vector)			1					9	FL*			
FCVTNU (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTNU (vector)			1					9	FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FCVTPS (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTPS (vector)			1					9	FL*			
FCVTPU (scalar)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTPU (vector)			1					9	FL*			
FCVTXN, FCVTXN2		Scalar	1					9	FL*			
		Vector	2					9 / [1]6	FL* / FLA			
FCVTZS (scalar, fixed-point)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTZS (scalar, integer)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTZS (vector, fixed-point)			1					9	FL*			
FCVTZS (vector, integer)			1					9	FL*			
FCVTZU (scalar, fixed-point)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTZU (scalar, integer)			1					9+1 ; 15	FLA + NULL ; EAG*	1	1	
FCVTZU (vector, fixed-point)			1					9	FL*			
FCVTZU (vector, integer)			1					9	FL*			
FDIV (scalar)		<R> = H	1				E	38	FLA			1
		<R> = S	1				E	29	FLA			
		<R> = D	1				E	43	FLA			
FDIV (vector)		<T> = {4H|8H}	1				E	38	FLA			1
		<T> = {2S|4S}	1				E	29	FLA			
		<T> = 2D	1				E	43	FLA			
FMADD			1					9	FL*			2
FMAX (scalar)			1					4	FL*			
FMAX (vector)			1					4	FL*			
FMAXNM (scalar)			1					4	FL*			
FMAXNM (vector)			1					4	FL*			
FMAXNMP (scalar)			2	✓				6 / [1]4	FLA / FL*			
FMAXNMP (vector)			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
FMAXNMV		<T> = {4H|8H}	7	✓				4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
		<T> = 4S	5	✓				4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
FMAXP (scalar)			2	✓				6 / [1]4	FLA / FL*			
FMAXP (vector)			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
FMAXV		<T>= {4H|8H}	7	✓				4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
		<T> = 4S	5	✓				4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
FMIN (scalar)			1					4	FL*			
FMIN (vector)			1					4	FL*			
FMINNM (scalar)			1					4	FL*			
FMINNM (vector)			1					4	FL*			
FMINNMP (scalar)			2	✓				6 / [1]4	FLA / FL*			
FMINNMP (vector)			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FMINNMV		<T> = {4H|8H}	7	✓				4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
		<T> = 4S	5	✓				4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
FMINP (scalar)			2	✓				6 / [1]4	FLA / FL*			
FMINP (vector)			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
FMINV		<T>= {4H|8H}	7	✓				4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
		<T> = 4S	5	✓				4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
FMLA (by element)			2	✓				6 / [1]9	FLA / FL*			2
FMLA (vector)			1					9	FL*			2
FMLS (by element)			2	✓				6 / [1]9	FLA / FL*			2
FMLS (vector)			1					9	FL*			2
FMOV (vector, immediate)			1					4	FLA			
FMOV (register)			1					4	FL*			
FMOV (general)		{Wn|Xn} to {Hd|Sd|Dd|Vd}	1					1+3+6	EXA + NULL + FLA			
		{Hn|Sn|Dn} to {Wd|Xd}	1					1 ; 13	FLA ; EAG*	1	1	
		Vn.D[1] to Xd	1					6+1 ; 18	FLA + NULL ; EAG*	1	1	
FMOV (scalar, immediate)			1					4	FLA			
FMSUB			1					9	FL*			2
FMUL (by element)			2	✓				6 / [1]9	FLA / FL*			1
FMUL (scalar)			1					9	FL*			1
FMUL (vector)			1					9	FL*			1
FMULX (by element)			2	✓				6 / [1]9	FLA / FL*			1
FMULX			1					9	FL*			1
FNEG (scalar)			1					4	FL*			
FNEG (vector)			1					4	FL*			
FNMADD			1					9	FL*			2
FNMSUB			1					9	FL*			2
FNMUL			1					9	FL*			1
FRECPE			1					4	FL*			
FRECPS			1					9	FLA			1
FRECPX			1					4	FL*			
FRINTA (scalar)			1					9	FL*			
FRINTA (vector)			1					9	FL*			
FRINTI (scalar)			1					9	FL*			
FRINTI (vector)			1					9	FL*			
FRINTM (scalar)			1					9	FL*			
FRINTM (vector)			1					9	FL*			
FRINTN (scalar)			1					9	FL*			
FRINTN (vector)			1					9	FL*			
FRINTP (scalar)			1					9	FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FRINTP (vector)			1					9	FL*			
FRINTX (scalar)			1					9	FL*			
FRINTX (vector)			1					9	FL*			
FRINTZ (scalar)			1					9	FL*			
FRINTZ (vector)			1					9	FL*			
FRSQRTE			1					4	FL*			
FRSQRTS			1					9	FLA			1
FSQRT (scalar)		<R> = H	1				E	38	FLA			1
		<R> = S	1				E	29	FLA			
		<R> = D	1				E	43	FLA			
FSQRT (vector)		<T> = {4H|8H}	1				E	38	FLA			1
		<T> = {2S|4S}	1				E	29	FLA			
		<T> = 2D	1				E	43	FLA			
FSUB (scalar)			1					9	FL*			1
FSUB (vector)			1					9	FL*			1
INS (element)	MOV (element)		1					6	FLA			
INS (general)	MOV (from general)		1					1+3+6	EXA + NULL + FLA			
LD1 (multiple structures)		No offset 1 register <T> = {8B|4H|2S|2D|1D}	1					8	EAG*	1		
		No offset 1 register <T> = {16B|8H|4S}	1					11	EAG*	1		
		No offset 2 registers <T> = {8B|4H|2S|2D|1D}	2					8 / 8	EAG* / EAG*	2		
		No offset 2 registers <T> = {16B|8H|4S}	2					11 / 11	EAG* / EAG*	2		
		No offset 3 registers <T> = {8B|4H|2S|2D|1D}	3					8 / 8 / 8	EAG* / EAG* / EAG*	3		
		No offset 3 registers <T> = {16B|8H|4S}	3					11 / 11 / 11	EAG* / EAG* / EAG*	3		
		No offset 4 registers <T> = {8B|4H|2S|2D|1D}	4					8 / 8 / 8 / 8	EAG* / EAG* / EAG* / EAG*	4		
		No offset 4 registers <T> = {16B|8H|4S}	4					11 / 11 / 11 / 11	EAG* / EAG* / EAG* / EAG*	4		
		Post-index 1 register <T> = {8B|4H|2S|2D|1D}	2					8 / 1	EAG* / EAG*	1		
		Post-index 1 register <T> = {16B|8H|4S}	2					11 / 1	EAG* / EAG*	1		
		Post-index 2 registers <T> = {8B|4H|2S|2D|1D}	3					8 / 8 / 1	EAG* / EAG* / EAG*	2		
		Post-index 2 registers <T> = {16B|8H|4S}	3					11 / 11 / 1	EAG* / EAG* / EAG*	2		
		Post-index 3 registers <T> = {8B|4H|2S|2D|1D}	4					8 / 8 / 8 / 1	EAG* / EAG* / EAG* / EAG*	3		
		Post-index 3 registers <T> = {16B|8H|4S}	4					11 / 11 / 11 / 1	EAG* / EAG* / EAG* / EAG*	3		
		Post-index 4 registers <T> = {8B|4H|2S|2D|1D}	5					8 / 8 / 8 / 8 / 1	EAG* / EAG* / EAG* / EAG* / EAG*	4		
		Post-index 4 registers <T> = {16B|8H|4S}	5					11 / 11 / 11 / 11 / 1	EAG* / EAG* / EAG* / EAG* / EAG*	4		
LD1 (single structure)		No offset	2	✓				8 / 6	EAG* / FLA	1		
		Post-index	3	✓				8 / 6 / 1	EAG* / FLA / EAG*	1		
LD1R		No offset	1					8	EAG*	1		
		Post-index	2					8 / 1	EAG* / EAG*	1		
LD2 (multiple structures)		No offset	2					11 / 11	EAG* / EAG*	2		
		Post-index	3					11 / 11 / 1	EAG* / EAG* / EAG*	2		

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
LD2 (single structure)		No offset	4	✓				(8 / 6) x 2	(EAG* / FLA) x 2	2		
		Post-index	5	✓				(8 / 6) x 2 / 1	(EAG* / FLA) x 2 / EAG*	2		
LD2R		No offset	2					8 / 8	EAG*/EAG*	2		
		Post-index	3					8 / 8 / 1	EAG* / EAG* / EAG*	2		
LD3 (multiple structures)		No offset	3					11 / 11 / 11	EAG* / EAG* / EAG*	3		
		Post-index	4					11 / 11 / 11 / 1	EAG* / EAG* / EAG* / EAG*	3		
LD3 (single structure)		No offset	6	✓				(8 / 6) x 3	(EAG* / FLA) x 3	3		
		Post-index	7	✓				(8 / 6) x 3 / 1	(EAG* / FLA) x 3 / EAG*	3		
LD3R		No offset	3					8 / 8 / 8	EAG* / EAG* / EAG*	3		
		Post-index	4					8 / 8 / 8 / 1	EAG* / EAG* / EAG* / EAG*	3		
LD4 (multiple structures)		No offset	4					11 / 11 / 11 / 11	EAG* / EAG* / EAG* / EAG*	4		
		Post-index	5					11 / 11 / 11 / 11 / 1	EAG* / EAG* / EAG* / EAG* / EAG*	4		
LD4 (single structure)		No offset	8	✓				(8 / 6) x 4	(EAG* / FLA) x 4	4		
		Post-index	9	✓				(8 / 6) x 4 / 1	(EAG* / FLA) x 4 / EAG*	4		
LD4R		No offset	4					8 / 8 / 8 / 8	EAG* / EAG* / EAG* / EAG*	4		
		Post-index	5					8 / 8 / 8 / 8 / 1	EAG* / EAG* / EAG* / EAG* / EAG*	4		
LDNP (SIMD&FP)			2					8 / 8	EAG* / EAG*	2		
LDP (SIMD&FP)		Post-index	3					8 / 8 / 1	EAG* / EAG* / EX*| EAG*	2		
		Pre-index	3					8 / 8 / 1	EAG* / EAG* / EX*| EAG*	2		
		Signed offset	2					8 / 8	EAG* / EAG*	2		
LDR (immediate, SIMD&FP)		Post-index	2					8 / 1	EAG* / EX*| EAG*	1		
		Pre-index	2					8 / 1	EAG* / EX*| EAG*	1		
		Unsigned offset	1					8	EAG*	1		
LDR (literal, SIMD&FP)			1					8	EAGB	1		
LDR (register, SIMD&FP)			1					8	EAG*	1		
LDUR (SIMD&FP)			1					8	EAG*	1		
MLA (by element)			2	✓				6 / [1]9	FLA / FL*			
MLA (vector)			1					9	FL*			
MLS (by element)			2	✓				6 / [1]9	FLA / FL*			
MLS (vector)			1					9	FL*			
MOVI			1					4	FLA			
MUL (by element)			2	✓				6 / [1]9	FLA / FL*			
MUL (vector)			1					9	FL*			
MVNI			1					4	FLA			
NEG (vector)			1					4	FL*			
NOT	MVN		1					4	FL*			
ORN (vector)			1					4	FL*			
ORR (vector, immediate)			1					4	FLA			
ORR (vector, register)	MOV (vector)		1					4	FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
			1					4	FL*			
PMUL			1				E	8	FLA			
PMULL, PMULL2			1				E	8	FLA			
RADDHN, RADDHN2			3	✓				4 / [1]4 / [1]6	FL* / FL* / FLB			
RBIT (vector)			1					4	FL*			
REV16 (vector)			1					4	FL*			
REV32 (vector)			1					4	FL*			
REV64			1					4	FL*			
RSHRN, RSHRN2			3	✓				4 / [1]4 / [1]6	FL* / FL* / FLB			
RSUBHN, RSUBHN2			3	✓				4 / [1]4 / [1]6	FL* / FLA / FLB			
SABA			2	✓				4 / [1]4	FL* / FL*			
SABAL, SABAL2			4	✓				6 / 6 / [1,2]4 / (1)4	FLB / FLB / FL* / FL*			
SABD			1					4	FL*			
SABDL, SABDL2			3	✓				6 / 6 / [1,2]4	FLB / FLB / FL*			
SADALP			3	✓				6 / [1]4 / [1]4	FLB / FL* / FL*			
SADDL, SADDL2			3					6 / 6 / [1,2]4	FLB / FLB / FL*			
SADDLP			2	✓				6 / [1]4	FLB / FL*			
SADDLV			6	✓				4 / [1]4 / [1]6 / [1,2]4 / [1]4 / [1]4	FL* / FL* / FLA / FL* / FL* / FL*			
SADDW, SADDW2			2	✓				6 / [1]4	FLB / FL*			
SCVTF (scalar, fixed-point)			1					1+3+9	EXA + NULL + FLA			
SCVTF (scalar, integer)			1					1+3+9	EXA + NULL + FLA			
SCVTF (vector, fixed-point)			1					9	FL*			
SCVTF (vector, integer)			1					9	FL*			
SHA1C			1				E	1+11	FLA + FLA			
SHA1H			1				E	8	FLA			
SHA1M			1				E	1+11	FLA + FLA			
SHA1P			1				E	1+11	FLA + FLA			
SHA1SU0			1				E	1+8	FLA + FLA			
SHA1SU1			1				E	8	FLA			
SHA256H2			1				E	1+11	FLA + FLA			
SHA256H			1				E	1+11	FLA + FLA			
SHA256SU0			1				E	8	FLA			
SHA256SU1			1				E	1+8	FLA + FLA			
SHADD			1					4	FL*			
SHL			1					4	FL*			
SHLL, SHLL2			2					6 / [1]4	FLB / FL*			
SHRN, SHRN2			2	✓				4 / [1]6	FL* / FLB			
SHSUB			1					4	FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
SLI			3	✓				4 / 4 / [1,2]4	FL* / FLA / FL*			
SMAX			1					4	FL*			
SMAXP			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
SMAXV			6	✓				4 / [1]6 / [1,2]4 / [1]4 / [1]4 / [1]4	FL* / FLA / FL* / FL* / FL* / FL*			
SMIN			1					4	FL*			
SMINP			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
SMINV			6	✓				4 / [1]6 / [1,2]4 / [1]4 / [1]4 / [1]4	FL* / FLA / FL* / FL* / FL* / FL*			
SMLAL, SMLAL2 (by element)		<Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
SMLAL, SMLAL2 (vector)		<Ta> = {8H|4S}	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
SMLSL, SMLSL2 (by element)		<Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
SMLSL, SMLSL2 (vector)		<Ta> = {8H|4S}	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
SMOV			1					6 + 1 + 18	FLA + NULL + EAG*	1	1	
SMULL, SMULL2 (by element)		<Ta> = 4S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLA / FLA			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
SMULL, SMULL2 (vector)		<Ta> = {8H|4S}	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLB / FLA			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
SQABS			1					4	FL*			
SQADD			1					4	FL*			
SQDMLAL, SQDMLAL2 (by element)		Scalar <Va> = S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		Scalar <Va> = D	3	✓				6 / [1]9 / [1]4	FLA / FL* / FL*			
		Vector <Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		Vector <Ta> = 2D	4	✓				6 / 6 / [1,2]9 / [1]4	FLB / FLA / FL* / FL*			
SQDMLAL, SQDMLAL2 (vector)		Scalar <Va> = S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FLA			
		Scalar <Va> = D	2	✓				9 / [1]4	FL* / FL*			
		Vector <Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FLA			
		Vector <Ta> = 2D	4	✓				6 / 6 / [1,2]9 / [1]4	FLB / FLB / FL* / FL*			
SQDMLSL, SQDMLSL2 (by element)		Scalar <Va> = S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		Scalar <Va> = D	3	✓				6 / [1]9 / [1]4	FLA / FL* / FL*			
		Vector <Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		Vector <Ta> = 2D	4	✓				6 / 6 / [1,2]9 / [1]4	FLB / FLA / FL* / FL*			
SQDMLSL, SQDMLSL2 (vector)		Scalar <Va> = S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FLA			
		Scalar <Va> = D	2	✓				9 / [1]4	FL* / FL*			
		Vector <Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FLA			
		Vector <Ta> = 2D	4	✓				6 / 6 / [1,2]9 / [1]4	FLB / FLB / FL* / FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
SQDMULH (by element)		Scalar <V> = H, Vector <T> = {4H|8H}	2	✓			/ E	6 / [1]8	FLA / FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	2	✓				6 / [1]9	FLA / FL*			
SQDMULH (vector)		Scalar <V> = H, Vector <T> = {4H|8H}	1				E	8	FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	1					9	FL*			
SQDMULL, SQDMULL2 (by element)		Scalar <Va> = S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLA / FLA			
		Scalar <Va> = D	2	✓				6 / [1]9	FLA / FL*			
		Vector <Ta> = 4S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLA / FLA			
		Vector <Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
SQDMULL, SQDMULL2 (vector)		Scalar <Va> = S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLB / FLA			
		Scalar <Va> = D	1					9	FL*			
		Vector <Ta> = 4S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLB / FLA			
		Vector <Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
SQNEG			1					4	FL*			
SQRDMLAH (by element)		Scalar <V> = H, Vector <T> = {4H|8H}	2	✓			/ E	6 / 1+[1]8	FLA / FLA + FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	2	✓				6 / [1]9	FLA / FL*			
SQRDMLAH (vector)		Scalar <V> = H, Vector <T> = {4H|8H}	1				E	1+8	FLA + FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	1					9	FL*			
SQRDMLSH (by element)		Scalar <V> = H, Vector <T> = {4H|8H}	2	✓			/ E	6 / 1+[1]8	FLA / FLA + FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	2	✓				6 / [1]9	FLA / FL*			
SQRDMLSH (vector)		Scalar <V> = H, Vector <T> = {4H|8H}	1				E	1+8	FLA + FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	1					9	FL*			
SQRDMULH (by element)		Scalar <V> = H, Vector <T> = {4H|8H}	2	✓			/ E	6 / [1]8	FLA / FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	2	✓				6 / [1]9	FLA / FL*			
SQRDMULH (vector)		Scalar <V> = H, Vector <T> = {4H|8H}	1				E	8	FLA			
		Scalar <V> = S, Vector <T> = {2S|4S}	1					9	FL*			
SQRSHL			2	✓				6 / [1]4	FLB / FL*			
SQRSHRN, SQRSHRN2			3	✓				4 / [1]4 / [1]6	FL* / FL* / FLB			
SQRSHRUN, SQRSHRUN2			3	✓				4 / [1]4 / [1]6	FL* / FL* / FLB			
SQSHL (immediate)			1					6	FLB			
SQSHL (register)			1					6	FLB			
SQSHLU			1					6	FLB			
SQSHRN, SQSHRN2			2	✓				4 / [1]6	FL* / FLB			
SQSHRUN, SQSHRUN2			2	✓				4 / [1]6	FL* / FLB			
SQSUB			1					4	FL*			
SQXTN, SQXTN2			1					6	FLB			
SQXTUN, SQXTUN2			1					6	FLB			
SRHADD			1					4	FL*			
SRI			3	✓				4 / 4 / [1,2]4	FL* / FLA / FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
SRSHL			2	✓				6 / [1]4	FLB / FL*			
SRSHR			2	✓				4 / [1]4	FL* / FL*			
SRSRA			3	✓				4 / [1]4 / [1]4	FL* / FL* / FL*			
SSHL			1					6	FLB			
SSHLL, SSHLL2	SXTL, SXTL2		2					6 / [1]4	FLB / FL*			
			2					6 / [1]4	FLB / FL*			
SSHR			1					4	FL*			
SSRA			2	✓				4 / [1]4	FL* / FL*			
SSUBL, SSUBL2			3					6 / 6 / [1,2]4	FLB / FLB / FL*			
SSUBW, SSUBW2			2	✓				6 / [1]4	FLB / FL*			
ST1 (multiple structures)		No offset 1 register	1					NA, NA	EAG*, FLA	1	1	
		No offset 2 registers	2					(NA, NA) x 2	(EAG*, FLA) x 2	2	2	
		No offset 3 registers	3					(NA, NA) x 3	(EAG*, FLA) x 3	3	3	
		No offset 4 registers	4					(NA, NA) x 4	(EAG*, FLA) x 4	4	4	
		Post-index 1 register	2					NA, NA / 1	EAG*, FLA / EAG*	1	1	
		Post-index 2 registers	3					(NA, NA) x 2 / 1	(EAG*, FLA) x 2 / EAG*	2	2	
		Post-index 3 registers	4					(NA, NA) x 3 / 1	(EAG*, FLA) x 3 / EAG*	3	3	
		Post-index 4 registers	5					(NA, NA) x 4 / 1	(EAG*, FLA) x 4 / EAG*	4	4	
ST1 (single structure)		No offset	1					NA, NA	EAG*, FLA	1	1	
		Post-index	2					NA, NA / 1	EAG*, FLA / EAG*	1	1	
ST2 (multiple structures)		No offset	2					(NA, NA) x 2	(EAG*, FLA) x 2	2	2	
		Post-index	3					(NA, NA) x 2 / 1	(EAG*, FLA) x 2 / EAG*	2	2	
ST2 (single structure)		No offset	2					(NA, NA) x 2	(EAG*, FLA) x 2	2	2	
		Post-index	3					(NA, NA) x 2 / 1	(EAG*, FLA) x 2 / EAG*	2	2	
ST3 (multiple structures)		No offset	3					(NA, NA) x 3	(EAG*, FLA) x 3	3	3	
		Post-index	4					(NA, NA) x 3 / 1	(EAG*, FLA) x 3 / EAG*	3	3	
ST3 (single structure)		No offset	3					(NA, NA) x 3	(EAG*, FLA) x 3	3	3	
		Post-index	4					(NA, NA) x 3 / 1	(EAG*, FLA) x 3 / EAG*	3	3	
ST4 (multiple structures)		No offset	4					(NA, NA) x 4	(EAG*, FLA) x 4	4	4	
		Post-index	5					(NA, NA) x 4 / 1	(EAG*, FLA) x 4 / EAG*	4	4	
ST4 (single structure)		No offset	4					(NA, NA) x 4	(EAG*, FLA) x 4	4	4	
		Post-index	5					(NA, NA) x 4 / 1	(EAG*, FLA) x 4 / EAG*	4	4	
STNP (SIMD&FP)			2					(NA, NA) x 2	(EAG*, FLA) x 2	2	2	
STP (SIMD&FP)		Post-index	3					(NA, NA) x 2 / 1	(EAG*, FLA) x 2 / EX*| EAG*	2	2	
		Pre-index	3					(NA, NA) x 2 / 1	(EAG*, FLA) x 2 / EX*| EAG*	2	2	
		Signed offset	2					(NA, NA) x 2	(EAG*, FLA) x 2	2	2	
STR (immediate, SIMD&FP)		Post-index	2					NA, NA / 1	EAG*, FLA / EX*| EAG*	1	1	
		Pre-index	2					NA, NA / 1	EAG*, FLA / EX*| EAG*	1	1	
		Unsigned offset	1					NA, NA	EAG*, FLA	1	1	

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
STR (register, SIMD&FP)			1					NA, NA	EAG*, FLA	1	1	
STUR (SIMD&FP)			1					NA, NA	EAG*, FLA	1	1	
SUB (vector)			1					4	FL*			
SUBHN, SUBHN2			2	✓				4 / [1]6	FL* / FLB			
SUQADD			1					4	FL*			
TBL		Single register table	1					6	FLB			
		Tow register table	3	✓				6 / 6 / [1,2]4	FLB / FLB / FL*			
		Three register table	5	✓				6 / (6 / [1,2]4) x 2	FLB / (FLB / FL*) x 2			
		Four register table	7	✓				6 / (6 / [1,2]4) x 3	FLB / (FLB / FL*) x 3			
TBX		Single register table	3	✓				6 / 6 / [1,2]4	FLB / FLB / FL*			
		Tow register table	5	✓				6 / (6 / [1,2]4) x 2	FLB / (FLB / FL*) x 2			
		Three register table	7	✓				6 / (6 / [1,2]4) x 3	FLB / (FLB / FL*) x 3			
		Four register table	9	✓				6 / (6 / [1,2]4) x 4	FLB / (FLB / FL*) x 4			
TRN1			1					6	FLA			
TRN2			1					6	FLA			
UABA			2	✓				4 / [1]4	FL* / FL*			
UABAL, UABAL2			4	✓				6 / 6 / [1,2]4 / [1]4	FLB / FLB / FL* / FL*			
UABD			1					4	FL*			
UABDL, UABDL2			3	✓				6 / 6 / [1,2]4	FLB / FLB / FL*			
UADALP		<Ta> = {4H|8H|2S|4S}	1					6	FLB			
		<Ta> = {1D|2D}	3	✓				6 / [1]4 / [1]4	FLB / FL* / FL*			
UADDL, UADDL2			3					6 / 6 / [1,2]4	FLB / FLB / FL*			
UADDLP			2	✓				6 / [1]4	FLB / FL*			
UADDLV			6	✓				4 / [1]4 / [1]6 / [1,2]4 / [1]4 / [1]4	FL* / FL* / FLA / FL* / FL* / FL*			
UADDW, UADDW2			2	✓				6 / [1]4	FLB / FL*			
UCVTF (scalar, fixed-point)			1					1+3+9	EXA + NULL + FLA			
UCVTF (scalar, integer)			1					1+3+9	EXA + NULL + FLA			
UCVTF (vector, fixed-point)			1					9	FL*			
UCVTF (vector, integer)			1					9	FL*			
UHADD			1					4	FL*			
UHSUB			1					4	FL*			
UMAX			1					4	FL*			
UMAXP			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
UMAXV			6	✓				4 / [1]6 / [1]4 / [1,2]4 / [1]4 / [1]4	FL* / FLA / FL* / FL* / FL* / FL*			
UMIN			1					4	FL*			
UMINP			3	✓				6 / 6 / [1,2]4	FLA / FLA / FL*			
UMINV			6	✓				4 / [1]6 / [1]4 / [1,2]4 / [1]4 / [1]4	FL* / FLA / FL* / FL* / FL* / FL*			
UMLAL, UMLAL2 (by element)		<Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			

Instruction	Alias	Control option	# of µOP	Seq. decode	Pre-sync	Post-sync	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
UMLAL, UMLAL2 (vector)		<Ta> = {8H|4S}	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
UMLSL, UMLSL2 (by element)		<Ta> = 4S	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLA / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
UMLSL, UMLSL2 (vector)		<Ta> = {8H|4S}	4	✓			/ / E /	6 / 6 / [1,2]8 / [1]4	FLB / FLB / FLA / FL*			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
UMOV	MOV (to general)		1					6+1+18	FLA + NULL + EAG*	1	1	
			1					6+1+18	FLA + NULL + EAG*	1	1	
UMULL, UMULL2 (by element)		<Ta> = 4S	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLA / FLA			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLA / FL*			
UMULL, UMULL2 (vector)		<Ta> = {8H|4S}	3	✓			/ / E	6 / 6 / [1,2]8	FLB / FLB / FLA			
		<Ta> = 2D	3	✓				6 / 6 / [1,2]9	FLB / FLB / FL*			
UQADD			1					4	FL*			
UQRSHL			2	✓				6 / [1]4	FLB / FL*			
UQRSHRN, UQRSHRN2			3	✓				4 / [1]4 / [1]6	FL* / FL* / FLB			
UQSHL (immediate)			1					6	FLB			
UQSHL (register)			1					6	FLB			
UQSHRN, UQSHRN2			2	✓				4 / [1]6	FL* / FLB			
UQSUB			1					4	FL*			
UQXTN, UQXTN2			1					6	FLB			
URECPE			1					4	FL*			
URHADD			1					4	FL*			
URSHL			2	✓				6 / [1]4	FLB / FL*			
URSHR			2	✓				4 / [1]4	FL* / FL*			
URSQRTE			1					4	FL*			
URSRA			3	✓				4 / [1]4 / [1]4	FL* / FL* / FL*			
USHL			1					6	FLB			
USHLL, USHLL2	UXTL, UXTL2		2					6 / [1]4	FLB / FL*			
			2					6 / [1]4	FLB / FL*			
USHR			1					4	FL*			
USQADD			1					4	FL*			
USRA			2	✓				4 / [1]4	FL* / FL*			
USUBL, USUBL2			3					6 / 6 / [1,2]4	FLB / FLB / FL*			
USUBW, USUBW2			2	✓				6 / [1]4	FLB / FL*			
UZP1			1					6	FLA			
UZP2			1					6	FLA			
XTN, XTN2			1					6	FLB			
ZIP1			1					6	FLA			
ZIP2			1					6	FLA			

16.3. SVE Instructions

Table 16-3 Instruction Attributes/Latency (SVE)

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
ABS				1				✓			4	FL*			
ADD (immediate)				1				✓			4	FL*			
ADD (vectors, predicated)				1				✓	✓		4	FL*			
ADD (vectors, unpredicated)				1							4	FL*			
ADDPL				1							1	EX*			
ADDVL				1							1	EX*			
ADR		Packed offsets		1							1+4	FLA + FLA			
				1							4	FLA			
AND (immediate)	BIC (immediate)			1				✓			4	FLA			
AND (predicates)	MOV (predicate, predicated, zeroing)			1							3	PRX			
				1							3	PRX			
AND (vectors, predicated)				1				✓	✓		4	FL*			
AND (vectors, unpredicated)				1							4	FL*			
ANDS (predicates)	MOVS (predicated)			1							3	PRX			
				1							3	PRX			
ANDV		<V> = B		10	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
ASR (immediate, predicated)				1				✓	✓		4	FL*			
ASR (immediate, unpredicated)				1							4	FL*			
ASR (vectors)				1				✓	✓		4	FL*			
ASR (wide elements, predicated)				1				✓	✓		4	FL*			
ASR (wide elements, unpredicated)				1							4	FL*			
ASRD				2	✓			✓	✓		4 / [1]4	FLA / FL*			
ASRR				1				✓	✓		4	FL*			
BIC (predicates)				1							3	PRX			
BIC (vectors, predicated)				1				✓	✓		4	FL*			
BIC (vectors, unpredicated)				1							4	FL*			
BICS (predicates)				1							3	PRX			
BRKA				1							3	PRX			
BRKAS				1							3	PRX			
BRKB				1							3	PRX			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
BRKBS				1							3	PRX			
BRKN				1							3	PRX			
BRKNS				1							3	PRX			
BRKPA				1							3	PRX			
BRKPAS				1							3	PRX			
BRKPB				1							3	PRX			
BRKPBS				1							3	PRX			
CLASTA (scalar)				1							1+3+6+1+18	EXA + NULL + EAG* + NULL + FLA	1	1	
CLASTA (SIMD&FP scalar)				1							6	FLA			
CLASTA (vectors)				1				✓			6	FLA			
CLASTB (scalar)				1							1+3+6+1+18	EXA + NULL + EAG* + NULL + FLA	1	1	
CLASTB (SIMD&FP scalar)				1							6	FLA			
CLASTB (vectors)				1				✓			6	FLA			
CLS				1				✓			4	FLA			
CLZ				1				✓			4	FLA			
CMPEQ (immediate)				1							4	PRX, FLA			
CMPEQ (vectors)				1							4	PRX, FLA			
CMPEQ (wide elements)				1							4	PRX, FLA			
CMPGE (immediate)				1							4	PRX, FLA			
CMPGE (vectors)	CMPLE (vectors)			1							4	PRX, FLA			
CMPGE (wide elements)				1							4	PRX, FLA			
CMPGT (immediate)				1							4	PRX, FLA			
CMPGT (vectors)	CMPLT (vectors)			1							4	PRX, FLA			
CMPGT (wide elements)				1							4	PRX, FLA			
CMPHI (immediate)				1							4	PRX, FLA			
CMPHI (vectors)	CMPLO (vectors)			1							4	PRX, FLA			
CMPHI (wide elements)				1							4	PRX, FLA			
CMPHS (immediate)				1							4	PRX, FLA			
CMPHS (vectors)	CMPLS (vectors)			1							4	PRX, FLA			
CMPHS (wide elements)				1							4	PRX, FLA			
CMPLE (immediate)				1							4	PRX, FLA			
CMPLE (wide elements)				1							4	PRX, FLA			
CMPLO (immediate)				1							4	PRX, FLA			
CMPLO (wide elements)				1							4	PRX, FLA			
CMPLS (immediate)				1							4	PRX, FLA			
CMPLS (wide elements)				1							4	PRX, FLA			
CMPLT (immediate)				1							4	PRX, FLA			
CMPLT (wide elements)				1							4	PRX, FLA			
CMPNE (immediate)				1							4	PRX, FLA			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
CMPNE (vectors)				1							4	PRX, FLA			
CMPNE (wide elements)				1							4	PRX, FLA			
CNOT				1				✓			4	FL*			
CNT				1				✓			4	FLB			
CNTB				1							1	EX*			
CNTD				1							1	EX*			
CNTH				1							1	EX*			
CNTP				1							3+2+1	PRX + NULL + EXA			
CNTW				1							1	EX*			
COMPACT				1							6	FLA			
CPY (immediate)	FMOV (zero, predicated)			1				✓			4	FLA			
	MOV (immediate, predicated)			1				✓			4	FLA			
CPY (scalar)	MOV (scalar, predicated)			1				✓			1+3+4	EXA + NULL + FLA			
CPY (SIMD&FP scalar)	MOV (SIMD&FP scalar, predicated)			1				✓			6	FLA			
CTERMEQ				1						E	1+1	EX* + EX*			
CTERMNE				1						E	1+1	EX* + EX*			
DECB				1							1	EX*			
DECD (scalar)				1							1	EX*			
DECD (vector)				1				✓			4	FL*			
DECH (scalar)				1							1	EX*			
DECH (vector)				1				✓			4	FL*			
DECP (scalar)				2							3+2+1 / [1]1	PRX+NULL+EXA / EXB			
DECP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
DECW (scalar)				1							1	EX*			
DECW (vector)				1				✓			4	FL*			
DUP (immediate)	FMOV (zero, unpredicated)			1							4	FLA			
	MOV (immediate, unpredicated)			1							4	FLA			
DUP (indexed)	MOV (SIMD&FP scalar, unpredicated)			1							6	FLA			
				1							6	FLA			
DUP (scalar)	MOV (scalar, unpredicated)			1							1+3+4	EXA + NULL + FLA			
DUPM	MOV (bitmask immediate)			1							4	FLA			
				1							4	FLA			
EOR (immediate)	EON			1				✓			4	FLA			
EOR (predicates)	NOT (predicate)			1							3	PRX			
				1							3	PRX			
EOR (vectors, predicated)				1				✓	✓		4	FL*			
EOR (vectors, unpredicated)				1							4	FL*			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
EORS	NOTS			1							3	PRX			
				1							3	PRX			
EORV		<V> = B		10	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
EXT				1				✓			6	FLA			
FABD				1				✓			9	FL*			1
FABS				1				✓			4	FL*			
FACGE	FACLE			1							4	FLA			
FACGT	FACLT			1							4	FLA			
FADD (immediate)				1				✓			9	FLA			1
FADD (vectors, predicated)				1				✓			9	FL*			1
FADD (vectors, unpredicated)				1							9	FL*			1
FADDA		<V> = H	128	15	✓						9 / 6 / ([1,2]9 / [1]6) x 6 / [1,2]9	FL* / FLA / (FL* / FLA) x 6 / FL*			1
			256	31	✓						9 / 6 / ([1,2]9 / [1]6) x 14 / [1,2]9	FL* / FLA / (FL* / FLA) x 14 / FL*			
			512	63	✓						9 / 6 / ([1,2]9 / [1]6) x 30 / [1,2]9	FL* / FLA / (FL* / FLA) x 30 / FL*			
		<V> = S	128	7	✓						9 / 6 / ([1,2]9 / [1]6) x 2 / [1,2]9	FL* / FLA / (FL* / FLA) x 2 / FL*			
			256	15	✓						9 / 6 / ([1,2]9 / [1]6) x 6 / [1,2]9	FL* / FLA / (FL* / FLA) x 6 / FL*			
			512	31	✓						9 / 6 / ([1,2]9 / [1]6) x 14 / [1,2]9	FL* / FLA / (FL* / FLA) x 14 / FL*			
		<V> = D	128	3	✓						9 / 6 / [1,2]9	FL* / FLA / FL*			
			256	7	✓						9 / 6 / ([1,2]9 / [1]6) x 2 / [1,2]9	FL* / FLA / (FL* / FLA) x 2 / FL*			
			512	15	✓						9 / 6 / ([1,2]9 / [1]6) x 6 / [1,2]9	FL* / FLA / (FL* / FLA) x 6 / FL*			
FADDV		<V> = H	128	7	✓						4 / 6 / ([1,2]9 / [1]6) x 2 / [1,2]9	FL* / FLA / (FL* / FLA) x 2 / FL*			1
			256	9	✓						4 / 6 / ([1,2]9 / [1]6) x 3 / [1,2]9	FL* / FLA / (FL* / FLA) x 3 / FL*			
			512	11	✓						4 / 6 / ([1,2]9 / [1]6) x 4 / [1,2]9	FL* / FLA / (FL* / FLA) x 4 / FL*			
		<V> = S	128	5	✓						4 / 6 / [1,2]9 / [1]6 / [1,2]9	FL* / FLA / FL* / FLA / FL*			
			256	7	✓						4 / 6 / ([1,2]9 / [1]6) x 2 / [1,2]9	FL* / FLA / (FL* / FLA) x 2 / FL*			
			512	9	✓						4 / 6 / ([1,2]9 / [1]6) x 3 / [1,2]9	FL* / FLA / (FL* / FLA) x 3 / FL*			
		<V> = D	128	3	✓						4 / 6 / [1,2]9	FL* / FLA / FL*			
			256	5	✓						4 / 6 / [1,2]9 / [1]6 / [1,2]9	FL* / FLA / FL* / FLA / FL*			
			512	7	✓						4 / 6 / ([1,2]9 / [1]6) x 2 / [1,2]9	FL* / FLA / (FL* / FLA) x 2 / FL*			
FCADD				2							6 / [1]9	FLA / FLB			1
FCMEQ (vectors)				1							4	FLA			
FCMEQ (zero)				1							4	FLA			
FCMGE (vectors)	FCMLE (vectors)			1							4	FLA			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FCMGE (zero)				1							4	FLA			
FCMGT (vectors)	FCMLT (vectors)			1							4	FLA			
FCMGT (zero)				1							4	FLA			
FCMLA (indexed)				3							6 / 6 / [1,2]9	FLA / FLA / FL*			2
FCMLA (vectors)				3							6 / 6 / [1,2]9	FLA / FLA / FL*			2
FCMLE (zero)				1							4	FLA			
FCMLT (zero)				1							4	FLA			
FCMNE (vectors)				1							4	FLA			
FCMNE (zero)				1							4	FLA			
FCMUO				1							4	FLA			
FCPY	FMOV (immediate, predicated)			1				✓			4	FLA			
FCVT				1				✓			9	FL*			
FCVTZS				1				✓			9	FL*			
FCVTZU				1				✓			9	FL*			
FDIV		<T> = H	128	1				✓	✓	E	38	FLA			1
			256	1				✓	✓	E	70	FLA			
			512	1				✓	✓	E	134	FLA			
		<T> = S	128	1				✓	✓	E	29	FLA			
			256	1				✓	✓	E	52	FLA			
			512	1				✓	✓	E	98	FLA			
		<T> = D	128	1				✓	✓	E	43	FLA			
			256	1				✓	✓	E	80	FLA			
			512	1				✓	✓	E	154	FLA			
FDIVR		<T> = H	128	1				✓	✓	E	38	FLA			1
			256	1				✓	✓	E	70	FLA			
			512	1				✓	✓	E	134	FLA			
		<T> = S	128	1				✓	✓	E	29	FLA			
			256	1				✓	✓	E	52	FLA			
			512	1				✓	✓	E	98	FLA			
		<T> = D	128	1				✓	✓	E	43	FLA			
			256	1				✓	✓	E	80	FLA			
			512	1				✓	✓	E	154	FLA			
FDUP	FMOV (immediate, unpredicated)			1							4	FLA			
FEXPA				1							4	FL*			
FMAD				1				✓	✓		9	FL*			2
FMAX (immediate)				1				✓	✓		4	FLA			
FMAX (vectors)				1				✓	✓		4	FL*			
FMAXNM (immediate)				1				✓	✓		4	FLA			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
FMAXNM (vectors)				1				✓	✓		4	FL*			
FMAXNMV		<V> = H	128	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			256	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
			512	11	✓						4 / ([1]6 / [1,2]4) x 5	FL* / (FLA / FL*) x 5			
		<V> = S	128	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			256	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			512	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
FMAXV		<V> = H	128	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			256	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
			512	11	✓						4 / ([1]6 / [1,2]4) x 5	FL* / (FLA / FL*) x 5			
		<V> = S	128	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			256	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			512	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
FMIN (immediate)				1				✓	✓		4	FLA			
FMIN (vectors)				1				✓	✓		4	FL*			
FMINNM (immediate)				1				✓	✓		4	FLA			
FMINNM (vectors)				1				✓	✓		4	FL*			
FMINNMV		<V> = H	128	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			256	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
			512	11	✓						4 / ([1]6 / [1,2]4) x 5	FL* / (FLA / FL*) x 5			
		<V> = S	128	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			256	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			512	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
FMINV		<V> = H	128	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			256	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			
			512	11	✓						4 / ([1]6 / [1,2]4) x 5	FL* / (FLA / FL*) x 5			
		<V> = S	128	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			256	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
			512	9	✓						4 / ([1]6 / [1,2]4) x 4	FL* / (FLA / FL*) x 4			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
FMLA				1				✓	✓		9	FL*			2
FMLA (indexed)				2							6 / [1]9	FLA / FLB			2
FMLS				1				✓	✓		9	FL*			2
FMLS (indexed)				2							6 / [1]9	FLA / FLB			2
FMSB				1				✓	✓		9	FL*			2
FMUL (immediate)				1				✓			9	FLA			1
FMUL (indexed)				2							6 / [1]9	FLA / FLB			1
FMUL (vectors, predicated)				1				✓			9	FL*			1
FMUL (vectors, unpredicated)				1							9	FL*			1
FMULX				1				✓			9	FL*			1
FNEG				1				✓			4	FL*			
FNMAD				1				✓	✓		9	FL*			2
FNMLA				1				✓	✓		9	FL*			2
FNMLS				1				✓	✓		9	FL*			2
FNMSB				1				✓	✓		9	FL*			2
FRECPE				1							4	FL*			
FRECPS				1							9	FLA			1
FRECPX				1				✓			4	FL*			
FRINTA				1				✓			9	FL*			
FRINTI				1				✓			9	FL*			
FRINTM				1				✓			9	FL*			
FRINTN				1				✓			9	FL*			
FRINTP				1				✓			9	FL*			
FRINTX				1				✓			9	FL*			
FRINTZ				1				✓			9	FL*			
FRSQRTE				1							4	FL*			
FRSQRTS				1							9	FLA			1
FSCALE				1				✓			9	FL*			1
FSQRT		<T> = H	128	1				✓		E	38	FLA			1
			256	1				✓		E	70	FLA			
			512	1				✓		E	134	FLA			
		<T> = S	128	1				✓		E	29	FLA			
			256	1				✓		E	52	FLA			
			512	1				✓		E	98	FLA			
		<T> = D	128	1				✓		E	43	FLA			
			256	1				✓		E	80	FLA			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
			512	1				✓		E	154	FLA			
FSUB (immediate)				1				✓			9	FLA			1
FSUB (vectors, predicated)				1				✓			9	FL*			1
FSUB (vectors, unpredicated)				1							9	FL*			1
FSUBR (immediate)				1				✓			9	FLA			1
FSUBR (vectors)				1				✓			9	FL*			1
FTMAD				1				✓			9	FL*			2
FTSMUL				1							9	FL*			1
FTSSEL				1							4	FL*			
INCB				1							1	EX*			
INCD (scalar)				1							1	EX*			
INCD (vector)				1				✓			4	FL*			
INCH (scalar)				1							1	EX*			
INCH (vector)				1				✓			4	FL*			
INCP (scalar)				2							3+2+1 / [1]1	PRX+NULL+EXA / EXB			
INCP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
INCW (scalar)				1							1	EX*			
INCW (vector)				1				✓			4	FL*			
INDEX (immediate, scalar)		<T> = {B|H}		2							1+3+4 / 1+3+[1]9	EXA+NULL+FLA / EXA+NULL+FLA			
		<T> = {S|D}		1							1+3+9	EXA + NULL + FLA			
INDEX (immediates)		<T> = {B|H}		2							4 / [1]9	FLA / FLA			
		<T> = {S|D}		1							9	FLA			
INDEX (scalar, immediate)		<T> = {B|H}		2							1+3+4 / 1+3+[1]9	EXA+NULL+FLA / EXA+NULL+FLA			
		<T> = {S|D}		1							1+3+9	EXA + NULL + FLA			
INDEX (scalars)		<T> = {B|H}		3							1+3+4 / 1+3+4 / [1,2]9	EXA+NULL+FLA / EXA+NULL+FLA / FLB			
		<T> = {S|D}		2							1+3+4 / [1]9	EXA+NULL+FLA / FLA			
INSR (scalar)				1				✓			1+3+6	EXA + NULL + FLA			
INSR (SIMD&FP scalar)				1				✓			6	FLA			
LASTA (scalar)				1							6+1+18	FLA + NULL + EAG*	1	1	
LASTA (SIMD&FP scalar)				1							6	FLA			
LASTB (scalar)				1							6+1+18	FLA + NULL + EAG*	1	1	
LASTB (SIMD&FP scalar)				1							6	FLA			
LD1B (scalar plus immediate)				1							11	EAG*	1		
LD1B (scalar plus scalar)				1							11	EAG*	1		
LD1B (scalar plus vector)		32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1B (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1D (scalar plus immediate)				1							11	EAG*	1		

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
LD1D (scalar plus scalar)				1							11	EAG*	1		
LD1D (scalar plus vector)				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA +Pipe((EAGA & EAGB), 2)	4	1	
LD1D (vector plus immediate)				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1H (scalar plus immediate)				1							11	EAG*	1		
LD1H (scalar plus scalar)				1							11	EAG*	1		
LD1H (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1H (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1RB				1							11	EAG*	1		
LD1RD				1							11	EAG*	1		
LD1RH				1							11	EAG*	1		
LD1RQB (scalar plus immediate)				1							11	EAG*	1		
LD1RQB (scalar plus scalar)				1							11	EAG*	1		
LD1RQD (scalar plus immediate)				1							11	EAG*	1		
LD1RQD (scalar plus scalar)				1							11	EAG*	1		
LD1RQH (scalar plus immediate)				1							11	EAG*	1		
LD1RQH (scalar plus scalar)				1							11	EAG*	1		
LD1RQW (scalar plus immediate)				1							11	EAG*	1		
LD1RQW (scalar plus scalar)				1							11	EAG*	1		
LD1RSB				1							11	EAG*	1		
LD1RSB (scalar plus scalar)				1							11	EAG*	1		
LD1RSH				1							11	EAG*	1		
LD1RSW				1							11	EAG*	1		
LD1RW				1							11	EAG*	1		
LD1SB (scalar plus immediate)				1							11	EAG*	1		
LD1SB (scalar plus scalar)				1							11	EAG*	1		
LD1SB (scalar plus vector)		32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1SB (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1SH (scalar plus immediate)				1							11	EAG*	1		
LD1SH (scalar plus scalar)				1							11	EAG*	1		
LD1SH (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1SH (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1SW (scalar plus immediate)				1							11	EAG*	1		
LD1SW (scalar plus scalar)				1							11	EAG*	1		

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
LD1SW (scalar plus vector)				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1SW (vector plus immediate)				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1W (scalar plus immediate)				1							11	EAG*	1		
LD1W (scalar plus scalar)				1							11	EAG*	1		
LD1W (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LD1W (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	1	1	
LD2B (scalar plus immediate)				3							1 / [1/2]((Pipe(11, 4)) x 2	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 2	8		
LD2B (scalar plus scalar)				3							1 / [1/2]((Pipe(11, 4)) x 2	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 2	8		
LD2D (scalar plus immediate)				2							11 / 11	EAG* / EAG*	2		
LD2D (scalar plus scalar)				3							1 / [1/2](11) x 2	EAG* / (EAG*) x 2	2		
LD2H (scalar plus immediate)				3							1 / [1/2]((Pipe(11, 4)) x 2	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 2	8		
LD2H (scalar plus scalar)				3							1 / [1/2]((Pipe(11, 4)) x 2	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 2	8		
LD2W (scalar plus immediate)				2							11 / 11	EAG* / EAG*	2		
LD2W (scalar plus scalar)				3							1 / [1/2](11) x 2	EAG* / (EAG*) x 2	2		
LD3B (scalar plus immediate)				4							1 / [1/2/3]((Pipe(11, 4)) x 3	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 3	12		
LD3B (scalar plus scalar)				4							1 / [1/2/3]((Pipe(11, 4)) x 3	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 3	12		
LD3D (scalar plus immediate)				3							11 / 11 / 11	EAG* / EAG* / EAG*	3		
LD3D (scalar plus scalar)				4							1 / [1/2/3](11) x 3	EAG* / (EAG*) x 3	3		
LD3H (scalar plus immediate)				4							1 / [1/2/3]((Pipe(11, 4)) x 3	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 3	12		
LD3H (scalar plus scalar)				4							1 / [1/2/3]((Pipe(11, 4)) x 3	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 3	12		
LD3W (scalar plus immediate)				3							11 / 11 / 11	EAG* / EAG* / EAG*	3		
LD3W (scalar plus scalar)				4							1 / [1/2/3](11) x 3	EAG* / (EAG*) x 3	3		
LD4B (scalar plus immediate)				5							1 / [1/2/3/4]((Pipe(11, 4)) x 4	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 4	16		
LD4B (scalar plus scalar)				5							1 / [1/2/3/4]((Pipe(11, 4)) x 4	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 4	16		
LD4D (scalar plus immediate)				4							11 / 11 / 11 / 11	EAG* / EAG* / EAG* / EAG*	4		
LD4D (scalar plus scalar)				5							1 / [1/2/3/4](11) x 4	EAG* / (EAG*) x 4	4		
LD4H (scalar plus immediate)				5							1 / [1/2/3/4]((Pipe(11, 4)) x 4	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 4	16		
LD4H (scalar plus scalar)				5							1 / [1/2/3/4]((Pipe(11, 4)) x 4	EAG* / (Pipe(EAGA, 4) | Pipe(EAGB, 4)) x 4	16		
LD4W (scalar plus immediate)				4							11 / 11 / 11 / 11	EAG* / EAG* / EAG* / EAG*	4		
LD4W (scalar plus scalar)				5							1 / [1/2/3/4](11) x 4	EAG* / (EAG*) x 4	4		
LDFF1B (scalar plus scalar)				1							11	EAG*	1		
LDFF1B (scalar plus vector)		32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1B (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1D (scalar plus scalar)				1							11	EAG*	1		

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
LDFF1D (scalar plus vector)				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1D (vector plus immediate)				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1H (scalar plus scalar)				1							11	EAG*	1		
LDFF1H (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1H (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SB (scalar plus scalar)				1							11	EAG*	1		
LDFF1SB (scalar plus vector)		32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SB (vector plus immediate)		32-bit element		1							1+1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SH (scalar plus scalar)				1							11	EAG*	1		
LDFF1SH (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SH (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SW (scalar plus scalar)				1							11	EAG*	1		
LDFF1SW (scalar plus vector)				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1SW (vector plus immediate)				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1W (scalar plus scalar)				1							11	EAG*	1		
LDFF1W (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		1							1+3+1+4+Pipe(11, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(11, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDFF1W (vector plus immediate)		32-bit element		1							1+4+Pipe(11, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(11, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
LDNF1B				1							11	EAG*	1		
LDNF1D				1							11	EAG*	1		
LDNF1H				1							11	EAG*	1		
LDNF1SB				1							11	EAG*	1		
LDNF1SH				1							11	EAG*	1		
LDNF1SW				1							11	EAG*	1		
LDNF1W				1							11	EAG*	1		
LDNT1B (scalar plus immediate)				1							11	EAG*	1		
LDNT1B (scalar plus scalar)				1							11	EAG*	1		
LDNT1D (scalar plus immediate)				1							11	EAG*	1		
LDNT1D (scalar plus scalar)				1							11	EAG*	1		
LDNT1H (scalar plus immediate)				1							11	EAG*	1		

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
LDNT1H (scalar plus scalar)				1							11	EAG*	1		
LDNT1W (scalar plus immediate)				1							11	EAG*	1		
LDNT1W (scalar plus scalar)				1							11	EAG*	1		
LDR (predicate)				1							11	EAGA	1		
LDR (vector)				1							11	EAGA	1		
LSL (immediate, predicated)				1				✓	✓		4	FL*			
LSL (immediate, unpredicated)				1							4	FL*			
LSL (vectors)				1				✓	✓		4	FL*			
LSL (wide elements, predicated)				1				✓	✓		4	FL*			
LSL (wide elements, unpredicated)				1							4	FL*			
LSLR				1				✓	✓		4	FL*			
LSR (immediate, predicated)				1				✓	✓		4	FL*			
LSR (immediate, unpredicated)				1							4	FL*			
LSR (vectors)				1				✓	✓		4	FL*			
LSR (wide elements, predicated)				1				✓	✓		4	FL*			
LSR (wide elements, unpredicated)				1							4	FL*			
LSRR				1				✓	✓		4	FL*			
MAD				1				✓	✓		9	FL*			
MLA				1				✓	✓		9	FL*			
MLS				1				✓	✓		9	FL*			
MOVPRFX (predicated)				1							4	FL*			
MOVPRFX (unpredicated)				1							4	FL*			
MSB				1				✓	✓		9	FL*			
MUL (immediate)				1				✓			9	FLA			
MUL (vectors)				1				✓			9	FL*			
NAND				1							3	PRX			
NANDS				1							3	PRX			
NEG				1				✓			4	FL*			
NOR				1							3	PRX			
NORS				1							3	PRX			
NOT (vector)				1				✓			4	FL*			
ORN (predicates)				1							3	PRX			
ORNS				1							3	PRX			
ORR (immediate)	ORN (immediate)			1				✓			4	FLA			
ORR (predicates)	MOV (predicate, unpredicated)			1							3	PRX			
				1							3	PRX			
ORR (vectors, predicated)				1				✓	✓		4	FL*			
ORR (vectors, unpredicated)	MOV (vector, unpredicated)			1							4	FL*			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
				1							4	FL*			
ORRS	MOVS (unpredicated)			1							3	PRX			
				1							3	PRX			
ORV		<V> = B		10	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
PFALSE				1							3	PRX			
PFIRST				1							3	PRX			
PNEXT				1							3	PRX			
PRFB (scalar plus immediate)				1							NA	EAG*	1		
PRFB (scalar plus scalar)				1							NA	EAG*	1		
PRFB (scalar plus vector)		32-bit scaled offset		1							1+3+1+4+Pipe(NA, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(NA, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFB (vector plus immediate)		32-bit element		1							1+4+Pipe(NA, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(NA, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFD (scalar plus immediate)				1							NA	EAG*	1		
PRFD (scalar plus scalar)				1							NA	EAG*	1		
PRFD (scalar plus vector)		32-bit scaled offset		1							1+3+1+4+Pipe(NA, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(NA, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFD (vector plus immediate)		32-bit element		1							1+4+Pipe(NA, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(NA, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFH (scalar plus immediate)				1							NA	EAG*	1		
PRFH (scalar plus scalar)				1							NA	EAG*	1		
PRFH (scalar plus vector)		32-bit scaled offset		1							1+3+1+4+Pipe(NA, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(NA, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFH (vector plus immediate)		32-bit element		1							1+4+Pipe(NA, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(NA, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFW (scalar plus immediate)				1							NA	EAG*	1		
PRFW (scalar plus scalar)				1							NA	EAG*	1		
PRFW (scalar plus vector)		32-bit scaled offset		1							1+3+1+4+Pipe(NA, 4)	EXA + NULL + FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							1+3+4+Pipe(NA, 2)	EXA + NULL + FLA + Pipe((EAGA & EAGB), 2)	4	1	
PRFW (vector plus immediate)		32-bit element		1							1+4+Pipe(NA, 4)	FLA + FLA + Pipe((EAGA & EAGB), 4)	8	1	
				1							4+Pipe(NA, 2)	FLA + Pipe((EAGA & EAGB), 2)	4	1	
PTEST				1							3	PRX			
PTRUE				1							3	PRX			
PTRUES				1							3	PRX			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
PUNPKHI				1							3	PRX			
PUNPKLO				1							3	PRX			
RBIT				1				✓			4	FL*			
RDFFR (predicated)				1							3	PRX			
RDFFR (unpredicated)				1							3	PRX			
RDFFRS				1							3	PRX			
RDVL				1							1	EX*			
REV (predicate)				1							3	PRX			
REV (vector)				1							6	FLA			
REVB				1				✓			4	FL*			
REVH				1				✓			4	FL*			
REVW				1				✓			4	FL*			
SABD				1				✓	✓		4	FL*			
SADDV		<T> = B		11	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4 / [1]4 / [1]4	FL* / FL* / (FL* / FLA) x 3 / FL* / FL* / FL*			
		<T> = H		10	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4 / [1]4	FL* / FL* / (FL* / FLA) x 3 / FL* / FL*			
		<T> = S	128	5	✓						4 / [1]4 / [1,2]4 / [1]6 / [1,2]4	FL* / FL* / FL* / FLA / FL*			
			256	7	✓						4 / [1]4 / ([1,2]4 / [1]6) x 2 / [1,2]4	FL* / FL* / (FL* / FLA) x 2 / FL*			
			512	9	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4	FL* / FL* / (FL* / FLA) x 3 / FL*			
SCVTF				1				✓			9	FL*			
SDIV		<T> = S	128	1				✓	✓	E	33	FLA			
			256	1				✓	✓	E	60	FLA			
			512	1				✓	✓	E	114	FLA			
		<T> = D	128	1				✓	✓	E	49	FLA			
			256	1				✓	✓	E	92	FLA			
			512	1				✓	✓	E	178	FLA			
SDIVR		<T> = S	128	1				✓	✓	E	33	FLA			
			256	1				✓	✓	E	60	FLA			
			512	1				✓	✓	E	114	FLA			
		<T> = D	128	1				✓	✓	E	49	FLA			
			256	1				✓	✓	E	92	FLA			
			512	1				✓	✓	E	178	FLA			
SDOT (indexed)				2							6 / [1]9	FLA / FLB			
SDOT (vectors)				1							9	FL*			
SEL (predicates)	MOV (predicate, predicated, merging)			1							3	PRX			
				1							3	PRX			
SEL (vectors)	MOV (vector, predicated)			1							4	FL*			
				1							4	FL*			
SETFFR				1							NA				
SMAX (immediate)				1				✓			4	FLA			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
SMAX (vectors)				1				✓	✓		4	FL*			
SMAXV		<V> = B		10	✓						4 / ([1])6 / ([1,2])4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1])6 / ([1,2])4) x 3 / (1)4 / (1)4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1])6 / ([1,2])4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1])6 / ([1,2])4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1])6 / ([1,2])4) x 3	FL* / (FLA / FL*) x 3			
SMIN (immediate)				1				✓			4	FLA			
SMIN (vectors)				1				✓	✓		4	FL*			
SMINV		<V> = B		10	✓						4 / ([1])6 / ([1,2])4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1])6 / ([1,2])4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1])6 / ([1,2])4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1])6 / ([1,2])4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1])6 / ([1,2])4) x 3	FL* / (FLA / FL*) x 3			
SMULH				1				✓			9	FL*			
SPLICE				1				✓			6	FLA			
SQADD (immediate)				1				✓			4	FL*			
SQADD (vectors)				1							4	FL*			
SQDECB				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQDECD (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQDECD (vector)				1				✓			4	FL*			
SQDECH (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQDECH (vector)				1				✓			4	FL*			
SQDECP (scalar)				2						/ P	3+2+1 / 1+[1]1	PRX + NULL + EXA / EXB + EXB			
SQDECP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
SQDECW (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQDECW (vector)				1				✓			4	FL*			
SQINCB				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQINCD (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQINCD (vector)				1				✓			4	FL*			
SQINCH (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQINCH (vector)				1				✓			4	FL*			
SQINCP (scalar)				2						/ P	3+2+1 / 1+[1]1	PRX + NULL + EXA / EXB + EXB			
SQINCP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
SQINCW (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
SQINCW (vector)				1				✓			4	FL*			
SQSUB (immediate)				1				✓			4	FL*			
SQSUB (vectors)				1							4	FL*			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
ST1B (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
ST1B (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
ST1B (scalar plus vector)		32-bit unscaled offset		8							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 4	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 4	16	16	
				4							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 2	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 2	8	8	
ST1B (vector plus immediate)		32-bit element		8							(4+Pipe(NA, 2) / NA) x 4	(FLA + FLA + Pipe((EAGA & EAGB), 2) / FLA) x 4	16	16	
				4							(4+Pipe(NA, 2) / NA) x 2	(FLA + Pipe((EAGA & EAGB), 2) / FLA) x 2	8	8	
ST1D (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
ST1D (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
ST1D (scalar plus vector)				4							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 2	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 2	8	8	
ST1D (vector plus immediate)				4							(4+Pipe(NA, 2) / NA) x 2	(FLA + Pipe((EAGA & EAGB), 2) / FLA) x 2	8	8	
ST1H (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
ST1H (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
ST1H (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		8							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 4	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 4	16	16	
				4							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 2	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 2	8	8	
ST1H (vector plus immediate)		32-bit element		8							(4+Pipe(NA, 2) / NA) x 4	(FLA + FLA + Pipe((EAGA & EAGB), 2) / FLA) x 4	16	16	
				4							(4+Pipe(NA, 2) / NA) x 2	(FLA + Pipe((EAGA & EAGB), 2) / FLA) x 2	8	8	
ST1W (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
ST1W (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
ST1W (scalar plus vector)		32-bit scaled offset, 32-bit unscaled offset		8							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 4	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 4	16	16	
				4							(1+3+4+Pipe(NA, 2) / 1+3+NA) x 2	(EXA + NULL + FLA + Pipe((EAGA & EAGB), 2) / EXA + NULL + FLA) x 2	8	8	
ST1W (vector plus immediate)		32-bit element		8							(4+Pipe(NA, 2) / NA) x 4	(FLA + Pipe((EAGA & EAGB), 2) / FLA) x 4	16	16	
				4							(4+Pipe(NA, 2) / NA) x 2	(FLA + Pipe((EAGA & EAGB), 2) / FLA) x 2	8	8	
ST2B (scalar plus immediate)				3							1 / [1/2](Pipe(NA, 4), Pipe(NA, 4)) x 2	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 2	8	8	
ST2B (scalar plus scalar)				3							1 / [1/2](Pipe(NA, 4), Pipe(NA, 4)) x 2	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 2	8	8	
ST2D (scalar plus immediate)				2							NA,NA / NA,NA	EAG*, FLA / EAG*, FLA	2	2	
ST2D (scalar plus scalar)				3							1 / [1/2](NA,NA) x 2	EAG* / (EAG*, FLA) x 2	2	2	
ST2H (scalar plus immediate)				3							1 / [1/2](Pipe(NA, 4), Pipe(NA, 4)) x 2	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 2	8	8	
ST2H (scalar plus scalar)				3							1 / [1/2](Pipe(NA, 4), Pipe(NA, 4)) x 2	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 2	8	8	
ST2W (scalar plus immediate)				2							NA,NA / NA,NA	EAG*, FLA / EAG*, FLA	2	2	
ST2W (scalar plus scalar)				3							1 / [1/2](NA,NA) x 2	EAG* / (EAG*, FLA) x 2	2	2	
ST3B (scalar plus immediate)				4							1 / [1/2/3](Pipe(NA, 4), Pipe(NA, 4)) x 3	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 3	12	12	
ST3B (scalar plus scalar)				4							1 / [1/2/3](Pipe(NA, 4), Pipe(NA, 4)) x 3	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 3	12	12	
ST3D (scalar plus immediate)				3							NA,NA / NA,NA / NA,NA	(EAG*, FLA) x 3	3	3	
ST3D (scalar plus scalar)				4							1 / [1/2/3](NA,NA) x 3	EAG* / (EAG*, FLA) x 3	3	3	
ST3H (scalar plus immediate)				4							1 / [1/2/3](Pipe(NA, 4), Pipe(NA, 4)) x 3	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 3	12	12	
ST3H (scalar plus scalar)				4							1 / [1/2/3](Pipe(NA, 4), Pipe(NA, 4)) x 3	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 3	12	12	

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
ST3W (scalar plus immediate)				3							NA,NA / NA,NA / NA,NA	EAG*, FLA / EAG*, FLA / EAG*, FLA	3	3	
ST3W (scalar plus scalar)				4							1 / [1/2/3](NA,NA) x 3	EAG* / (EAG*, FLA) x 3	3	3	
ST4B (scalar plus immediate)				5							1 / [1/2/3/4](Pipe(NA, 4), Pipe(NA, 4)) x 4	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 4	16	16	
ST4B (scalar plus scalar)				5							1 / [1/2/3/4](Pipe(NA, 4), Pipe(NA, 4)) x 4	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 4	16	16	
ST4D (scalar plus immediate)				4							NA,NA / NA,NA / NA,NA / NA,NA	EAG*, FLA / EAG*, FLA / EAG*, FLA / EAG*, FLA	4	4	
ST4D (scalar plus scalar)				5							1 / [1/2/3/4](NA,NA) x 4	EAG* / (EAG*, FLA) x 4	4	4	
ST4H (scalar plus immediate)				5							1 / [1/2/3/4](Pipe(NA, 4), Pipe(NA, 4)) x 4	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 4	16	16	
ST4H (scalar plus scalar)				5							1 / [1/2/3/4](Pipe(NA, 4), Pipe(NA, 4)) x 4	EAG* / ((Pipe(EAGA, 4) | Pipe(EAGB, 4)), Pipe(FLA, 4)) x 4	16	16	
ST4W (scalar plus immediate)				4							NA,NA / NA,NA / NA,NA / NA,NA	EAG*, FLA / EAG*, FLA / EAG*, FLA / EAG*, FLA	4	4	
ST4W (scalar plus scalar)				5							1 / [1/2/3/4](NA,NA) x 4	EAG* / (EAG*, FLA) x 4	4	4	
STNT1B (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
STNT1B (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
STNT1D (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
STNT1D (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
STNT1H (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
STNT1H (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
STNT1W (scalar plus immediate)				1							NA, NA	EAG*, FLA	1	1	
STNT1W (scalar plus scalar)				1							NA, NA	EAG*, FLA	1	1	
STR (predicate)				1							NA, NA	EAGA, PRX	1	1	
STR (vector)				1							NA, NA	EAGA, FLA	1	1	
SUB (immediate)				1				✓			4	FL*			
SUB (vectors, predicated)				1				✓	✓		4	FL*			
SUB (vectors, unpredicated)				1							4	FL*			
SUBR (immediate)				1				✓			4	FLA			
SUBR (vectors)				1				✓	✓		4	FL*			
SUNPKHI				1							6	FLA			
SUNPKLO				1							6	FLA			
SXTB				1				✓			4	FL*			
SXTH				1				✓			4	FL*			
SXTW				1				✓			4	FL*			
TBL				1							6	FLA			
TRN1 (predicates)				1							3	PRX			
TRN1 (vectors)				1							6	FLA			
TRN2 (predicates)				1							3	PRX			
TRN2 (vectors)				1							6	FLA			
UABD				1				✓	✓		4	FL*			
UADDV		<T> = B		11	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4 / [1]4 / [1]4	FL* / FL* / (FL* / FLA) x 3 / FL* / FL* / FL*			
		<T> = H		10	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4 / [1]4	FL* / FL* / (FL* / FLA) x 3 / FL* / FL*			
		<T> = {S|D}	128	5	✓						4 / [1]4 / [1,2]4 / [1]6 / [1,2]4	FL* / FL* / FL* / FLA / FL*			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
			256	7	✓						4 / [1]4 / ([1,2]4 / [1]6) x 2 / [1,2]4	FL* / FL* / (FL* / FLA) x 2 / FL*			
			512	9	✓						4 / [1]4 / ([1,2]4 / [1]6) x 3 / [1,2]4	FL* / FL* / (FL* / FLA) x 3 / FL*			
UCVTF				1				✓			9	FL*			
UDIV		<T> = S	128	1				✓	✓	E	33	FLA			
			256	1				✓	✓	E	60	FLA			
			512	1				✓	✓	E	114	FLA			
		<T> = D	128	1				✓	✓	E	49	FLA			
			256	1				✓	✓	E	92	FLA			
			512	1				✓	✓	E	178	FLA			
UDIVR		<T> = S	128	1				✓	✓	E	33	FLA			
			256	1				✓	✓	E	60	FLA			
			512	1				✓	✓	E	114	FLA			
		<T> = D	128	1				✓	✓	E	49	FLA			
			256	1				✓	✓	E	92	FLA			
			512	1				✓	✓	E	178	FLA			
UDOT (indexed)				2							6 / [1]9	FLA / FLB			
UDOT (vectors)				1							9	FL*			
UMAX (immediate)				1				✓			4	FLA			
UMAX (vectors)				1				✓	✓		4	FL*			
UMAXV		<V> = B		10	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
UMIN (immediate)				1				✓			4	FLA			
UMIN (vectors)				1				✓	✓		4	FL*			
UMINV		<V> = B		10	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL* / FL*			
		<V> = H		9	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4 / [1]4	FL* / (FLA / FL*) x 3 / FL* / FL*			
		<V> = S		8	✓						4 / ([1]6 / [1,2]4) x 3 / [1]4	FL* / (FLA / FL*) x 3 / FL*			
		<V> = D	128	3	✓						4 / [1]6 / [1,2]4	FL* / FLA / FL*			
			256	5	✓						4 / ([1]6 / [1,2]4) x 2	FL* / (FLA / FL*) x 2			
			512	7	✓						4 / ([1]6 / [1,2]4) x 3	FL* / (FLA / FL*) x 3			
UMULH				1				✓			9	FL*			
UQADD (immediate)				1				✓			4	FL*			
UQADD (vectors)				1							4	FL*			
UQDECB				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQDECD (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			

Instruction	Alias	Control option	VL	# of µOP	Seq. decode	Pre-sync	Post-sync	Pack	Extra µOP	Blocking	Latency	Pipeline	# of FP	# of SP	FLOPS
UQDECD (vector)				1				✓			4	FL*			
UQDECH (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQDECH (vector)				1				✓			4	FL*			
UQDECP (scalar)				2						/ P	3+2+1 / 1+[1]1	PRX + NULL + EXA / EXB + EXB			
UQDECP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
UQDECW (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQDECW (vector)				1				✓			4	FL*			
UQINCB				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQINCD (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQINCD (vector)				1				✓			4	FL*			
UQINCH (scalar)				1						P	1+1	EX* + EX*			
UQINCH (vector)				1				✓			4	FL*			
UQINCP (scalar)				2						/ P	3+2+1 / 1+[1]1	PRX + NULL + EXA / EXB + EXB			
UQINCP (vector)				1				✓			3+5+4	PRX + NULL + FLA			
UQINCW (scalar)				1						P	1+1	(EXA + EXA) | (EXB + EXB)			
UQINCW (vector)				1				✓			4	FL*			
UQSUB (immediate)				1				✓			4	FL*			
UQSUB (vectors)				1							4	FL*			
UUNPKHI				1							6	FLA			
UUNPKLO				1							6	FLA			
UXTB				1				✓			4	FL*			
UXTH				1				✓			4	FL*			
UXTW				1				✓			4	FL*			
UZP1 (predicates)				1							3	PRX			
UZP1 (vectors)				1							6	FLA			
UZP2 (predicates)				1							3	PRX			
UZP2 (vectors)				1							6	FLA			
WHILELE				1							1+3	EXA + PRX			
WHILELO				1							1+3	EXA + PRX			
WHILELS				1							1+3	EXA + PRX			
WHILELT				1							1+3	EXA + PRX			
WRFFR				2	✓	✓	✓				NA / 3	/ PRX			
ZIP1 (predicates)				1							3	PRX			
ZIP1 (vectors)				1							6	FLA			
ZIP2 (predicates)				1							3	PRX			
ZIP2 (vectors)				1							6	FLA			