
GRAPHCORE

Tile Vertex ISA

Release 1.2.3

IPU2

Mar 30, 2022

TABLE OF CONTENTS

1	Introduction	3
1.1	Scope	3
1.2	Revision history	3
1.2.1	Release 1.2.3	3
1.2.2	Implementation variant	4
1.3	Conventions	4
1.3.1	Arithmetic notation and operators	4
1.4	Glossary	4
2	Machine Framework	7
2.1	Overview	7
2.2	Logical Block structure	8
2.3	Hardware Contexts	8
2.4	Execution Pipelines	9
2.5	Program Order	9
2.5.1	Co-issue	10
2.6	Execution semantics	12
2.6.1	Register read values	12
2.6.2	Run Modes	12
2.6.3	Quiescence	13
2.7	Register model	14
2.7.1	Register access semantics	14
2.7.2	Context register state	14
2.7.3	MRF	14
2.7.3.1	MRF Read and write ports	17
2.7.4	ARF	17
2.7.4.1	ARF Read and write ports	18
2.7.5	Control and Status registers	19
2.7.5.1	Supervisor CSRs	19
2.7.5.1.1	\$FP_CTL	19
2.7.5.1.2	\$CCCSLOAD	20
2.7.5.1.3	\$CTXT_STS	21
2.7.5.2	Worker CSRs	21
2.7.5.2.1	\$PC	22
2.7.5.2.2	\$WSR	22
2.7.5.2.3	\$VERTEX_BASE	23
2.7.5.2.4	\$WORKER_BASE	23
2.7.5.2.5	\$REPEAT_COUNT	24
2.7.5.2.6	\$REPEAT_FIRST	24
2.7.5.2.7	\$REPEAT_END	25
2.7.5.2.8	\$COUNT_L	25
2.7.5.2.9	\$COUNT_U	25
2.7.5.2.10	\$DBG_DATA	26
2.7.5.2.11	\$DBG_BRK_ID	26
2.7.5.2.12	\$FP_STS	27
2.7.5.2.13	\$FP_CLR	27
2.7.5.2.14	\$FP_CTL	27

	2.7.5.2.15	\$PRNG_0_0	28
	2.7.5.2.16	\$PRNG_0_1	29
	2.7.5.2.17	\$PRNG_1_0	29
	2.7.5.2.18	\$PRNG_1_1	30
	2.7.5.2.19	\$PRNG_SEED	30
	2.7.5.2.20	\$TAS	30
2.7.6		Pipeline internal state	31
	2.7.6.1	aux	31
2.7.7		Common Compute Configuration State	32
	2.7.7.1	Common Compute Configuration Space	32
	2.7.7.1.1	\$CWEL_n_0	33
	2.7.7.1.2	\$CWEL_n_1	34
	2.7.7.1.3	\$CWEL_n_2	34
	2.7.7.1.4	\$CWEL_n_3	35
2.8		Memory model	36
	2.8.1	Overview	36
	2.8.2	Address format	36
	2.8.2.1	Pointers	37
	2.8.2.1.1	Full	37
	2.8.2.1.2	Packed	37
	2.8.2.2	Delta Offsets	37
	2.8.2.3	Mini deltas	37
	2.8.3	Endianness	37
	2.8.4	Memory regions	37
	2.8.5	Memory Map	38
	2.8.6	Data access sizes	38
	2.8.7	Buffering	38
	2.8.8	Memory Protection	39
	2.8.9	Memory access semantics	39
	2.8.9.1	Instruction fetch semantics	39
	2.8.9.2	Data load/store semantics	40
	2.8.10	Memory system errors	41
	2.8.10.1	Uncorrectable Errors	41
	2.8.11	Memory clashes	41
	2.8.11.1	Instruction stream based	41
	2.8.11.2	Exchange based	42
2.9		Floating Point Unit	43
	2.9.1	Overview	43
	2.9.2	Number formats	43
	2.9.2.1	Scalars	43
	2.9.2.1.1	Single Precision Constants	43
	2.9.2.1.2	Half Precision Class	44
	2.9.2.2	Vectors	46
	2.9.3	Control and Status Registers	47
	2.9.4	General Accuracy	48
	2.9.4.1	Single-precision	48
	2.9.4.2	Half-precision	48
	2.9.5	Rounding modes	48
	2.9.6	Format Conversion and Transformations	49
	2.9.7	Floating point exceptions	51
	2.9.7.1	Exception conditions	52
	2.9.7.1.1	Invalid operation	52
	2.9.7.1.2	Divide-by-zero	52
	2.9.7.1.3	Overflow	52
	2.9.7.1.4	Underflow	52
	2.9.7.1.5	Inexact result	52
	2.9.8	Comparisons	52
	2.9.9	Accumulation	53
	2.9.9.1	Single-precision multiply-accumulate	53
	2.9.9.2	Half-precision multiply-accumulate	53
	2.9.10	Dot-products	53

2.9.10.1	Half-precision vector dot-products	53
2.9.11	AMP and SLIC instructions	54
2.9.12	Transcendental functions	56
2.9.13	IEEE 754-2008 clarifications	57
2.9.13.1	NaN generation	57
2.9.14	IEEE 754-2008 caveats and differences	57
2.9.14.1	Transcendentals	57
2.10	Exception model	58
2.10.1	Exception types	58
2.10.2	Exception Events	58
2.11	Debug	59
2.12	Exchange interface	60
2.13	Pseudorandom number generator	61
2.13.1	State	61
2.13.2	Random Number Generation	61
2.13.2.1	Quality	62
2.13.3	Discrete Uniform Distribution	64
2.13.3.1	Integer Instructions	64
2.13.3.2	Floating-point conversion instructions	64
2.13.4	Irwin-Hall Distribution	64
2.13.4.1	Properties	64
2.13.4.2	Instructions	65
2.13.5	Element Masking	65
2.13.5.1	Instructions	65
2.13.6	Stochastic Rounding	65
3	Instructions	69
3.1	Instruction signatures	69
3.2	Register specifiers	71
3.3	Types of immediate	72
3.4	Memory addressing modes	72
3.4.1	Base address with delta and scaled unsigned register offset	72
3.4.2	Base address with delta and scaled zero-extended immediate offset	73
3.4.3	Post-incrementing absolute address	73
3.4.4	Base address with post-incrementing delta and signed register stride	73
3.4.5	Base address with post-incrementing delta and signed immediate stride	73
3.4.6	Base address with upper 16-bit delta	73
3.4.7	Base address with lower 16-bit delta, with simultaneous delta load from absolute, post-incrementing address	74
3.4.8	Base address with post-incrementing delta-pair	74
3.4.9	Post-incrementing packed absolute addresses (with packed strides)	74
3.5	Mnemonic conventions	75
3.5.1	Load/store instructions	75
3.5.2	Floating-point instructions	75
3.6	Instruction execution semantics	76
3.7	Instructions by class	82
3.7.1	Bit	83
3.7.1.1	and	83
3.7.1.2	and64	84
3.7.1.3	andc	84
3.7.1.4	andc64	85
3.7.1.5	bitrev8	85
3.7.1.6	clz	86
3.7.1.7	cms	86
3.7.1.8	not	87
3.7.1.9	not64	87
3.7.1.10	or	87
3.7.1.11	or64	88
3.7.1.12	popc	88
3.7.1.13	roll16	88
3.7.1.14	roll32	89

3.7.1.15	roll8l	90
3.7.1.16	roll8r	90
3.7.1.17	setzi	91
3.7.1.18	shuf8x8hi	91
3.7.1.19	shuf8x8lo	92
3.7.1.20	sort4x16hi	93
3.7.1.21	sort4x16lo	94
3.7.1.22	sort4x32hi	94
3.7.1.23	sort4x32lo	95
3.7.1.24	sort8	95
3.7.1.25	sort8x8hi	96
3.7.1.26	sort8x8lo	97
3.7.1.27	swap8	98
3.7.1.28	xnor	99
3.7.1.29	xor	99
3.7.2	Control	100
3.7.2.1	br	100
3.7.2.2	bri	101
3.7.2.3	brneg	101
3.7.2.4	brnz	102
3.7.2.5	brnzdec	102
3.7.2.6	brpos	103
3.7.2.7	brz	104
3.7.2.8	call	105
3.7.2.9	exitneg	105
3.7.2.10	exitnz	106
3.7.2.11	exitpos	106
3.7.2.12	exitz	107
3.7.2.13	rpt	107
3.7.3	Float	110
3.7.3.1	Format conversion	110
3.7.3.1.1	f16tof32	110
3.7.3.1.2	f16v2sufromui	110
3.7.3.1.3	f16v2tof32	111
3.7.3.1.4	f16v4sufromui	112
3.7.3.1.5	f32fromi32	113
3.7.3.1.6	f32fromui32	113
3.7.3.1.7	f32sufromui	114
3.7.3.1.8	f32tof16	114
3.7.3.1.9	f32toi32	115
3.7.3.1.10	f32toui32	116
3.7.3.1.11	f32v2sufromui	117
3.7.3.1.12	f32v2tof16	118
3.7.3.1.13	f32v4tof16	119
3.7.3.2	f16 2-element vector	121
3.7.3.2.1	f16v2absadd	121
3.7.3.2.2	f16v2absmax	123
3.7.3.2.3	f16v2add	123
3.7.3.2.4	f16v2clamp	125
3.7.3.2.5	f16v2class	126
3.7.3.2.6	f16v2cmac	126
3.7.3.2.7	f16v2cmpeq	127
3.7.3.2.8	f16v2cmpge	128
3.7.3.2.9	f16v2cmpgt	129
3.7.3.2.10	f16v2cmple	130
3.7.3.2.11	f16v2cmpplt	131
3.7.3.2.12	f16v2cmpne	132
3.7.3.2.13	f16v2exp	134
3.7.3.2.14	f16v2exp2	135
3.7.3.2.15	f16v2gina	135
3.7.3.2.16	f16v2grand	138

3.7.3.2.17	f16v2ln	139
3.7.3.2.18	f16v2log2	140
3.7.3.2.19	f16v2max	140
3.7.3.2.20	f16v2maxc	141
3.7.3.2.21	f16v2min	142
3.7.3.2.22	f16v2mul	143
3.7.3.2.23	f16v2sigm	145
3.7.3.2.24	f16v2sub	145
3.7.3.2.25	f16v2sum	147
3.7.3.2.26	f16v2tanh	148
3.7.3.3	f16 4-element vector	148
3.7.3.3.1	f16v4absacc	148
3.7.3.3.2	f16v4absadd	149
3.7.3.3.3	f16v4absmax	151
3.7.3.3.4	f16v4acc	152
3.7.3.3.5	f16v4add	152
3.7.3.3.6	f16v4clamp	154
3.7.3.3.7	f16v4class	155
3.7.3.3.8	f16v4cmac	156
3.7.3.3.9	f16v4cmpeq	158
3.7.3.3.10	f16v4cmpge	159
3.7.3.3.11	f16v4cmpgt	160
3.7.3.3.12	f16v4cmple	161
3.7.3.3.13	f16v4cmplt	162
3.7.3.3.14	f16v4cmpne	163
3.7.3.3.15	f16v4gacc	164
3.7.3.3.16	f16v4hihoamp	165
3.7.3.3.17	f16v4hihoslic	170
3.7.3.3.18	f16v4hihov4amp	174
3.7.3.3.19	f16v4hihov4slic	177
3.7.3.3.20	f16v4istacc	180
3.7.3.3.21	f16v4max	181
3.7.3.3.22	f16v4maxc	182
3.7.3.3.23	f16v4min	183
3.7.3.3.24	f16v4mix	184
3.7.3.3.25	f16v4mul	186
3.7.3.3.26	f16v4rmask	188
3.7.3.3.27	f16v4sihoamp	189
3.7.3.3.28	f16v4sihoslic	192
3.7.3.3.29	f16v4sisoamp	194
3.7.3.3.30	f16v4sisoslic	200
3.7.3.3.31	f16v4stacc	206
3.7.3.3.32	f16v4sub	208
3.7.3.3.33	f16v4sum	210
3.7.3.4	f16 8-element vector	211
3.7.3.4.1	f16v8absacc	211
3.7.3.4.2	f16v8acc	211
3.7.3.4.3	f16v8sqacc	212
3.7.3.5	f32 2-element vector	213
3.7.3.5.1	f32v2absadd	213
3.7.3.5.2	f32v2absmax	214
3.7.3.5.3	f32v2add	215
3.7.3.5.4	f32v2aop	216
3.7.3.5.5	f32v2axpy	218
3.7.3.5.6	f32v2clamp	219
3.7.3.5.7	f32v2class	220
3.7.3.5.8	f32v2cmpeq	221
3.7.3.5.9	f32v2cmpge	222
3.7.3.5.10	f32v2cmpgt	223
3.7.3.5.11	f32v2cmple	224
3.7.3.5.12	f32v2cmplt	225

3.7.3.5.13	f32v2cmpne	226
3.7.3.5.14	f32v2gina	227
3.7.3.5.15	f32v2grand	230
3.7.3.5.16	f32v2mac	231
3.7.3.5.17	f32v2max	231
3.7.3.5.18	f32v2min	232
3.7.3.5.19	f32v2mul	233
3.7.3.5.20	f32v2rmask	234
3.7.3.5.21	f32v2sub	235
3.7.3.6	f32 4-element vector	236
3.7.3.6.1	f32v4absacc	236
3.7.3.6.2	f32v4acc	237
3.7.3.6.3	f32v4sqacc	238
3.7.3.7	f32 scalar	239
3.7.3.7.1	f32absadd	239
3.7.3.7.2	f32absmax	240
3.7.3.7.3	f32add	240
3.7.3.7.4	f32clamp	241
3.7.3.7.5	f32class	242
3.7.3.7.6	f32cmpeq	243
3.7.3.7.7	f32cmpge	244
3.7.3.7.8	f32cmpgt	244
3.7.3.7.9	f32cmple	245
3.7.3.7.10	f32cmplt	246
3.7.3.7.11	f32cmpne	246
3.7.3.7.12	f32div	247
3.7.3.7.13	f32exp	248
3.7.3.7.14	f32exp2	249
3.7.3.7.15	f32int	249
3.7.3.7.16	f32ln	250
3.7.3.7.17	f32log2	250
3.7.3.7.18	f32mac	251
3.7.3.7.19	f32max	252
3.7.3.7.20	f32min	252
3.7.3.7.21	f32mul	253
3.7.3.7.22	f32oorx	253
3.7.3.7.23	f32oox	254
3.7.3.7.24	f32sigm	254
3.7.3.7.25	f32sisoamp	255
3.7.3.7.26	f32sisoslic	260
3.7.3.7.27	f32sisov2amp	265
3.7.3.7.28	f32sisov2slic	268
3.7.3.7.29	f32sqrt	271
3.7.3.7.30	f32sub	271
3.7.3.7.31	f32tanh	272
3.7.4	Integer	273
3.7.4.1	abs	273
3.7.4.2	add	273
3.7.4.3	cmpeq	274
3.7.4.4	cmpne	274
3.7.4.5	cmpslt	275
3.7.4.6	cmpult	275
3.7.4.7	max	275
3.7.4.8	min	276
3.7.4.9	movz	276
3.7.4.10	mul	277
3.7.4.11	shl	277
3.7.4.12	shr	277
3.7.4.13	shrs	277
3.7.4.14	sub	278
3.7.4.15	tapack	278

3.7.4.16	urand32	279
3.7.4.17	urand64	279
3.7.5	Memory	280
3.7.5.1	Load-store	281
3.7.5.1.1	ldst64pace	281
3.7.5.1.2	ld2xst64pace	283
3.7.5.2	Multi-load	285
3.7.5.2.1	ldd16b16	285
3.7.5.2.2	ldd16a32	287
3.7.5.2.3	ldd16v2a32	289
3.7.5.2.4	ldd16a64	291
3.7.5.2.5	ldb16b16	293
3.7.5.2.6	ld64a32	296
3.7.5.2.7	ld64b16pace	298
3.7.5.2.8	ld64a32pace	300
3.7.5.2.9	ld2x64pace	302
3.7.5.3	Single-load	305
3.7.5.3.1	atom	305
3.7.5.3.2	ldz8	305
3.7.5.3.3	lds8	306
3.7.5.3.4	ldz16	307
3.7.5.3.5	lds16	308
3.7.5.3.6	ldb16	309
3.7.5.3.7	ld32	310
3.7.5.3.8	ld64	311
3.7.5.3.9	ld128	313
3.7.5.3.10	ldz8step	314
3.7.5.3.11	lds8step	315
3.7.5.3.12	ldz16step	316
3.7.5.3.13	lds16step	317
3.7.5.3.14	ldb16step	318
3.7.5.3.15	ld32step	319
3.7.5.3.16	ld64step	320
3.7.5.3.17	ld128step	321
3.7.5.3.18	ld64putcs	322
3.7.5.3.19	ld128putcs	323
3.7.5.4	Single-store	324
3.7.5.4.1	st32	324
3.7.5.4.2	stm32	325
3.7.5.4.3	st64	326
3.7.5.4.4	st64pace	327
3.7.5.4.5	st32step	328
3.7.5.4.6	stm32step	329
3.7.5.4.7	st64step	330
3.7.6	System	332
3.7.6.1	get	332
3.7.6.2	put	332
3.7.6.3	run	333
3.7.6.4	runall	334
3.7.6.5	trap	336
3.7.6.6	uget	336
3.7.6.7	uput	336
3.8	Instructions by attribute	338
3.8.1	Full instruction list per pipeline	338
3.8.1.1	main	338
3.8.1.2	aux	338
3.8.1.3	both	339
3.8.2	Instructions by FP exception	339
3.8.2.1	TFPEXCPT_INV	339
3.8.2.2	TFPEXCPT_DIV0	340
3.8.2.3	TFPEXCPT_OFLO	340

3.8.3	Instructions with broadcast	340
3.8.4	Floating-point operations x number format and vector length	341
4	Implementation Specifics	343
4.1	[IPU2]	343
4.1.1	General	344
4.1.1.1	Tile_ZeroExtend	344
4.1.1.2	Tile_SignExtend	344
4.1.1.3	Tile_ExtractPackedStride	344
4.1.1.4	Tile_ExtractPackedAddress	344
4.1.1.5	Tile_TripleAddressPack_Lower	344
4.1.1.6	Tile_TripleAddressPack_Upper	345
4.1.2	Instructions	346
4.1.2.1	TInstr_GetLatency	346
4.1.3	Floating_Point	347
4.1.3.1	Transcendental, divide, square-root and reciprocal instructions	347
4.1.3.2	Parameters	347
4.1.3.3	TileRoundMode	347
4.1.3.4	TileFPAccuracy	348
4.1.3.5	TFPU_F32FromBits	348
4.1.3.6	TFPU_BitsFromF32	348
4.1.3.7	TFPU_F32_QNan	349
4.1.3.8	TFPU_F32_IsQNan	349
4.1.3.9	TFPU_F32_IsSNan	349
4.1.3.10	TFPU_RoundFP64ToFmt	349
4.1.3.11	TFPU_RoundFP32ToIntegral	350
4.1.3.12	TFPU_F32_QuietenNan	351
4.1.3.13	TFPU_SNanCheck	351
4.1.3.14	TFPU_GenSNanCheck	351
4.1.3.15	TFPU_GenOFLOCheck	351
4.1.3.16	TFPU_AACCRresetValue	352
4.1.3.17	TFPU_AACCRreadFlags	352
4.1.3.18	TFPU_F16DotProduct	353
4.1.3.19	TFPU_DoAddPreExecute	353
4.1.3.20	TFPU_Add	354
4.1.3.21	TFPU_DoAxpbyPreExecute	354
4.1.3.22	TFPU_DoMacPreExecute	355
4.1.3.23	TFPU_Mac	356
4.1.3.24	TFPU_DoMulPreExecute	357
4.1.3.25	TFPU_Mul	357
4.1.3.26	TFPU_DoDivPreExecute	358
4.1.3.27	TFPU_DoSqrtPreExecute	359
4.1.3.28	TFPU_DoRecipPreExecute	360
4.1.3.29	TFPU_DoRSqrtPreExecute	360
4.1.3.30	TFPU_DoExpPreExecute	360
4.1.3.31	TFPU_DoLogPreExecute	361
4.1.3.32	TFPU_DoTanhPreExecute	361
4.1.3.33	TFPU_DoSigmoidPreExecute	362
4.1.3.34	TFPU_F32Sigmoid	362
4.1.3.35	TFPU_F16Sigmoid	362
4.1.3.36	TFPU_F32Exp	363
4.1.3.37	TFPU_F16Exp	363
4.1.3.38	TFPU_F32Log	363
4.1.3.39	TFPU_F16Log	363
4.1.3.40	TFPU_F32Tanh	363
4.1.3.41	TFPU_F16Tanh	363
4.1.3.42	TFPU_F32Recip	363
4.1.3.43	TFPU_F32Sqrt	364
4.1.3.44	TFPU_F32RSqrt	364
4.1.3.45	TFPU_Min	364
4.1.3.46	TFPU_Max	365

4.1.3.47	TFPU_Relation	365
4.1.3.48	TFPU_F32DivExceptIsImprecise	366
4.1.3.49	TFPU_IsMalign	366
4.1.4	Exceptions	367
4.1.4.1	Imprecise Exceptions	367
4.1.4.2	Super-imprecise Exceptions	367
4.1.4.3	RBRK and imprecise-exceptions	367
4.1.4.4	TileException	367
4.1.5	Contexts	368
4.1.5.1	TileCtxtStatus	368
4.1.6	Memory	369
4.1.6.1	Memory regions	369
4.1.6.2	Memory clashes	369
4.1.6.3	Striding support	369
4.1.6.4	Parameters	370
4.1.6.5	TMem_RegionId	370
4.1.6.6	TMem_ElementId	370
4.1.6.7	TMem_ElementOffset	371
4.1.6.8	TMem_IsValidAddress	371
4.1.6.9	TMem_RegionBaseAddress	371
4.1.6.10	TMem_RegionInterleaveFactor	371
4.1.6.11	TMem_AddressInterleaveFactor	372
4.1.6.12	TMem_RegionIsExecutable	372
4.1.6.13	TMem_AddressIsExecutable	372
4.1.7	Registers	373
4.1.7.1	TileRFAccessSize	373
4.1.7.2	TReg_RFIndices	373
4.1.7.3	TReg_IsValidCCCS	373
4.1.7.4	TReg_WriteException	374
4.1.7.5	TReg_IsValidCSR	374
4.1.7.6	MRF	374
4.1.7.6.1	Parameters	374
4.1.7.6.2	TReg_MRFRresetValue	375
4.1.7.7	ARF	375
4.1.7.7.1	Parameters	375
4.1.7.7.2	TReg_ARFRresetValue	375

Bibliography	377
---------------------	------------

Index	379
--------------	------------

INTRODUCTION

1.1 Scope

This document describes the *Colossus Tile Processor Architecture* pertinent to the operation of *Worker* contexts. This includes:

- The *Tile instruction set*
- The *execution model*
- The *register model*
- The *memory model*
- The *floating point unit*
- The *exception model*
- *Implementation Specifics*

1.2 Revision history

1.2.1 Release 1.2.3

Table 1.1: Changes since previous release

Change Id	Date	Comments
1.2.3	11-01-22	
1.2.2	07-10-21	
1.2.1	24-09-21	
1.2.0	08-02-21	
1.1.3	21-10-19	
1.1.2	04-02-19	
1.1.1	30-11-18	
1.1.0	02-11-18	
1.0.2	01-11-18	
1.0.1	22-08-18	
1.0.0	27-02-18	
0.9.3	21-02-18	
0.9.2	07-02-18	
0.9.1	02-01-18	
0.9.0	20-11-17	
0.8.3	13-11-17	

Continued on next page

Table 1.1 – continued from previous page

Change Id	Date	Comments
0.8.2	03-11-17	
0.8.1	03-10-17	
0.8.0	13-09-17	
0.7.4	25-08-17	
0.7.3	28-07-17	
0.7.2	07-07-17	
0.7.1	04-05-17	
0.7.0	06-02-17	

1.2.2 Implementation variant

Unless stated otherwise, the implementation specific sections of this document are relevant to the following variant:

- *IPU2*

1.3 Conventions

1.3.1 Arithmetic notation and operators

The textual descriptions use the following notation:

Table 1.2: Arithmetic operators

Symbol	Meaning
<i>Oxvalue</i>	A value written using hexadecimal notation
<i>Obvalue</i>	A value written using binary notation
<i>value</i>	A value written using decimal notation
\$SOME_STATE	Refers to Tile architectural state
\$SOME_STATE.FIELD	Refers to a field of some Tile architectural state

The instruction semantics are defined using the *C* language.

1.4 Glossary

ARF The *auxiliary* (or arithmetic) register file

Atomic Sections Sections of *Supervisor* code that cannot be interrupted without breaking the functionality.

aux The name given to one of *Tiles Execution Pipelines*

Barrier Synchronisation A system-wide synchronisation and the first phase in a *Superstep*, following which it is safe to perform an *Exchange Phase*.

BREAK A recoverable exception event.

BSP Bulk synchronous parallel. A programming methodology for parallel algorithms.

Codelet Multi-input, multi-output functions which declare the state and behaviour of vertices.

Colossus A chip-scale parallel processor to accelerate machine learning. A single *Colossus* consists of a number of *Tile* processors connected by an on-chip *Exchange Fabric*.

Commit The final phase of instruction execution. See *Instruction execution semantics*

Compute The phase of instruction execution where results are computed. See *Instruction execution semantics*

Context The complete and distinct environment for a single thread of execution. *Tile* supports a single *Supervisor* context alongside a number of *Worker* contexts.

CSR Control and/or Status Register. See *Control and Status registers*.

ECC Error Correction Code.

Exception An unusual execution condition.

Except In A phase of instruction execution, during which exceptions may be raised by the instruction execution unit. See *Instruction execution semantics*

Except Out A phase of instruction execution, during which exceptions may be raised by the instruction execution unit. See *Instruction execution semantics*

Exception Event See *Exception Events*

Exchange Fabric The communication network through which *Tile* exchanges data with other *Tile* instances and IO engines.

Exchange Phase The communication phase of a *Superstep*, during which all necessary data exchanges are performed.

Execution Bundle A set of independently executable instructions that are issued together. See *Program Order*

External Exchange An *Exchange Phase* involving *Tile* instances in other Colossus instances.

f32 A *single-precision* floating-point value.

f32v2 A 2-element vector of single-precision floating-point values.

f32v4 A 4-element vector of single-precision floating-point values.

f16 A *half-precision* floating-point value.

f16v2 A 2-element vector of 16-bit floating-point values.

f16v4 A 4-element vector of 16-bit floating-point values.

f16v8 An 8-element vector of 16-bit floating-point values.

FAULT An unrecoverable exception event.

Half-precision A 16-bit floating-point number format. See *Number formats*

Immediate An operand value encoded within an instruction field

Imprecise A term used to describe *exception events* where the architectural state of *Tile* at *Event raise* is consistent with that defined by the *Commit* phase(s) of the instruction (or Execution Bundle) for which the exception was detected.

Inactive A worker *context* run mode, see *Run Modes*.

Interleave factor Every *Tile Memory region* has an interleave factor which influences the conditions under which memory element clashes occur. A *n*-way interleaved memory region has an interleave factor of *n*.

Internal Exchange An *Exchange Phase* involving only the *Tiles* within *this* Colossus instance.

IPU Intelligence Processing Unit. A synonym for *Colossus*.

ISA *Instruction Set Architecture*

main The name given to one of *Tiles Execution Pipelines*.

Memory The phase of instruction execution where memory is accessed. See *Instruction execution semantics*

MRF The *main* (or memory) register file

Naturally Aligned A data object of size *n* bytes, stored in memory at addresses *a* to *a+n-1* is naturally aligned if and only if *n* is a factor of *a*.

A set of *n* contiguously indexed registers is naturally aligned if and only if *n* is a factor of the smallest index in the set.

Patched Breakpoint A *trap* instruction that can be used to raise a *PBRK exception event*.

PC The *Program Counter* is the address from which instructions will be fetched.

Precise A term used to describe *exception events* where the architectural state of *Tile* at *Event raise* is consistent with that defined by the *Prepare* phase(s) of the instruction (or Execution Bundle) for which the exception was detected.

Prepare An early phase of instruction execution, performed after instruction issue and before the *Except In* phase.

Quiescent A specific resting state of execution of a *Worker*, *Supervisor*, or entire *Tile* instance. See *Quiescence*.

Receive The part(s) of an *Exchange Phase* dealing with the reception of data from the *Exchange Fabric*.

Register File An array of instruction operand registers with a specified number of read/write access ports. *Tile* has 2 register files, *MRF* and *ARF*.

Retirement The final phase of an instructions *lifespan*. See *Program Order*.

Sibling instruction Each instruction within an *Execution Bundle* is a sibling to the other instructions in that bundle.

Sign extended Where the sign-bit of a binary value is replicated into the most significant bits.

Single-precision A floating-point number format. See *Number formats*

Superstep A BSP *Superstep*, consisting of three *Phases*:

1. *Barrier Synchronisation*
2. *Communication/Exchange Phase*
3. *Computation Phase*

Supervisor A single execution *context*. The Supervisor execution thread being responsible for:

- Initiating worker threads
- Performing the *Exchange Phase* and *Barrier Synchronisation* phases of a *Superstep*.

A single *Tile* instance supports just one *Supervisor context*.

TDI Tile Debug Interface

Thread A sequential software task, including all associated state, executed on a *Tile context*.

Tile The Colossus Tile Processor (*Tile*) is a highly deterministic, asymmetric dual LIW processor, supporting multiple hardware resident execution contexts.

Undefined The architecture does not define an initial value for the state. No assumptions can be made about the value of the state immediately following a reset or worker launch and must be explicitly initialised before use.

Vertex A node in the graph that represents the entire application. The function of each vertex is defined in a *codelet* that is run on a *worker* context during the *compute* phases.

Vertex state A data structure, unique to a particular *vertex* instance containing (either directly or indirectly) the state of the vertex.

Word A data size; 32-bits

Worker A single execution *context*. A Worker execution thread being responsible for performing the *Computation Phase* of a *Superstep*.

A single *Tile* instance supports an implementation specific number of *Worker* contexts (6 on this implementation).

Zero extended Where the *n* most significant bits of a binary value are forced to zero.

Zero tailed Where the *n* least significant bits of a binary value are forced to zero.

MACHINE FRAMEWORK

2.1 Overview

The Colossus Tile Processor (*Tile*) is a highly deterministic, asymmetric dual LIW processor, supporting multiple hardware resident execution contexts. *Tile*'s multiple contexts are time multiplexed onto shared hardware resources in a manner that achieves high utilisation by hiding local instruction latencies, including memory access and branch latencies.

Each *Tile* instance includes a tightly coupled local memory used to store all code and data required by the execution contexts, as well as an instruction driven interface onto an exchange fabric.

Tile is the compute building block of the Colossus chip-scale parallel processor. As such it is designed to efficiently perform computation on sparse, mixed-precision floating-point data in collaboration with other *Tile* instances, using the *BSP* operating model.

The Tile Architecture specification is the result of a wonderfully collaborative, interdisciplinary engineering effort.

2.2 Logical Block structure

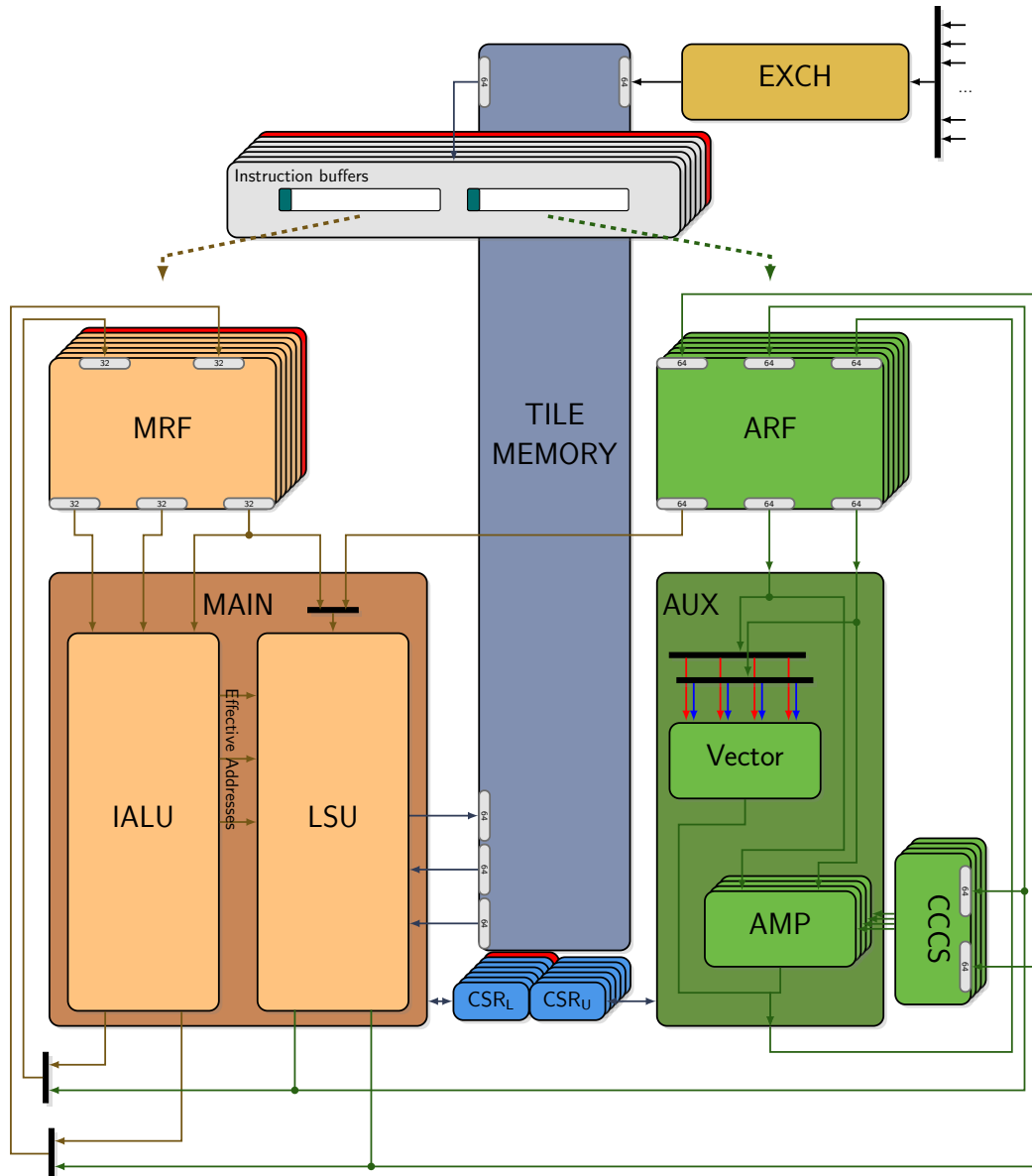


Fig. 2.1: Tile logical block structure

2.3 Hardware Contexts

Tile supports a single *Supervisor* context alongside a number of *Worker* contexts. The total number of *Worker* contexts supported (CTXT_WORKERS (6)) is *implementation dependent*.

Context specific architectural state is replicated per execution context (see *Register model*).

The *Supervisor* context has the numerical Id 0. The *Worker* contexts have numerical Ids in the range [1, 6].

Tile supports a number of execution slots, with the total number of slots being equal to the number of *Worker* contexts supported. Initially the *Supervisor* context claims all execution slots and is responsible for allocating workers to execution slots via the *run* instruction. Once all execution slots have been allocated to *Workers*, the *Supervisor* context is *Suspended* until an execution slot is made available by the termination of a *Worker*.

A round-robin schedule is used to time multiplex the execution slots (and therefore active contexts) onto the shared hardware resources. Only the *Supervisor* thread can claim multiple execution slots.

At any point in time a *Worker* will be operating in exactly one *Run Mode*.

2.4 Execution Pipelines

Tile employs a pair of asymmetric execution pipelines, *main* and *aux*. Note that the *Supervisor* context cannot use the *aux* pipeline and its associated state.

Every valid *Tile* instruction is defined to be exactly one of the following:

- executable by the *main* pipeline only
- executable by the *aux* pipeline only
- executable by both execution pipelines¹

Instructions may be issued:

- In solo to either pipeline, with the appropriate execution pipeline being determined from the instruction opcode. Any attempt to execute a solo *aux* instruction by the *Supervisor* context will result in a `TEX-CPT_INVALID_INSTR Exception`.
- In parallel, so that each instruction in an *Execution Bundle* is executed independently, in parallel by the appropriate pipeline (see *Co-issue*).

Note that since the *Supervisor* context cannot use the *aux* pipeline, it is also incapable of parallel instruction execution.

Each pipeline is affiliated with a particular *register file* and for the vast majority of instructions that specific register file is used to provide both the source and destination register operands. The *main* execution pipeline is associated (and tightly coupled) with *MRF* and the *aux* pipeline with *ARF*. In terms of providing instruction register operands, the following exceptions apply:

- *atom* takes its source register operand from *ARF* and writes its result to *MRF*.
- Some store instructions take their source operands from either *MRF*, *ARF* or both.
- Some load instructions perform writebacks to either *MRF*, *ARF*, or both.

main and *aux* can also take input operand values from *immediate* instruction fields.

In addition to the *ARF*, the *aux* execution pipeline utilises a small amount of internal state (see *Pipeline internal state*).

The *Tile ISA* defines a clear split in the intended role of the two pipelines:

- *main* is designed primarily to perform control flow, address manipulation, integer arithmetic and load/store operations
- *aux* is designed primarily to perform floating-point based compute

2.5 Program Order

To define program order, the lifespan of a *Tile* instruction instance is broken down into a number of phases:

1. *Instruction fetch*, where the instruction encoding is read from *Tile Memory*
2. *Instruction issue*, where the instruction (or *Execution Bundle*) is dispatched to an execution unit for further processing.
3. *Instruction execution*, which itself is broken down into a number of sub-phases (see *Instruction execution semantics*)
4. *Instruction retirement*, where architectural state updates resulting from the instruction execution are committed.

Architecturally speaking, an instruction instance will always progress through all phases unimpeded once it has entered the first phase. The only mechanism that can alter this behaviour is that of *exception events*. At any time and for each context, *\$PC* holds the *Tile Memory* address of the next instruction that would be retired by that context in the absence of *exception events*.

For *Worker* contexts the initial value of *\$PC* is determined by the *run* instruction (executed by the *Supervisor*).

¹ In which case the two instruction variants will be distinguishable by their opcode values.

For all contexts, instructions (or *Execution Bundles*) are always executed in the sequence defined by the updates of **\$PC**. All non-control class solo instructions (*Control*) implicitly increment the value of **\$PC** by 4 during *Instruction retirement* (8 for *Execution Bundles*). For control class instructions, the new value of **\$PC** to be committed at *Instruction retirement* is defined by the individual instruction semantics.

For *Workers*, each instruction may be issued in solo, or in parallel with another instruction as part of an *Execution Bundle*, depending on the precise mixture and nature of instructions presented for issue (see *Co-issue*). Each instruction within an *Execution Bundle* is executed independently, using distinct execution pipelines.

2.5.1 Co-issue

For *Worker* contexts, instructions may be issued (and therefore executed) in solo or in parallel with another instruction, as part of an *Execution Bundle*.

The semantics of parallel execution of a *Tile Execution Bundle* are as follows:

- All *Tile* architectural state is consistent across the co-issued instructions and with that resulting from the *Instruction retirement* phase of the previous solo instruction or *Execution Bundle*.
- There is no guaranteed relative ordering of *Instruction retirement* between the co-issued instructions.
- *Tile* architectural state for the next solo instruction or *Execution Bundle* will be consistent with that resulting from the *Instruction retirement* phase for both co-issued instructions.
- *Precise Exceptions* raised by one execution pipeline will squash all architectural updates associated with **both** co-issued instructions.
- *Imprecise Exceptions* (which can only be raised by the *aux* pipeline) will not squash the architectural updates associated with **either** of the co-issued instructions.
- *Semi-precise Exceptions* (which can only be raised by the store instructions and therefore raised only by the *main* pipeline) will squash all architectural updates associated with **both** co-issued instructions, except for the modification of *Tile Memory*.

The **Co-issue field of an instruction (bit 31)** dictates the eligibility of an instruction for parallel issue:

- [0]: A *Co-issue* field value of 0 indicates that this instruction is not part of an *Execution Bundle*
- [1]: A *Co-issue* field value of 1 indicates that this instruction is part of an *Execution Bundle*

The behaviour for any adjacent pair of instructions in the context specific instruction stream is given by *Co-issue behaviour for adjacent instructions*.

The following conditions and restrictions also apply:

1. The *Supervisor* context can only execute solo instructions and only using the *main* pipeline.
2. For some instructions, the *Co-issue* field is forced to zero. For these instructions, attempting to use a non-zero *Co-issue* value will result in a `TEXCPT_INVALID_INSTR` Exception.
3. The same register must not be used as a destination register for both instructions. Attempting to use a common destination register will result in a `TEXCPT_INVALID_INSTR` Exception.
4. It is not possible to execute a solo instruction as part of a *rpt* repeat-body. Attempt to do so will result in a `TEXCPT_INVALID_INSTR` instruction being raised.

Table 2.1: Co-issue behaviour for adjacent instructions

Instruction <i>i</i>	Instruction <i>i+1</i>	Supervisor action	Worker action
0 main	⊗ main/aux	Solo issue of instruction <i>i</i> to main pipeline	If executing a repeat-loop, <code>TEXCPT_INVALID_INSTR</code> exception is raised. Otherwise, solo issue of instruction <i>i</i> to <i>main</i>
0 aux	⊗ main/aux	<code>TEXCPT_INVALID_INSTR</code> exception raised	If executing a repeat-loop, <code>TEXCPT_INVALID_INSTR</code> exception is raised. Otherwise, solo issue of instruction <i>i</i> to <i>aux</i>

Continued on next page

Table 2.1 – continued from previous page

Instruction <i>i</i>	Instruction <i>i+1</i>	Supervisor action	Worker action
1 main	0 main/aux	<i>TEXCPT_INVALID_INSTR</i> exception raised	<i>TEXCPT_INVALID_INSTR</i> exception raised
1 main	1 main	<i>TEXCPT_INVALID_INSTR</i> exception raised	<i>TEXCPT_INVALID_INSTR</i> exception raised
1 main	1 aux	<i>TEXCPT_INVALID_INSTR</i> exception raised	Issue of Execution Bundle (comprised of <i>main</i> instruction <i>i</i> and <i>aux</i> instruction <i>i+1</i>)
1 aux	x main/aux	<i>TEXCPT_INVALID_INSTR</i> exception raised	<i>TEXCPT_INVALID_INSTR</i> exception raised

2.6 Execution semantics

2.6.1 Register read values

The value of all source registers is that following the *retirement* phase of the previous instruction (if issued in solo) or *Execution Bundle* if *Co-issued*. Naturally, for the first instruction executed in each context, the source register values are consistent with their architecturally defined reset values.

2.6.2 Run Modes

At any point in time a *Tile* context will be operating in exactly one *Run Mode*. Transitions between run modes are performed either by the execution of certain instructions or automatically at the completion of certain activities. Not all run modes are valid for all context types.

Table 2.2: Worker run modes

Run mode	Quiescent?	Description
Inactive	✓	The default (reset) run mode for <i>Worker</i> contexts. The <i>context</i> is not executing instructions or participating in any other type of activity. The context will remain <i>Inactive</i> until either: <ol style="list-style-type: none">1. it is allocated an execution slot by the <i>Supervisor</i> context executing a <i>run</i> instruction, or2. an instruction is <i>injected</i> into the <i>Worker</i> context Entry to the <i>Inactive</i> run mode is via the <i>exit</i> instructions.
Executing	✗	The context is actively issuing instructions. <i>Worker</i> contexts are placed into the Executing run mode by the <i>Supervisor</i> context executing <i>run</i> .
Excepted	✓	The <i>Worker</i> has raised an <i>exception event</i> . For certain types of exception it is possible to transition back into the <i>Executing</i> run mode by clearing down (recovering from) the exception.
Repeating	✗	The <i>Worker</i> has executed a <i>rpt</i> instruction and <i>\$REPEAT_COUNT</i> is non-zero. The <i>Worker</i> context run mode will transition back to <i>Executing</i> when <i>\$REPEAT_COUNT</i> becomes 0.
Executing injected instruction	✗	A <i>Worker</i> will transition into this state from a quiescent run mode whenever an <i>injected</i> instruction for this <i>Worker</i> enters the pipeline. If the injected instruction causes an <i>exception event</i> the run mode will transition to <i>Excepted</i> . Otherwise the run mode will transition back to its previous state.

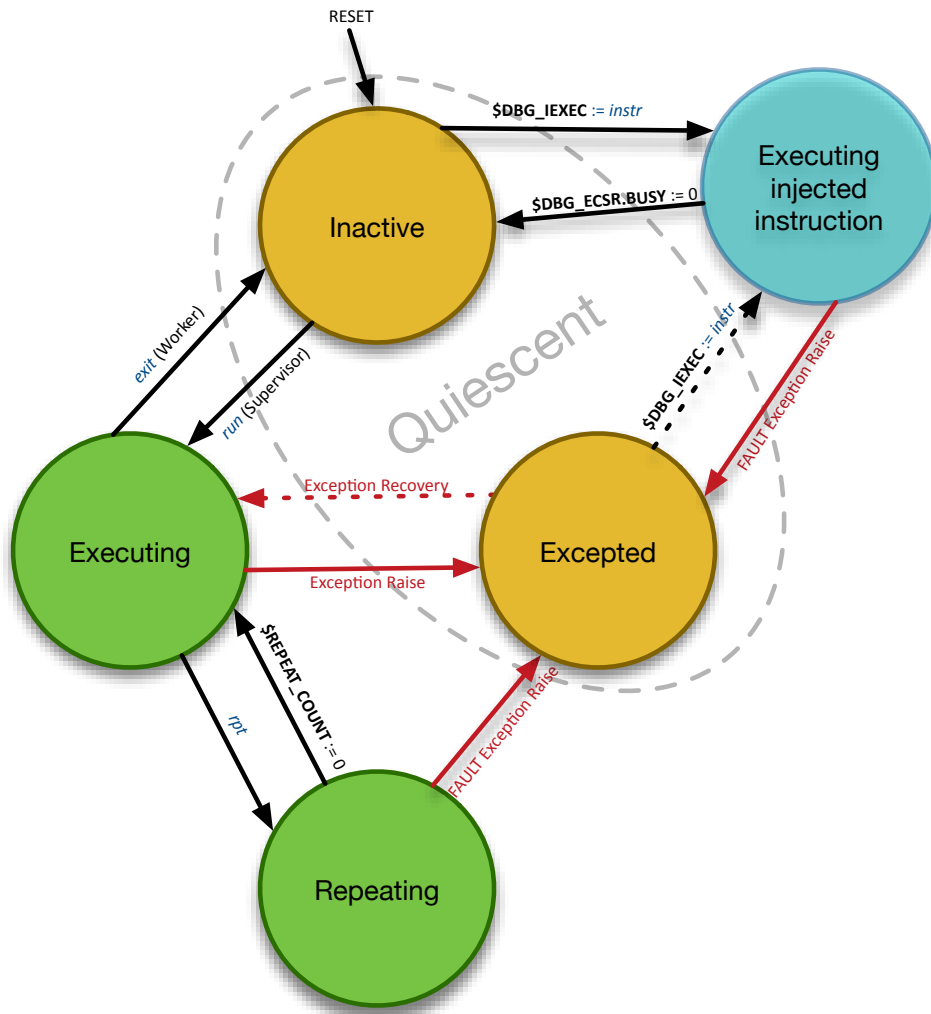


Fig. 2.2: Worker Context Run Mode Transitions

2.6.3 Quiescence

Note: The details provided by this section are not relevant to the *Tile Vertex ISA*.

2.7 Register model

2.7.1 Register access semantics

Table 2.3: Register field access semantics

Type	Write behaviour	Read behaviour	Dynamically modified by hardware?	Example
RO	Ignored	Reads always yield the same value.	X	<code>\$WSR.CTXTID_M1</code>
RvO	Ignored	Reads yield current value of dynamic state.	✓	<code>\$COUNT_L.VALUE</code>
RW	Write value committed to state.	Reads yield last value written (or reset value if never written to).	X	<code>MRF</code>
RvW	Write value committed to state.	Reads yield current value of dynamic state.	✓	<code>\$FP_CTL.INV</code>
WC	Write value committed to state.	Yields 0	X	<code>\$PRNG_SEED.VALUE</code>
W1C	Write of 0b1 initiates a background operation.	Yields 0	X	<code>\$FP_CLR.INV</code>
WO	Write value committed to state.	Not applicable. There is no mechanism available to directly read this state. It may be possible to read the state indirectly.	X	<code>\$CWEI_n_0</code>
RESERVED	Ignored	Yields 0	X	<code>\$PC.RESERVED</code>

2.7.2 Context register state

Each execution context includes a unique and private set of the following architectural state:

- *Control and Status registers*
- An array of 32-bit registers supplying operand values to the *main* pipeline *MRF*
- When applicable², an array of 32-bit registers supplying operand values to the *aux* pipeline *ARF*
- When applicable, a small amount of pipeline specific internal state

In addition, *Tile* provides an amount of state common to all execution contexts. This shared state can be configured by the *Supervisor* context only and is implicitly used (read) by the *Worker* contexts when performing certain operations. This common state is typically used when all *worker* contexts are performing the same basic compute function in a collaborative manner. Referred to as *Common Compute Configuration State*, this state is considered part of the *Supervisor* context's architectural state.

2.7.3 MRF

As depicted by *Logical Block structure*, the *Memory Register File*:

² The *Supervisor* context cannot issue instructions to the *aux* pipeline and so doesn't include an *ARF*. Any attempt made by the Supervisor context to execute an instruction that would require access to the *ARF* will result in a `TEXCPT_INVALID_INSTR` *Exception*.

1. provides register (or constant valued) source operand values to the *main* execution pipeline
2. accepts writebacks from the *main* execution pipeline

The MRF has 16 distinct indices, each of which may be:

1. *Populated*: These indices exhibit 32-bit *RW* semantics.
2. *Constant valued*: These indices are used to present specified constant values to the execution pipelines. Those constants may be a specific integer value, or a copy of a *RO CSR*. Writes to these indices are ignored.
3. *Unpopulated*: Writes to these indices are ignored. Reads yield either 0, or the contents of one of the populated indices. The precise read behaviour is *implementation dependent*.

The populated registers occupy a contiguous range at the lowest indices. The actual number of populated indices is *implementation dependent*.

The constant valued locations occupy a contiguous range at the upper indices (working back from 15).

Any unpopulated indices occupy the contiguous range between the top of the populated region and the bottom of the constant valued region. $\$m14$ may also be constant valued or unpopulated (*implementation dependent*).

Instruction reads and writes from/to the MRF are identified by the $\$m$ register operand mnemonic. Instructions may reference:

1. single MRF registers, such as $\$m1$
2. *naturally aligned* MRF register pairs, such as $\$m2:3$ (in which case the least significant bit of the register index field is ignored and assumed to be 0b0)

Table 2.4: MRF

Register name	Register index	Width (bits)	Semantics	Notes
$\$m0$	0	32	<i>RW</i>	Populated region. Reset value is implementation dependent and given by <i>TReg_MRFRResetValue</i> .
$\$m1$	1			
$\$m2$	2			
...	...			
$\$mN-1$	$N-1$			Unpopulated region. Does not exist when <i>MRF_GP_REGISTERS</i> is 12.
$\$mN$	N		<i>RO</i> . Reads yield implementation dependent values.	
$\$mN+1$	$N+1$			Constant value register.
$\$m12$	12		<i>RO</i> . Reads by Worker contexts return <i>\$WORKER_BASE</i> . Reads by Supervisor context return 0.	
$\$m13$	13		<i>RO</i> . Reads by Worker contexts return <i>\$VERTEX_BASE</i> . Reads by Supervisor context return 0.	Constant value register.
$\$m14$	14		<i>RO</i> . Reads yield implementation dependent values.	
$\$m15$	15		<i>RO</i> . Reads return 0.	Constant value register.

Note: N is *MRF_GP_REGISTERS*

The following code illustrates the MRF read and write semantics:

Listing 2.1: MRF Register Read

```
uint32_t *MRF_RegRead(unsigned baseIndex, TileRFAccessSize_t size, ContextState &context) {
    assert(size <= TRF_ACCESS_SIZE_PAIR);
    static uint32_t mReg[TRF_ACCESS_SIZE_PAIR];
```

```

std::vector<unsigned> indices;
TReg_RFIndices(indices, baseIndex, size);

for (size_t i = 0; i < indices.size(); i++) {
    unsigned r = indices[i];

    switch (r) {
    case 15:
        // Constant region
        mReg[i] = 0;
        break;

    case 14:
        // Unpopulated region
        mReg[i] = TReg_MRFUnpopRegRead(r);
        break;

    case 13:
        if (context.isSupervisor) {
            mReg[i] = 0;
        } else {
            mReg[i] = context.csr[CSR_W_VERTEX_BASE__INDEX]->get();
        }
        break;

    case 12:
        if (context.isSupervisor) {
            mReg[i] = 0;
        } else {
            mReg[i] = context.csr[CSR_W_WORKER_BASE__INDEX]->get();
        }
        break;

    default:
        if (r < MRF_GP_REGISTERS) {
            // Populated region
            mReg[i] = context.mrf[r];
        } else {
            mReg[i] = TReg_MRFUnpopRegRead(r);
        }
        break;
    }
}
return mReg;
}

```

Listing 2.2: MRF Register Write

```

void MRF_RegWrite(unsigned baseIndex,
                 TileRFAccessSize_t size,
                 uint32_t *data,
                 ContextState &context) {
    assert(size <= TRF_ACCESS_SIZE_PAIR);

    std::vector<unsigned> indices;
    TReg_RFIndices(indices, baseIndex, size);

    for (size_t i = 0; i < indices.size(); i++) {
        unsigned r = indices[i];

        if (r < MRF_GP_REGISTERS) {
            // Populated registers region
            context.mrf[r] = data[i];
        } else {
            // Unpopulated or constant region.
            // - Writes ignored
        }
    }
}

```

See also: *TReg_RFIndices*, *TileRFAccessSize*.

2.7.3.1 MRF Read and write ports

In order to satisfy all instruction source and destination register operand requirements efficiently, the MRF typically provides:

- 3 x 32-bit independent read ports
- 2 x 32-bit independent write ports

2.7.4 ARF

As depicted by *Logical Block structure*, the *Arithmetic Register File*:

1. provides register (or constant valued) source operand values to the *aux* execution pipeline
2. accepts writebacks from the *aux* execution pipeline
3. provides data values for certain *main store* instructions
4. accepts writebacks from the *main* execution pipeline for certain *load* instructions

The ARF has 16 distinct indices, each of which may be:

1. *Populated*: These indices exhibit 32-bit *RW* semantics.
2. *Constant valued*: These indices are used to present specified constant values to the execution pipelines. Writes to these indices are ignored.
3. *Unpopulated*: Writes to these indices are ignored. Reads yield either 0, or the contents of one of the populated indices. The precise read behaviour is *implementation dependent*.

The populated registers occupy a contiguous range at the lowest indices. The actual number of populated indices is *implementation dependent*.

The constant valued locations occupy a contiguous range at the upper indices (working back from 15).

Any unpopulated indices occupy the contiguous range between the top of the populated region and the bottom of the constant valued region.

Instruction reads and writes from/to the ARF are identified by the *\$a* register operand mnemonic. Instructions may reference:

1. single ARF registers, such as *\$a1*
2. *naturally aligned* ARF register pairs, such as *\$ap2* or *\$a4:5* (in which case the least significant bit of the register index field is ignored and assumed to be 0b0)
3. *naturally aligned* ARF register quads, such as *\$aq1* or *\$a4:7* (in which case the 2 least significant bits of the register index field are ignored and assumed to be 0b00)

Table 2.5: ARF

Register name	Register index	Width (bits)	Semantics	Notes
\$a0	0	32	<i>RW</i>	Populated region. Reset value is implementation dependent and given by <i>TReg_ARFResetValue</i> .
\$a1	1			
\$a2	2			
...	...			
\$aM-1	<i>M-1</i>			Unpopulated region. Will not exist if <i>ARF_GP_REGISTERS</i> is 14.
\$aM	<i>M</i>		<i>RO</i> . Reads yield implementation dependent values.	
\$aM+1	<i>M+1</i>			
\$a14	14		<i>RO</i> . Reads return 0.	Constant value register.
\$a15	15			

Note: M is `ARF_GP_REGISTERS`

The following code illustrates the ARF read and write semantics:

Listing 2.3: ARF Register Read

```
uint32_t *ARF_RegRead(unsigned baseIndex, TileRFAccessSize_t size, ContextState &context) {
    assert(size <= TRF_ACCESS_SIZE_MAX);
    static uint32_t aReg[TRF_ACCESS_SIZE_MAX];

    std::vector<unsigned> indices;
    TReg_RFIndices(indices, baseIndex, size);

    bool zeroReg = TReg_ARFIsZeroReg(baseIndex, size);

    for (size_t i = 0; i < indices.size(); i++) {
        unsigned r = indices[i];
        if (zeroReg) {
            // Constant region
            aReg[i] = 0;
        } else if (r < ARF_GP_REGISTERS) {
            // Populated region
            aReg[i] = context.arf[r];
        } else {
            // Unpopulated region - return value is implementation dependent
            aReg[i] = TReg_ARFUnpopRegRead(r, size);
        }
    }
    return aReg;
}
```

Listing 2.4: ARF Register Write

```
void ARF_RegWrite(unsigned baseIndex,
                 TileRFAccessSize_t size,
                 uint32_t *data,
                 ContextState &context,
                 uint32_t noWriteBackMask) {
    assert(size <= TRF_ACCESS_SIZE_MAX);

    std::vector<unsigned> indices;
    TReg_RFIndices(indices, baseIndex, size);

    bool zeroReg = TReg_ARFIsZeroReg(baseIndex, size);

    for (size_t i = 0; i < indices.size(); i++) {
        unsigned r = indices[i];
        // Skip register if writeback bit is set
        if (noWriteBackMask & (1 << i))
            continue;
        if (zeroReg) {
            // constant region
            // - Writes ignored
        } else if (r < ARF_GP_REGISTERS) {
            // Populated registers region
            context.arf[r] = data[i];
        } else {
            // Unpopulated region.
            // - Writes ignored
        }
    }
}
```

See also: `TReg_RFIndices`, `TileRFAccessSize`.

2.7.4.1 ARF Read and write ports

In order to satisfy all instruction source and destination register operand requirements efficiently, the ARF typically provides:

- 3 x 64-bit independent read ports

- 3 x 64-bit independent write ports³

2.7.5 Control and Status registers

The *Control and Status Register sets (CSRs)* form part of an execution *context*. They present pertinent information related to the status of an execution context as well as controlling specific execution behaviour. *CSRs* are accessed via a specific *CSR address space*, which is split into 2 halves:

- The *lower CSR space*, spanning the address range from 0 to 255 (inclusive): *Control and Status* registers within this region can be read using the *get* instruction. Some *Control and Status* registers in this region can be written using the *put* instructions.
- The *upper CSR space*, spanning the address range from 256 to 511 (inclusive): *Control and Status* registers within this region can be read using the *uget* instruction. Some *Control and Status* registers in this region can be written using the *uput* instruction.

The *Control and Status* register sets are identical across all *workers* (see *Worker CSRs*).

2.7.5.1 Supervisor CSRs

Table 2.6: Supervisor Control and Status registers

Index	Register name	Reset value	Description
0x20	<i>\$FP_CTL</i>	0x00000000	Floating point control register initial value.
0x50	<i>\$CCCSLOAD</i>	0x00000000	Post-incrementing load address for <i>ld64putcs</i> and <i>ld128putcs</i> .
0x72	<i>\$CTXT_STS</i>	0x00000000	Context execution status register.

2.7.5.1.1 \$FP_CTL

Register index: 32

Floating point control register initial value. The *worker* floating-point control register *\$FP_CTL* is initialised with the contents of this register prior to worker activation.

See *run* and *runall*

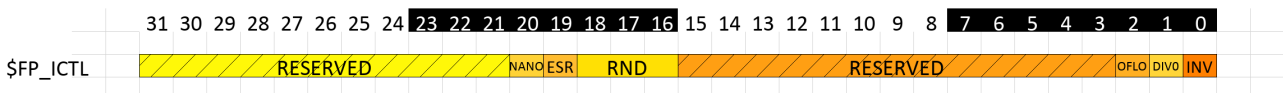


Fig. 2.3: *\$FP_CTL* register format

Table 2.7: *\$FP_CTL* register fields

Field name	Bit field	Reset value	Access semantics	Description
INV	[0]	0	RW	<ul style="list-style-type: none"> • 0b0: Floating-point invalid operation exceptions treated as benign. • 0b1: Floating-point invalid operation exceptions treated as malign.

Continued on next page

³ 2 of the write ports are used exclusively by instructions executed by the *main* pipeline. The other write port is used exclusively by instructions executed by the *aux* pipeline.

Table 2.7 – continued from previous page

Field name	Bit field	Reset value	Access semantics	Description
DIV0	[1]	0	<i>RW</i>	<ul style="list-style-type: none"> • 0b0: Floating-point divide-by-zero exceptions treated as benign. • 0b1: Floating-point divide-by-zero exceptions treated as malign.
OFLO	[2]	0	<i>RW</i>	<ul style="list-style-type: none"> • 0b0: Floating-point overflow exceptions treated as benign. • 0b1: Floating-point overflow exceptions treated as malign.
RND	[18:16]	0	<i>RO</i>	<p>Floating-point rounding behaviour. See <i>TileRound-Mode</i>.</p> <p>Note: Not all rounding modes are supported by all implementations (see <i>Parameters</i>). Attempts to write values corresponding to RESERVED or unimplemented modes will be ignored. Reads will yield non-RESERVED, implemented modes.</p>
ESR	[19]	0	<i>RW</i>	Enable stochastic rounding for those instructions that support it.
NANOO	[20]	0	<i>RW</i>	<p>Enable NAN On Overflow mode. When enabled <i>half-precision</i> calculations that have overflowed will:</p> <ol style="list-style-type: none"> 1. produce a quiet NaN result, rather than saturating to the f16 maximum/minimum values. 2. Set the invalid operation flag

2.7.5.1.2 \$CCCSLOAD

Register index: 80

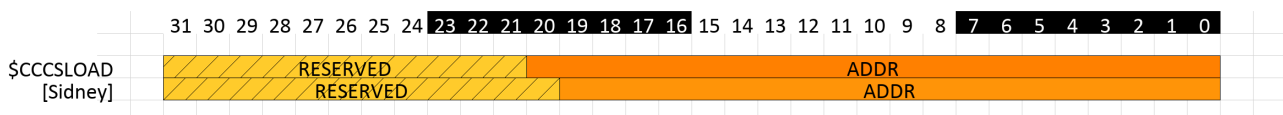


Fig. 2.4: \$CCCSLOAD register format

Table 2.8: \$CCCSLOAD register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:0]	0	<i>RvW</i>	Byte address

\$CCCSLOAD register field sizes are influenced by the following parameters: *TMEM_BYTE_ADDRESS_WIDTH*

2.7.5.1.3 \$CTXT_STS

Register index: 114

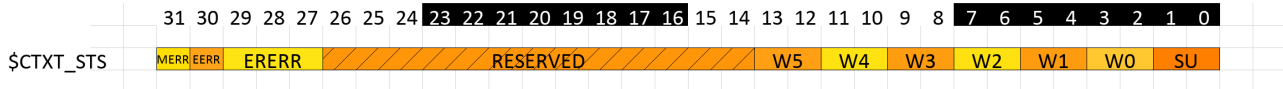


Fig. 2.5: \$CTXT_STS register format

Table 2.9: \$CTXT_STS register fields

Field name	Bit field	Reset value	Access semantics	Description
SU	[1:0]	0	RvO	Supervisor context status, as specified by <i>TileCtxtStatus</i>
W0	[3:2]	0	RvO	Worker context 0 status, as specified by <i>TileCtxtStatus</i>
W1	[5:4]	0	RvO	Worker context 1 status, as specified by <i>TileCtxtStatus</i>
W2	[7:6]	0	RvO	Worker context 2 status, as specified by <i>TileCtxtStatus</i>
W3	[9:8]	0	RvO	Worker context 3 status, as specified by <i>TileCtxtStatus</i>
W4	[11:10]	0	RvO	Worker context 4 status, as specified by <i>TileCtxtStatus</i>
W5	[13:12]	0	RvO	Worker context 5 status, as specified by <i>TileCtxtStatus</i>
ERERR	[29:27]	0	RvO	Exchange receive error status flag.
EERR	[30]	0	RvO	Exchange error status flag <ul style="list-style-type: none"> • 0b0: No exchange error detected • 0b1: Exchange parity error detected
MERR	[31]	0	RvO	Memory error status flag <ul style="list-style-type: none"> • 0b0: No memory error detected • 0b1: Memory ECC/Parity error detected

2.7.5.2 Worker CSRs

Table 2.10: Worker Control and Status registers

Index	Register name	Reset value	Description
0x00	<i>\$PC</i>	0x0004c000	Context Program Counter.
0x01	<i>\$WSR</i>	0x00000000	Worker context status register.
0x02	<i>\$VERTEX_BASE</i>	0x00000000	Vertex data structure pointer.
0x03	<i>\$WORKER_BASE</i>	0x00000000	Worker context scratch space base address.
0x04	<i>\$REPEAT_COUNT</i>	0x00000000	<i>rpt</i> loop down-counter.
0x05	<i>\$REPEAT_FIRST</i>	0x00000000	<i>rpt</i> loop start address.
0x06	<i>\$REPEAT_END</i>	0x00000000	<i>rpt</i> loop end address.

Continued on next page

Table 2.10 – continued from previous page

Index	Register name	Reset value	Description
0x60	<i>\$COUNT_L</i>	0x00000000	Tile cycle counter value. Lower 32-bits .
0x61	<i>\$COUNT_U</i>	0x00000000	Tile cycle counter value. Upper 32-bits .
0x70	<i>\$DBG_DATA</i>	0x00000000	Alias for <i>\$DBG_DATA</i> debug register.
0x71	<i>\$DBG_BRK_ID</i>	0x00000000	Id of BRK channel which caused the last <i>BREAK exception event</i> . See <i>Debug</i> .
0x100	<i>\$FP_STS</i>	0x00000000	Floating point status register.
0x101	<i>\$FP_CLR</i>	0x00000000	Floating point exception/state clear.
0x102	<i>\$FP_CTL</i>	0x00000000	Floating point control register.
0x103	<i>\$PRNG_0_0</i>	0x00000000	The least significant 32-bits of <i>\$PRNG_0</i> .
0x104	<i>\$PRNG_0_1</i>	0x00000000	The most significant 32-bits of <i>\$PRNG_0</i> .
0x105	<i>\$PRNG_1_0</i>	0x00000000	The least significant 32-bits of <i>\$PRNG_1</i> .
0x106	<i>\$PRNG_1_1</i>	0x00000000	The most significant 32-bits of <i>\$PRNG_1</i> .
0x107	<i>\$PRNG_SEED</i>	0x00000000	32-bit Pseudo-random-number-generator initialisation.
0x108	<i>\$TAS</i>	0x00000000	The axpy scale (or Temporary amp storage).

2.7.5.2.1 \$PC

Register index: 0

Context Program Counter.

Note: For Worker contexts, the initial value of this register following worker launch is set by the Supervisor run instruction.

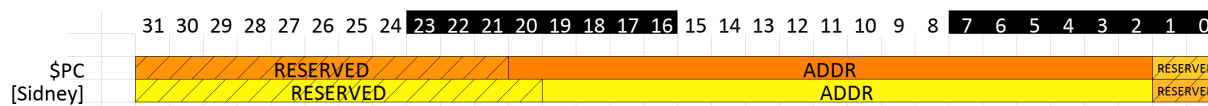


Fig. 2.6: \$PC register format

Table 2.11: \$PC register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:2]	77824	<i>RvO</i>	Program counter address.

\$PC register field sizes are influenced by the following parameters: *TMEM_WORD_ADDRESS_WIDTH*

2.7.5.2.2 \$WSR

Register index: 1

Worker context status register.

Note: The value of this register is retained between worker termination via *exit* and launch via *run*.

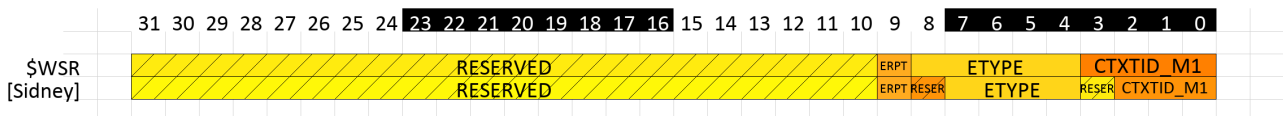


Fig. 2.7: \$WSR register format

Table 2.12: \$WSR register fields

Field name	Bit field	Reset value	Access semantics	Description
CTXTID_M1	[2:0]	0	RO	Unique id for this hardware context, minus 1. Reset value specified is for the first worker context only.
ETYPE	[7:4]	0	RvO	Exception type (see <i>TileException</i>).
ERPT	[9]	0	RvO	Exception was raised within body of <i>rpt</i> . When set, the resulting <i>exception</i> is always <i>malign</i> .

\$WSR register field sizes are influenced by the following parameters: *CTXT_TOTAL_BITWIDTH*, *TEXT_CPT_ENUM_BITWIDTH*

2.7.5.2.3 \$VERTEX_BASE

Register index: 2

Vertex data structure pointer. Initialised on behalf of a Worker context by the Supervisor via *run* or *runall*. This register can also be read through an *MRF* alias.

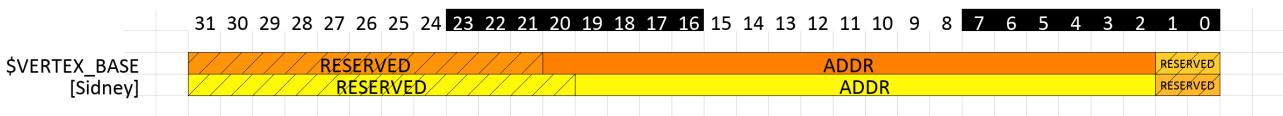


Fig. 2.8: \$VERTEX_BASE register format

Table 2.13: \$VERTEX_BASE register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:2]	0	RO	32-bit aligned address

\$VERTEX_BASE register field sizes are influenced by the following parameters: *TMEM_WORD_ADDRESS_WIDTH*

2.7.5.2.4 \$WORKER_BASE

Register index: 3

Worker context scratch space base address. Read alias of Supervisor \$WORKER<ctxId>_BASE.

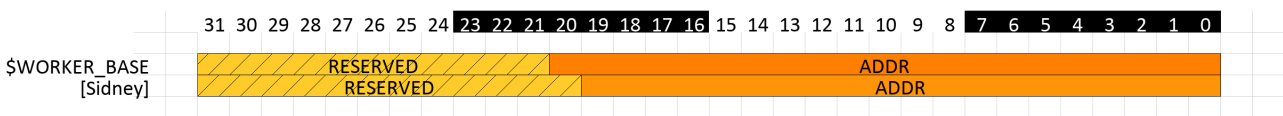


Fig. 2.9: \$WORKER_BASE register format

Table 2.14: **\$WORKER_BASE** register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:0]	0	RO	Byte address

\$WORKER_BASE register field sizes are influenced by the following parameters: *TMEM_BYTE_ADDRESS_WIDTH*

2.7.5.2.5 \$REPEAT_COUNT

Register index: 4

The number of repetitions of a *rpt* repeat-body remaining. Note that any *exception* detected when **\$REPEAT_COUNT** is non-zero will be treated as *malign* (including Debug exceptions).

Note: The value of this register is retained between worker exit and launch.

\$REPEAT_COUNT.VALUE register field is 16 bits wide.

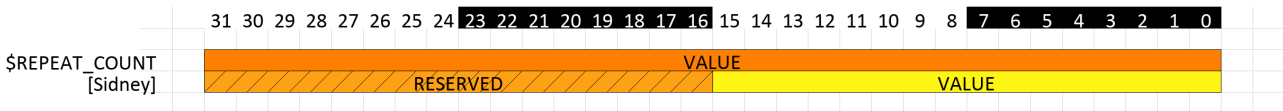


Fig. 2.10: **\$REPEAT_COUNT** register format

Table 2.15: **\$REPEAT_COUNT** register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[15:0]	0	RvO	Value

\$REPEAT_COUNT register field sizes are influenced by the following parameters: *TREG_REPEAT_COUNT_WIDTH*

2.7.5.2.6 \$REPEAT_FIRST

Register index: 5

The address of the initial *Execution Bundle* of the current (previous if not currently executing a repeat block) *rpt* repeat-body.

Note: The value of this register is *undefined* at worker launch

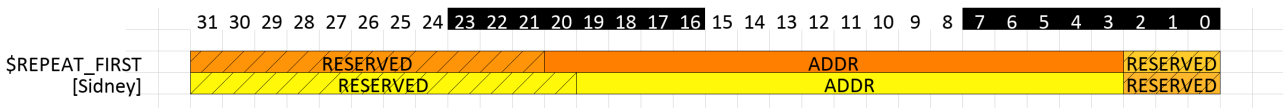


Fig. 2.11: **\$REPEAT_FIRST** register format

Table 2.16: \$REPEAT_FIRST register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:3]	0	RvO	64-bit aligned address

\$REPEAT_FIRST register field sizes are influenced by the following parameters:
TMEM_DWORD_ADDRESS_WIDTH

2.7.5.2.7 \$REPEAT_END

Register index: 6

The address of the very next instruction following the current (previous if not currently executing a repeat block) *rpt* repeat-body.

Note: The value of this register is *undefined* at worker launch

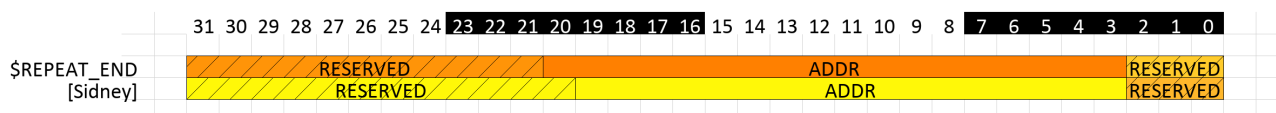


Fig. 2.12: \$REPEAT_END register format

Table 2.17: \$REPEAT_END register fields

Field name	Bit field	Reset value	Access semantics	Description
ADDR	[19:3]	0	RvO	64-bit aligned address

\$REPEAT_END register field sizes are influenced by the following parameters:
TMEM_DWORD_ADDRESS_WIDTH

2.7.5.2.8 \$COUNT_L

Register index: 96

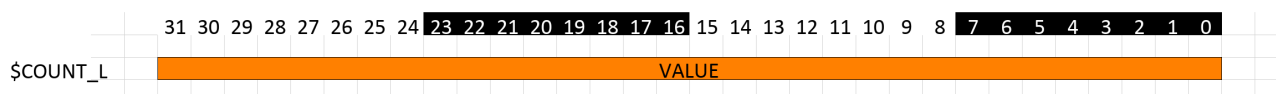


Fig. 2.13: \$COUNT_L register format

Table 2.18: \$COUNT_L register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvO	Value

2.7.5.2.9 \$COUNT_U

Register index: 97

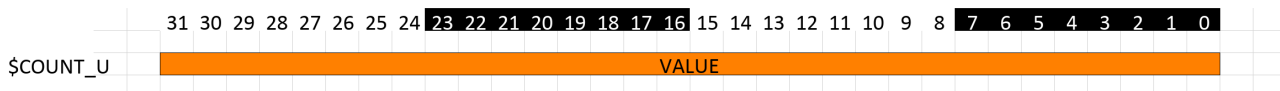


Fig. 2.14: \$COUNT_U register format

Table 2.19: \$COUNT_U register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvO	Value

2.7.5.2.10 \$DBG_DATA

Register index: 112

Alias for \$DBG_DATA debug register. Visible and writable from all contexts but only 1 actual piece of architectural state.

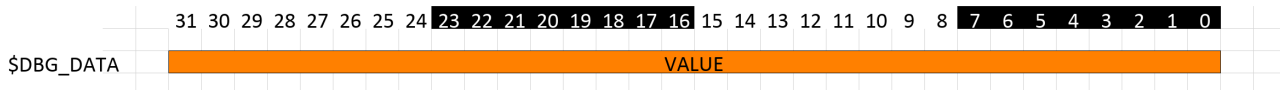


Fig. 2.15: \$DBG_DATA register format

Table 2.20: \$DBG_DATA register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvW	Value

2.7.5.2.11 \$DBG_BRK_ID

Register index: 113

Id of BRK channel which caused the last BREAK *exception event*. See *Debug*.

Note: For Worker contexts, the value of this register is retained between worker exit and launch.

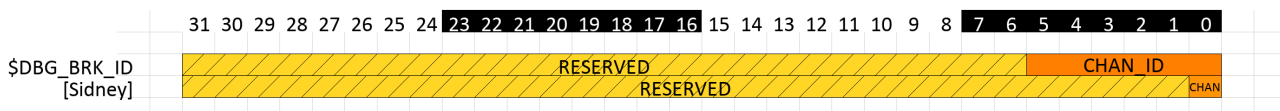


Fig. 2.16: \$DBG_BRK_ID register format

Table 2.21: \$DBG_BRK_ID register fields

Field name	Bit field	Reset value	Access semantics	Description
CHAN_ID	[0]	0	RvO	Channel Id.

\$DBG_BRK_ID register field sizes are influenced by the following parameters: *TDBG_TOTAL_CHANNELS_CEIL_LOG2*

2.7.5.2.12 \$FP_STS

Register index: 256

Floating point status register.

Note: This register is explicitly reset on worker launch

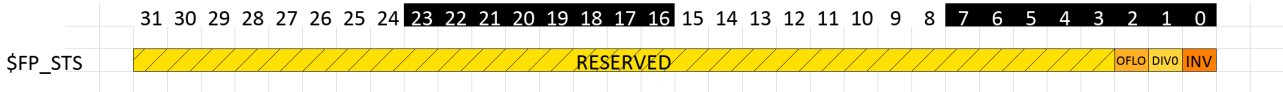


Fig. 2.17: \$FP_STS register format

Table 2.22: \$FP_STS register fields

Field name	Bit field	Reset value	Access semantics	Description
INV	[0]	0	<i>RvO</i>	Floating point invalid operation exception flag
DIVO	[1]	0	<i>RvO</i>	Floating point divide-by-zero exception flag
OFLO	[2]	0	<i>RvO</i>	Floating point overflow exception flag

2.7.5.2.13 \$FP_CLR

Register index: 257

Floating point exception/state clear. Write 0b1 to clear the specified floating point exception flag or internal state. Reads always return 0.

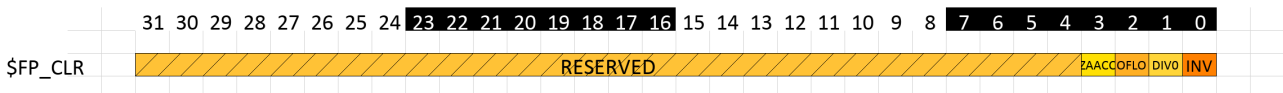


Fig. 2.18: \$FP_CLR register format

Table 2.23: \$FP_CLR register fields

Field name	Bit field	Reset value	Access semantics	Description
INV	[0]	0	<i>WIC</i>	Floating point invalid operation exception flag
DIVO	[1]	0	<i>WIC</i>	Floating point divide-by-zero exception flag
OFLO	[2]	0	<i>WIC</i>	Floating point overflow exception flag
ZAACC	[3]	0	<i>WIC</i>	Force all accumulators \$AACC[][] to zero.

2.7.5.2.14 \$FP_CTL

Register index: 258

Floating point control register.

Note: The initial value of this register on worker launch is provided by the :term'Supervisor'.

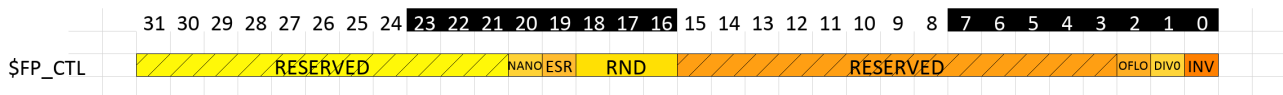


Fig. 2.19: \$FP_CTL register format

Table 2.24: \$FP_CTL register fields

Field name	Bit field	Reset value	Access semantics	Description
INV	[0]	0	<i>RvW</i>	<ul style="list-style-type: none"> • 0b0: Floating-point invalid operation exceptions treated as benign. • 0b1: Floating-point invalid operation exceptions treated as malign.
DIV0	[1]	0	<i>RvW</i>	<ul style="list-style-type: none"> • 0b0: Floating-point divide-by-zero exceptions treated as benign. • 0b1: Floating-point divide-by-zero exceptions treated as malign.
OFLO	[2]	0	<i>RvW</i>	<ul style="list-style-type: none"> • 0b0: Floating-point overflow exceptions treated as benign. • 0b1: Floating-point overflow exceptions treated as malign.
RND	[18:16]	0	<i>RvO</i>	<p>Floating-point rounding behaviour. See <i>TileRound-Mode</i>.</p> <hr/> <p>Note: Not all rounding modes are supported by all implementations (see <i>Parameters</i>). Attempts to write values corresponding to RESERVED or unimplemented modes will be ignored. Reads will yield non-RESERVED, implemented modes.</p> <hr/>
ESR	[19]	0	<i>RvW</i>	Enable stochastic rounding for those instructions that support it.
NANOO	[20]	0	<i>RvW</i>	<p>Enable NAN On Overflow mode. When enabled <i>half-precision</i> calculations that have overflowed will:</p> <ol style="list-style-type: none"> 1. produce a quiet NaN result, rather than saturating to the f16 maximum/minimum values. 2. Set the invalid operation flag

2.7.5.2.15 \$PRNG_0_0

Register index: 259

The least significant 32-bits of \$PRNG_0.

Note: The value of this register is retained between worker exit and launch.

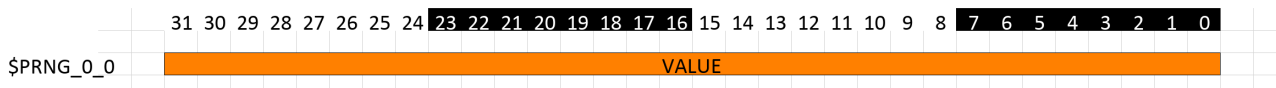


Fig. 2.20: \$PRNG_0_0 register format

Table 2.25: \$PRNG_0_0 register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvW	Value

2.7.5.2.16 \$PRNG_0_1

Register index: 260

The most significant 32-bits of \$PRNG_0.

Note: The value of this register is retained between worker exit and launch.

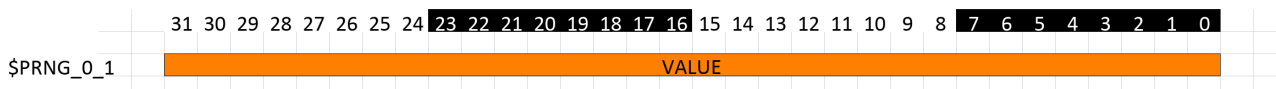


Fig. 2.21: \$PRNG_0_1 register format

Table 2.26: \$PRNG_0_1 register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvW	Value

2.7.5.2.17 \$PRNG_1_0

Register index: 261

The least significant 32-bits of \$PRNG_1.

Note: The value of this register is retained between worker exit and launch.

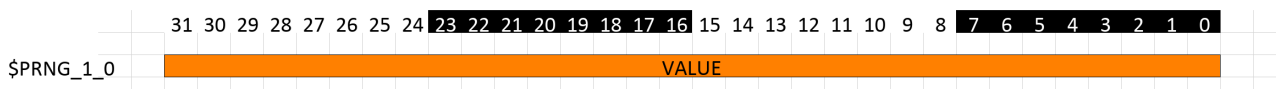


Fig. 2.22: \$PRNG_1_0 register format

Table 2.27: \$PRNG_1_0 register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvW	Value

2.7.5.2.18 \$PRNG_1_1

Register index: 262

The most significant 32-bits of \$PRNG_1.

Note: The value of this register is retained between worker exit and launch.

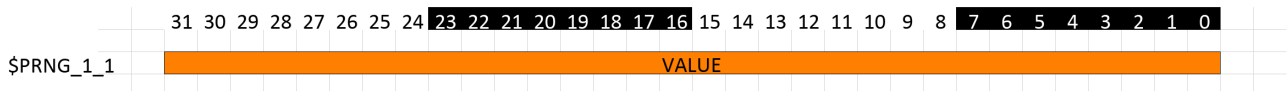


Fig. 2.23: \$PRNG_1_1 register format

Table 2.28: \$PRNG_1_1 register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	RvW	Value

2.7.5.2.19 \$PRNG_SEED

Register index: 263

Writes to this register cause the following assignments to the \$PRNG state:

- $\$PRNG_0_0 = value$
- $\$PRNG_0_1 = \sim value$
- $\$PRNG_1_0 = (value \ll 13) | (\sim value \gg 19)$
- $\$PRNG_1_1 = (\sim value \ll 13) | (value \gg 19)$

Reads return 0.

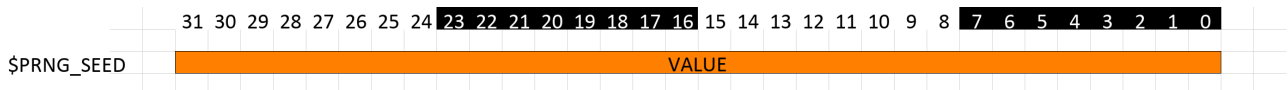


Fig. 2.24: \$PRNG_SEED register format

Table 2.29: \$PRNG_SEED register fields

Field name	Bit field	Reset value	Access semantics	Description
VALUE	[31:0]	0	WC	Value

2.7.5.2.20 \$TAS

Register index: 264

The `axy` scale(s) (or Temporary `amp` storage). The scalar value(s) used by the various `*axp(b)y` and `*raxb` instructions (`f32v2axy`, `f16v4mix` for example), (or temporary storage for `f32sisoamp`, `f32sisoslic`, `f16v8topk-instr-defn` and `f32v4topk-instr-defn`)

Note: The value of this register is *undefined* at worker launch

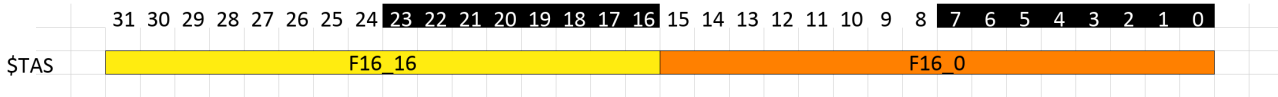


Fig. 2.25: `$TAS` register format

Table 2.30: `$TAS` register fields

Field name	Bit field	Reset value	Access semantics	Description
F16_0	[15:0]	0	<i>RvW</i>	<i>Half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_16	[31:16]	0	<i>RvW</i>	<i>Half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).

2.7.6 Pipeline internal state

2.7.6.1 `aux`

The `Aux` pipeline includes a small amount of per-context internal state, implicitly accessed by specific instructions. Access to this internal state is as described by the semantics of those instructions only. This state serves a very specific purpose and is not intended to store general instruction operand data.

Table 2.31: `Aux` pipeline internal state

Name	Size	Reset value	Description
<code>\$AACC[N]</code>	$N \times 32$ -bits	<code>TFPU_AACCRResetValue</code>	<p><i>Single-precision</i> floating-point internal accumulator state.</p> <p>This state is implicitly used by some <code>aux</code> floating-point instructions as an <i>accumulating</i> source and destination operand.</p> <p>This state can be explicitly initialised from and read into the <code>ARF</code> using the <code>f16v2gina</code> and <code>f32v2gina</code> instructions. In the case of <code>f16v2gina</code>, the single-precision accumulator values are automatically rounded and converted to <i>half-precision</i> format when read into the <code>ARF</code>.</p> <p>This state can also be explicitly forced to 0 using <code>\$FP_CLR.ZAACC</code>.</p>

Note: N is `TFPU_NUM_ACCUM`

`$AACC[N]` is implicitly used by the following instructions:

`f16v2cmac`, `f16v2gina`, `f16v4absacc`, `f16v4acc`, `f16v4cmac`, `f16v4gacc`, `f16v4hihoamp`, `f16v4hihoslic`, `f16v4hihov4amp`, `f16v4hihov4slic`, `f16v4istacc`, `f16v4mix`, `f16v4sihoamp`, `f16v4sihoslic`, `f16v4sisoamp`, `f16v4sisoslic`, `f16v4stacc`, `f16v8absacc`, `f16v8acc`, `f16v8sqacc`, `f32mac`, `f32sisoamp`, `f32sisoslic`, `f32sisov2amp`, `f32sisov2slic`, `f32v2aop`, `f32v2axy`, `f32v2gina`, `f32v2mac`, `f32v4absacc`, `f32v4acc`, `f32v4sqacc`

2.7.7 Common Compute Configuration State

The Common Compute Configuration State is *Tile* architectural state which is shared amongst all execution contexts. This state can only be configured by the *Supervisor* context and is used by the *Worker* contexts when performing certain *common* compute operations.

Common Compute Configuration State is configured via a 64-bit *common compute configuration space* using the *ld64putcs* and *ld128putcs* instructions. The configuration space provides access to the following shared configuration state elements:

2.7.7.1 Common Compute Configuration Space

Table 2.32: Collaborative Compute Configuration Space

Index	Register name	Reset value	Description
0x00	<i>\$CWEI_0_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x01	<i>\$CWEI_0_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x02	<i>\$CWEI_0_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x03	<i>\$CWEI_0_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x04	<i>\$CWEI_1_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x05	<i>\$CWEI_1_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x06	<i>\$CWEI_1_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x07	<i>\$CWEI_1_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x08	<i>\$CWEI_2_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x09	<i>\$CWEI_2_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x0A	<i>\$CWEI_2_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x0B	<i>\$CWEI_2_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x0C	<i>\$CWEI_3_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x0D	<i>\$CWEI_3_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x0E	<i>\$CWEI_3_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x0F	<i>\$CWEI_3_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x10	<i>\$CWEI_4_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x11	<i>\$CWEI_4_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x12	<i>\$CWEI_4_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x13	<i>\$CWEI_4_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x14	<i>\$CWEI_5_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x15	<i>\$CWEI_5_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x16	<i>\$CWEI_5_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x17	<i>\$CWEI_5_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x18	<i>\$CWEI_6_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x19	<i>\$CWEI_6_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x1A	<i>\$CWEI_6_2</i>	0x0	Common f16v4/f32v2 weight vector.
0x1B	<i>\$CWEI_6_3</i>	0x0	Common f16v4/f32v2 weight vector.
0x1C	<i>\$CWEI_7_0</i>	0x0	Common f16v4/f32v2 weight vector.
0x1D	<i>\$CWEI_7_1</i>	0x0	Common f16v4/f32v2 weight vector.
0x1E	<i>\$CWEI_7_2</i>	0x0	Common f16v4/f32v2 weight vector.

Continued on next page

Table 2.32 – continued from previous page

Index	Register name	Reset value	Description
0x1F	<code>\$CWEI_7_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x20	<code>\$CWEI_8_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x21	<code>\$CWEI_8_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x22	<code>\$CWEI_8_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x23	<code>\$CWEI_8_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x24	<code>\$CWEI_9_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x25	<code>\$CWEI_9_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x26	<code>\$CWEI_9_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x27	<code>\$CWEI_9_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x28	<code>\$CWEI_10_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x29	<code>\$CWEI_10_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x2A	<code>\$CWEI_10_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x2B	<code>\$CWEI_10_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x2C	<code>\$CWEI_11_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x2D	<code>\$CWEI_11_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x2E	<code>\$CWEI_11_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x2F	<code>\$CWEI_11_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x30	<code>\$CWEI_12_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x31	<code>\$CWEI_12_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x32	<code>\$CWEI_12_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x33	<code>\$CWEI_12_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x34	<code>\$CWEI_13_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x35	<code>\$CWEI_13_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x36	<code>\$CWEI_13_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x37	<code>\$CWEI_13_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x38	<code>\$CWEI_14_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x39	<code>\$CWEI_14_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x3A	<code>\$CWEI_14_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x3B	<code>\$CWEI_14_3</code>	0x0	Common f16v4/f32v2 weight vector.
0x3C	<code>\$CWEI_15_0</code>	0x0	Common f16v4/f32v2 weight vector.
0x3D	<code>\$CWEI_15_1</code>	0x0	Common f16v4/f32v2 weight vector.
0x3E	<code>\$CWEI_15_2</code>	0x0	Common f16v4/f32v2 weight vector.
0x3F	<code>\$CWEI_15_3</code>	0x0	Common f16v4/f32v2 weight vector.

2.7.7.1.1 `$CWEI_n_0`

Register index: $0 + (n * 4)$, $n \in \{0..15\}$

Shared f16v4/f32v2 weight vector. Reset to zero by the *Supervisor*. Initialised from *Tile Memory* by the *ld64putcs* and *ld128putcs* instructions.

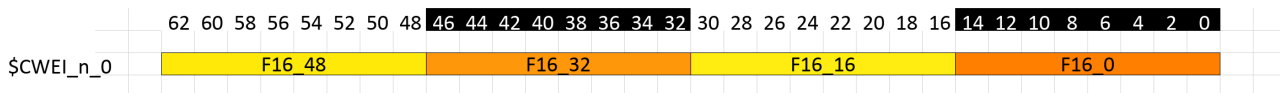


Fig. 2.26: \$CWEI_n_0 register format

Table 2.33: \$CWEI_n_0 register fields

Field name	Bit field	Reset value	Access semantics	Description
F16_0	[15:0]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_16	[31:16]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).
F16_32	[47:32]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_48	[63:48]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).

2.7.7.1.2 \$CWEI_n_1

Register index: $1 + (n * 4)$, $n \in \{0..15\}$

Shared f16v4/f32v2 weight vector.

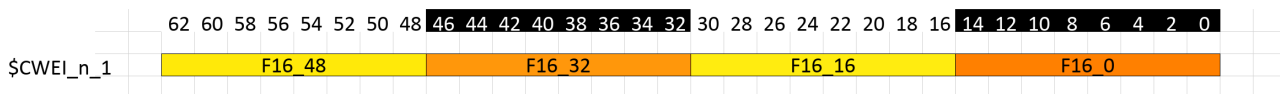


Fig. 2.27: \$CWEI_n_1 register format

Table 2.34: \$CWEI_n_1 register fields

Field name	Bit field	Reset value	Access semantics	Description
F16_0	[15:0]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_16	[31:16]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).
F16_32	[47:32]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_48	[63:48]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).

2.7.7.1.3 \$CWEI_n_2

Register index: $2 + (n * 4)$, $n \in \{0..15\}$

Shared f16v4/f32v2 weight vector.

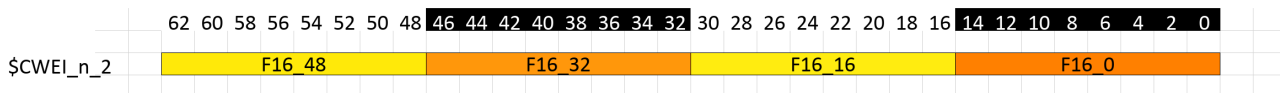


Fig. 2.28: \$CWEI_n_2 register format

Table 2.35: \$CWEI_n_2 register fields

Field name	Bit field	Reset value	Access semantics	Description
F16_0	[15:0]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_16	[31:16]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).
F16_32	[47:32]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_48	[63:48]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).

2.7.7.1.4 \$CWEI_n_3

Register index: $3 + (n * 4)$, $n \in \{0..15\}$

Shared f16v4/f32v2 weight vector.

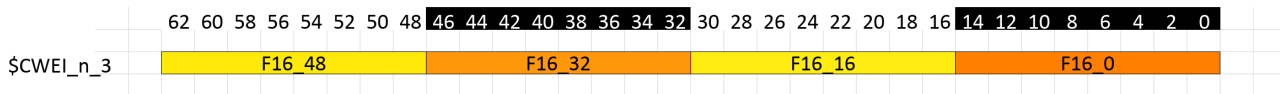


Fig. 2.29: \$CWEI_n_3 register format

Table 2.36: \$CWEI_n_3 register fields

Field name	Bit field	Reset value	Access semantics	Description
F16_0	[15:0]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_16	[31:16]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).
F16_32	[47:32]	0	WO	CWEI <i>half-precision</i> floating-point value (or bottom 16-bits of a <i>single-precision</i> value).
F16_48	[63:48]	0	WO	CWEI <i>half-precision</i> floating-point value (or top 16-bits of a <i>single-precision</i> value).

2.8 Memory model

2.8.1 Overview

Each *Tile* instance includes a tightly coupled local memory used to store all code and data, for all execution contexts. The *Tile* instance local memory is the only memory directly accessible by *Tile* instruction streams.

The *Tile* architecture features a common, contiguous unsigned 21-bit address space, beginning at address 0x0 and every context has visibility of the entire space.

Implementations of *Tile* are free to populate only part of this space, with the populated *Tile Memory* area presented as a contiguous unsigned address range from `TMEM_BASE_ADDR` to $(\text{TMEM_BASE_ADDR} + (\text{TMEM_SIZE} * 1024) - 1)$ (inclusive).

Unless explicitly stated otherwise, *Tile* address calculations are performed using unsigned 32-bit integer arithmetic and attempts to access an address outside the implemented address space will cause an *exception event*⁴. *Tile Memory* accesses must be naturally aligned to the payload size. Misaligned accesses will result in an *exception event*.

Tile Memory is composed of an *implementation dependent* number of 64-bit wide memory *elements*. The capacity of the individual *Tile Memory* elements is also *implementation dependent* and given by `TMEM_ELEMSIZE`.

In addition, *Tile Memory* may be divided into an *implementation dependent* number of *Memory regions*. Each memory region:

- is composed of a whole number of memory elements
- has a base address which is aligned to the memory element size (*TMem_RegionBaseAddress*)
- spans a single contiguous address range inside $[\text{TMEM_REGION0_BASE_ADDR}, (\text{TMEM_BASE_ADDR} + \text{TMEM_SIZE} - 1)]$ (where $\text{TMEM_REGION0_BASE_ADDR} \leq \text{TMEM_BASE_ADDR}$)
- may logically *interleave* its memory elements differently to other memory regions (*TMem_RegionInterleaveFactor*)
- may or may not be addressable for Instruction fetch (*TMem_RegionIsExecutable*)

Memory accesses are fully pipelined and due to the time-multiplexed arrangement of context execution, *Worker* contexts experience only a single execution cycle memory latency.

Note: The region with the smallest base address (region 0) is special in that its base address `TMEM_REGION0_BASE_ADDR` may or may not be equal to `TMEM_BASE_ADDR`. When they differ (in which case $\text{TMEM_REGION0_BASE_ADDR} < \text{TMEM_BASE_ADDR}$), the discrepancy is due to the first memory element in region 0 being only partially populated, with the populated area occupying the upper part of the memory-element's address space. Attempted accesses to the unpopulated address range $[\text{TMEM_REGION0_BASE_ADDR}, \text{TMEM_BASE_ADDR} - 1]$ will result in an *exception event*.

The existence of such an unpopulated region has no effect on the *Tile Memory* access semantics for the valid address region (see *TMem_IsValidAddress*).

2.8.2 Address format

Tile Memory is byte addressed and the architectural address range spans a contiguous 21-bit space from 0x0. A full absolute *Tile Memory* address is effectively composed of:

- a *memory region* identifier
- a memory-element identifier
- a byte offset within the memory-element.

The mapping from *Tile Memory* address to memory-region, memory-element and offset is dependent on the interleave factor associated with the address, as well as the ordering and size of the memory-regions within the address space.

See *TMem_RegionId*, *TMem_ElementId* and *TMem_ElementOffset* for the mapping definitions.

⁴ Address range checking is as if the calculated effective address is an unsigned 32-bit value. If an effective address calculation produces a result that would overflow an unsigned 32-bit address, the overflow will not be detected.

2.8.2.1 Pointers

2.8.2.1.1 Full

Full pointers are unsigned, 32-bit absolute byte address values. When loaded, a full pointer occupies a single *MRF* register. The 21-bit address range of *Tile* implies that the top 11-bits of a valid full, 32-bit pointer will always be zero.

Post-increments on full pointers are expressed in terms of bytes.

2.8.2.1.2 Packed

Packed pointers are unsigned, 21-bit absolute byte address values. *Tile* has explicit support for packing up to 3 such pointer values into a *MRF* register pair (see *tapack*). Various load and store instructions use post-incrementing packed pointers (*ldst64pace* for example).

Post-increments on packed pointers are expressed in terms of bytes.

2.8.2.2 Delta Offsets

Delta offsets (also referred to as delta pointers) provide a relative, byte addressed, short form of address offset. Instructions that support delta offsets allow any *MRF* based full pointer value to be specified as the base address for the specified delta. The effective address of the memory access will be the full pointer base address added to the delta.

Generally speaking delta offsets can be any bit size, up to 32 (see *ld32* for example), although some instructions do limit the size to exactly 16-bits (see *ldd16v2a32* for example).

Post increments on delta offsets are expressed in terms of bytes.

2.8.2.3 Mini deltas

Mini delta values provide a short range, relative, scaled form of offset. Mini deltas are 4-bit offset values, with up to 8 such deltas being packed into a 32-bit *word*.

See *ldb16b16*.

2.8.3 Endianness

Tile uses the **little-endian** byte ordering representation.

2.8.4 Memory regions

A memory region is a single, contiguous logical address range within *Tile Memory* with specific access characteristics. Typically those characteristics differ to those of neighbouring memory regions. Memory regions are always composed of a whole number of memory elements.

The following characteristics may vary between memory regions:

- Memory element interleave factor

A memory region may interleave the memory elements that together form that particular area of memory.

When non-interleaved, the memory elements are arranged linearly such that each single memory element spans a contiguous memory-element sized area of memory.

When n -way interleaved, n memory elements are grouped (address wise) in such a way that n consecutive 64-bit logical memory locations span the n distinct memory elements. In this arrangement, simultaneous accesses to n consecutive 64-bit memory locations are guaranteed not to *clash* (see *Memory clashes*). For example, a 2-way interleaved memory region allows for the 2 x 64-bit aligned addresses a and $a + 8$ to be accessed simultaneously. Such simultaneous accesses would cause a memory clash in a non-interleaved memory region (unless the two addresses happened to straddle a memory-element boundary).

An n -way interleaved memory region has an interleave factor of n . A non-interleaved memory region is 1-way interleaved and so has an interleave factor of 1.

- Executability

Not all memory regions are accessible for Instruction fetch (see *Instruction fetch semantics*).

2.8.5 Memory Map

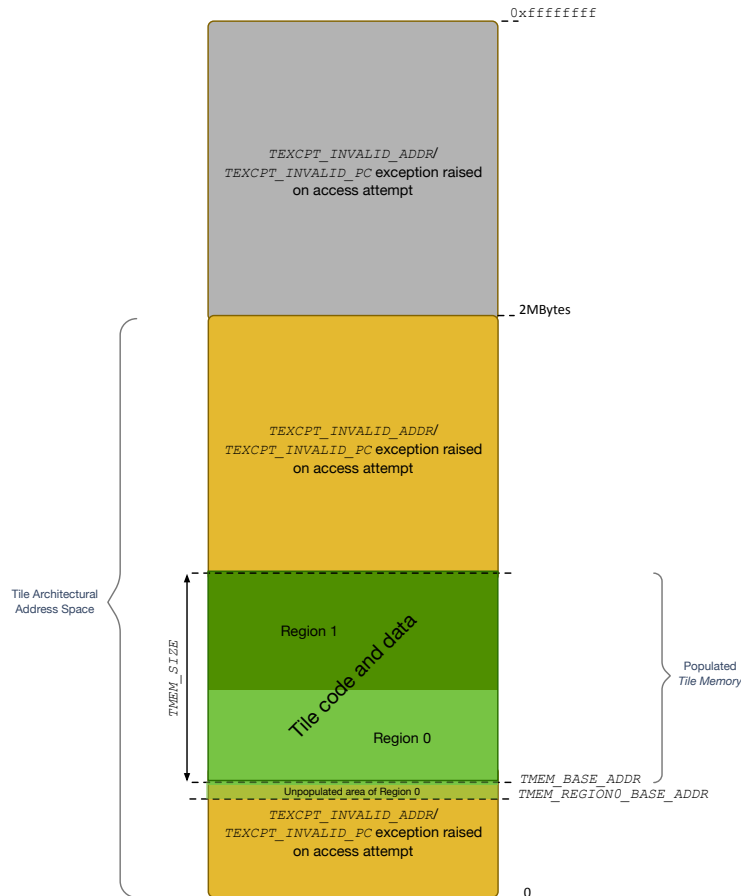


Fig. 2.30: Tile local memory map

See *Memory parameters* for variable definitions.

2.8.6 Data access sizes

Tile provides data load instructions for the following data sizes:

- 8-bit
- 16-bit
- 32-bit
- 64-bit
- 128-bit (for accesses to memory regions with an *interleave factor* of at least 2 only (see *Memory regions*))

Tile provides data store instructions for the following data sizes:

- 32-bit
- 64-bit

All load and store effective addresses must be naturally aligned. Misaligned accesses will result in an *exception event*.

Tile also provides instructions that perform multiple simultaneous loads as well as simultaneous load and store instructions.

2.8.7 Buffering

There is no architecturally visible buffering of data or instructions between the Tile processor and Tile Memory. Most implementations will implement some amount of architecturally invisible buffering. Those affecting the

programming model will be documented in the *implementation specifics* section.

2.8.8 Memory Protection

Tile provides no hardware mechanisms for memory protection.

2.8.9 Memory access semantics

2.8.9.1 Instruction fetch semantics

Instructions must be naturally aligned in *Tile Memory* and reside within an executable memory region. A *TEXCPT_INVALID_PC exception event* will be raised if a *Control* instruction attempts to set *PC* to a non 4-byte aligned address. A *TEXCPT_INVALID_PC exception event* will also be raised if a control instruction attempts to set *PC* to an address outside the logical address range of *Tile Memory*, or a memory region that is inaccessible for Instruction fetch.

In order to effectively feed the dual pipelines of *Tile*, implementations will typically fetch instructions at a rate of 2 instructions per cycle (8-bytes), from 8-byte aligned addresses. In addition, architecturally invisible internal instruction buffers may be used to manage solo instruction or *Execution Bundle* issue.

The following *pseudo-code* serves to illustrate the instruction fetch semantics:

Listing 2.5: Instruction Fetch Semantics

```
// Copyright (c) 2017-2022 Graphcore Ltd. All rights reserved.
(uint64_t, int, int)InstructionFetch() {
    // $PC is always 4-byte aligned, since the bottom two bits
    // of the register are RESERVED (and therefore read as 0).
    // Instruction fetches are always 8-byte aligned
    //
    uint32_t alignedAddress = $PC & ~0x7;

    if (!TMem_IsValidAddress(alignedAddress)) {
        // An exception will be raised if $PC represents an address that is below
        // TMem_BASE_ADDR
        EXCEPT(TEXCPT_INVALID_PC);
    } else if (!TMem_AddressIsExecutable(alignedAddress)) {
        // An exception will be raised if $PC represents an address in
        // a non-executable memory region
        EXCEPT(TEXCPT_INVALID_PC);
    } else {
        // $PC is effectively [MemoryElementId|OffsetWithinElement]
        unsigned elemId = TMem_ElementId(alignedAddress);
        unsigned eOffset = TMem_ElementOffset(alignedAddress);
        int nextinstr = ($PC >> 2) & 1;

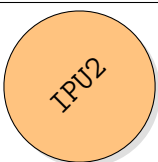
        // Select the memory element
        TMemElement &element = TileMemory[elemId];

        // Perform the 64-bit, naturally aligned memory access
        uint64_t instructionBundle = element[eOffset];

        return (instructionBundle, nextInstr, elemId);
    }
}
```

Function references: *TMem_IsValidAddress*, *TMem_AddressIsExecutable*, *TMem_ElementId*, *TMem_ElementOffset*

Implementation detail



Due to the strict 8-byte alignment of memory accesses for instruction fetch, branching to an Execution Bundle that is 4-byte but not 8-byte aligned imposes a single-cycle penalty (versus branching to an 8-byte aligned Execution Bundle).

2.8.9.2 Data load/store semantics

Precise data load and store semantics are provided by the individual load/store instruction definitions (see *Memory*).

The following behaviour applies to all load and store instructions:

- Data accesses must be to naturally aligned addresses, appropriate to the access size. A *TEXCPT_INVALID_ADDR exception event* will be raised by any load/store instruction which attempts to make a misaligned data access.
- Data access addresses must be within the valid logical address bounds defined by the implementation. A *TEXCPT_INVALID_ADDR exception event* will be raised by any load/store instruction which attempts to make an out-of-bounds data access.
- Load accesses which are narrower than 32-bits will be extended to 32-bits in a manner determined by the precise instruction semantics.
- If a load or store instruction attempts to make more than 1 data access, the effective addresses must be to separate memory elements. A *TEXCPT_CONFLICT exception event* will be raised by any load/store instruction which attempts multiple accesses to the same memory element.
- Exceptions raised by store instructions or by *Execution Bundles* containing store instructions do not squash the write to memory.

The following *pseudo-code* serves to illustrate the data load semantics for a single load access. See *Memory* for the full semantics.

Listing 2.6: Data load Semantics

```
// Copyright (c) 2017-2022 Graphcore Ltd. All rights reserved.
uint64_t DataLoad(uint32_t eaddr,
                 int accessSizeInBytes,
                 bool signNotZeroExtend,
                 int zeroTailShift) {
    if ((eaddr & ((accessSizeInBytes)-1)) != 0) {
        // Memory accesses must be naturally aligned
        EXCEPT(TEXCPT_INVALID_ADDR);
    } else if (!TMem_IsValidAddress(eaddr)) {
        // An exception will be raised if eaddr is beyond the logical memory bounds
        EXCEPT(TEXCPT_INVALID_ADDR);
    } else {
        // Perform 8-byte memory access
        uint32_t alignedAddress = eaddr & ~0x7;
        unsigned elemId = TMem_ElementId(alignedAddress);
        unsigned eOffset = TMem_ElementOffset(alignedAddress);

        // Select the memory element
        TMemElement &element = TileMemory[elemId];

        // Extract 8-bytes from the memory element
        uint64_t octabyte = element[eOffset];

        // Pick the relevant data portion - leaving the msbs as 0
        uint64_t data = (octabyte >> (8 * (eaddr & 0x7))) & ((1 << (accessSizeInBytes * 8)) - 1);

        // Sign extend if necessary
        if (signNotZeroExtend) {
            data <- SignExtend32((data << zeroTailShift), (accessSizeInBytes * 8) + zeroTailShift);
        }

        return data;
    }
}
```

Function references: *TMem_IsValidAddress*, *TMem_ElementId*, *TMem_ElementOffset*

2.8.10 Memory system errors

Tile Memory typically implements hardware error detection or error detection with correction techniques, such as parity checking or ECC.

2.8.10.1 Uncorrectable Errors

The presence of an uncorrectable error is regarded as terminal and will result in a `TEXCPT_MEMERR` type *exception event*.

There are a number of situations where an uncorrectable memory error may be detected:

1. The read memory access resulting from an instruction or Execution Bundle fetch
2. The read memory access(es) performed explicitly by a single or multi-load instruction
3. The read memory access(es) performed explicitly by a load-store instruction

For case 1 the error is detected between the IBRK and DBRK checks of the instruction execution (see *Instruction execution semantics*) (i.e. no DBRK condition check is performed in the presence of an uncorrectable instruction fetch memory error and the instruction doesn't enter the *Except In* phase).

For cases 2 and 3, for the context that is performing the load operation, the exception raise behaviour is as for any other instruction based *exception event*, with the memory error taking priority (i.e. a `TEXCPT_MEMERR` *exception event* will be raised as if immediately before the *Exception-check* phase of the instruction).

For load-store instructions, as for other instruction based *exception events*, the store will be committed to memory.

2.8.11 Memory clashes

In order to:

1. provide the ability for *Tile* to access multiple memory locations simultaneously (required for performance)
2. provide those accesses with a strictly fixed latency (required for determinism)

Tile Memory is arranged as an *implementation dependent* number of simultaneously accessible *independent* memory-elements. The only architectural access restriction being that simultaneous memory accesses cannot target the same memory element (a clash). An implementation may impose further restrictions.

This section describes the potential sources of such memory clashes and the resulting behaviour.

2.8.11.1 Instruction stream based

Any memory instruction which accesses multiple address locations must ensure that those effective addresses map to distinct memory-elements, as defined by `TMem_ElementId`.

Any memory-element clash between the accesses of a multi-load/store instruction will be detected by the hardware and result in a `TEXCPT_CONFLICT` *exception event*.

Specific implementations of the *Tile* architecture may place additional restrictions on simultaneous accesses to *Tile Memory* from the instruction stream. See *Implementation/Memory clashes* for further details.

2.8.11.2 Exchange based

Note: The details provided by this section are not relevant to the *Tile Vertex ISA*.

2.9 Floating Point Unit

2.9.1 Overview

Tile supports operations on a number of floating-point *number formats*, for both *scalar* and short *vector* variables. Operands for floating-point instructions are provided by the *ARF* and floating-point results are written to the *ARF*.

Tile's support for floating-point is based on [IEEE754], both in terms of the storage formats and in the accuracy of calculations.

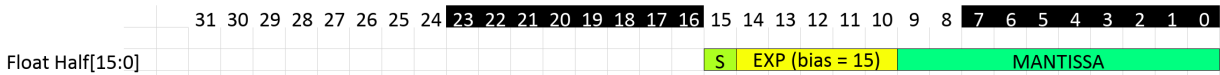
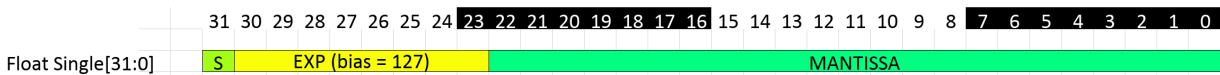
2.9.2 Number formats

2.9.2.1 Scalars

Tile provides direct support for the following scalar floating-point number formats:

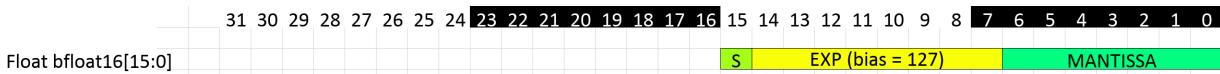
Table 2.37: Floating-point scalar formats

Common name	IEEE 754-2008	Size	Sign-bit	Exponent	Significand	Zero offset
Single-precision	binary32	32-bits	31	[30:23]	[22:0]	127
Half-precision	binary16	16-bits	15	[14:10]	[9:0]	15



Note: All formats have an assumed lead bit with value 1 unless the exponent field is 0.

Note: *Tile* doesn't provide direct support for the 16-bit bfloat16 *truncated single* format (8-bit exponent, 7-bit significand, Zero offset of 127). Use of this storage format is possible but requires explicit (*zero tailed*) conversion from/to full *single-precision*.



2.9.2.1.1 Single Precision Constants

Listing 2.7: Single Precision Constants

```

/*****
 * Single.h: Shared defines for Single precision floating point
 *
 * Copyright (c) 2020-2022 Graphcore Ltd. All rights reserved.
 *
 *****/
#ifndef _ciss_Single_h_
#define _ciss_Single_h_

/**
 * @file
 * @brief MACROS for manipulating the raw bit format of single-precision values.
 */

#define _SINGLE_MANT_SHIFT (0)

```

```

#define _SINGLE_MANT_SIZE (23)
#define _SINGLE_MANT_MASK ((1 << _SINGLE_MANT_SIZE) - 1)
#define _SINGLE_EXP_SHIFT _SINGLE_MANT_SIZE
#define _SINGLE_EXP_SIZE (8)
#define _SINGLE_EXP_MASK ((1 << _SINGLE_EXP_SIZE) - 1)
#define _SINGLE_MAX_EXP _SINGLE_EXP_MASK
#define _SINGLE_SIGN_SHIFT (_SINGLE_EXP_SHIFT + _SINGLE_EXP_SIZE)
#define _SINGLE_Q_SHIFT (_SINGLE_EXP_SHIFT - 1)
#define _SINGLE_BIAS (127)
#define _SINGLE_EXP(v) (((v) >> _SINGLE_EXP_SHIFT) & _SINGLE_EXP_MASK)
#define _SINGLE_MANT(v) (((v) >> _SINGLE_MANT_SHIFT) & _SINGLE_MANT_MASK)
#define _SINGLE_SIGN(v) (((v) >> _SINGLE_SIGN_SHIFT) & 1)
#define _SINGLE_IS_NEG(v) (_SINGLE_SIGN(v) != 0)
#define _SINGLE_IS_ZERO(v) ((_SINGLE_EXP(v) == 0) && (_SINGLE_MANT(v) == 0))
#define _SINGLE_IS_SUBNORM(v) ((_SINGLE_EXP(v) == 0) && (_SINGLE_MANT(v) != 0))
#define _SINGLE_IS_INFINITY(v) ((_SINGLE_EXP(v) == _SINGLE_MAX_EXP) && (_SINGLE_MANT(v) == 0))
#define _SINGLE_IS_NAN(v) ((_SINGLE_EXP(v) == _SINGLE_MAX_EXP) && (_SINGLE_MANT(v) != 0))
#define _SINGLE_IS_QNAN(v) (_SINGLE_IS_NAN(v) && (((v) >> _SINGLE_Q_SHIFT) & 1) == 1)
#define _SINGLE_IS_SNAN(v) (_SINGLE_IS_NAN(v) && (((v) >> _SINGLE_Q_SHIFT) & 1) == 0)
#define _SINGLE_INFINITY (_SINGLE_MAX_EXP << _SINGLE_EXP_SHIFT)
#define _SINGLE_NEG_INFINITY ((1 << _SINGLE_SIGN_SHIFT) | _SINGLE_INFINITY)

#endif // _ciss_Single_h_

```

2.9.2.1.2 Half Precision Class

Listing 2.8: Half Precision Class

```

/*****
 * Half.h: Half-precision floating-point abstraction for Colossus
 *      Instruction Set Simulator
 *
 * Copyright (c) 2015-2022 Graphcore Ltd. All rights reserved.
 *
 * Notes:
 *   Most operations are performed by:
 *   1 Converting the half-precision value to single-precision (float)
 *   2 Performing the operation at single-precision accuracy
 *   3 Converting the result back to half-precision
 *
 *****/

#ifndef _ciss_Half_h_
#define _ciss_Half_h_

/**
 * @file
 * @brief Half-precision floating-point abstraction.
 */

#include <stdint.h>

#include "colossus/tileimplconsts.h"

/*
 * MACROS for manipulating the raw bit format of half-precision values
 */
#define HALF_MANT_SHIFT (0)
#define HALF_MANT_SIZE (10)
#define HALF_MANT_MASK ((1 << HALF_MANT_SIZE) - 1)
#define HALF_EXP_SHIFT HALF_MANT_SIZE
#define HALF_EXP_SIZE (5)
#define HALF_EXP_MASK ((1 << HALF_EXP_SIZE) - 1)
#define HALF_MAX_EXP HALF_EXP_MASK
#define HALF_SIGN_SHIFT (HALF_EXP_SHIFT + HALF_EXP_SIZE)
#define HALF_Q_SHIFT (HALF_EXP_SHIFT - 1)
#define HALF_BIAS (15)
#define HALF_EXP(v) (((v) >> HALF_EXP_SHIFT) & HALF_EXP_MASK)
#define HALF_MANT(v) (((v) >> HALF_MANT_SHIFT) & HALF_MANT_MASK)
#define HALF_SIGN(v) (((v) >> HALF_SIGN_SHIFT) & 1)
#define HALF_IS_NEG(v) (HALF_SIGN(v) != 0)
#define HALF_IS_ZERO(v) ((HALF_EXP(v) == 0) && (HALF_MANT(v) == 0))
#define HALF_IS_SUBNORM(v) ((HALF_EXP(v) == 0) && (HALF_MANT(v) != 0))
#define HALF_IS_INFINITY(v) ((HALF_EXP(v) == HALF_MAX_EXP) && (HALF_MANT(v) == 0))

```



```

#define HALF_IS_NAN(v) ((HALF_EXP(v) == HALF_MAX_EXP) && (HALF_MANT(v) != 0))
#define HALF_IS_QNAN(v) (HALF_IS_NAN(v) && (((v >> HALF_Q_SHIFT) & 1) == 1))
#define HALF_IS_SNAN(v) (HALF_IS_NAN(v) && (((v >> HALF_Q_SHIFT) & 1) == 0))
#define HALF_INFINITY (HALF_MAX_EXP << HALF_EXP_SHIFT)
#define HALF_NEG_INFINITY ((1 << HALF_SIGN_SHIFT) | HALF_INFINITY)
#define HALF_MAX ((30 << HALF_EXP_SHIFT) | (HALF_MANT_MASK << HALF_MANT_SHIFT))

/*
 * Fixed graphcore f16 qNaN bitcode
 */
#define HALF_GCNAN_MANT (0x2ce)
#define HALF_GCNAN ((uint16_t)((HALF_EXP_MASK << HALF_EXP_SHIFT) | HALF_GCNAN_MANT))
namespace ciss {

class Half {

public:
    /**
     * Initialise from a single-precision fp value
     */
    Half(float value);

    /**
     * Initialise as zero
     */
    Half();

    /**
     * Initialise from a raw 16-bit pattern (conforming to IEEE 754-2008
     * binary16 format)
     */
    explicit Half(uint16_t bitPattern);

    /**
     * Type-cast to single-precision
     */
    operator float() const {
        return single;
    }

    /**
     * Obtain half-precision bit-pattern
     * @param smode Specified saturation mode
     * @returns raw 16-bit saturated bit-pattern. Saturation as per c_satMode
     */
    uint16_t bit16(HalfSaturationMode_t smode) const;

    /**
     * Obtain half-precision bit-pattern
     * @param smode Specified saturation mode
     * @returns raw zero-extended saturated 16-bit bit-pattern. Saturation as per c_satMode
     */
    uint32_t bit32(HalfSaturationMode_t smode) const {
        return (uint32_t)bit16(smode);
    }

    // Destructive operators
    Half &operator=(const Half &other);
    Half &operator+=(const float &other);
    Half &operator-=(const float &other);
    Half &operator*=(const float &other);
    Half &operator/=(const float &other);

    Half &log(const float &other);

    /**
     * Set to HALF_GCNAN
     */
    Half &setGCSNaN();

    /**
     * Get sign
     */
    bool sign() const;

```

```

/**
 * Check if qNaN
 */
bool isqNaN() const;

/**
 * Check if sNaN
 */
bool issNaN() const;

/**
 * Check if sNaN/qNaN
 */
bool isNaN() const;

/**
 * Check if infinity
 */

bool isInf() const;

/**
 * Check if finite
 */
bool isFinite() const;

/**
 * Check if normalized
 */
bool isNorm() const;

/**
 * Check if zero
 */
bool isZero() const;

/**
 * Check if subnorm
 */
bool isDenorm() const;

// What's the maximum representable finite number in half-precision?
static float max();

private:
uint16_t ihalf;
float single;

uint16_t toHalf(float value);

// Produce a rounded mantissa value -
// using round-to-nearest, ties-to-even rounding only
uint16_t roundMantissa(uint32_t mant, int *overflow, uint32_t extraLSBits);
}; // class Half

/**
 * Extract a half-precision value from the top or bottom 16-bits of a 32-bit input value.
 * @param value Single precision input value.
 * @param zeroOrOne Extract top or bottom 16-bits.
 * @param preserveInf Preserve infinities and don't convert them to qNaNs.
 * @returns Half-precision value extracted from input value.
 */
Half pickHalf(uint32_t value, int zeroOrOne, bool preserveInf = false);
} // namespace ciss

#endif // _ciss_Half_h_

```

2.9.2.2 Vectors

Tile supports the following vector floating-point formats:

Table 2.38: Floating-point vector formats

Base type	Vector size	Registers/vector	Comments	Example
<i>Single-precision</i>	2 elements	2		<i>f32v2add</i>
<i>Single-precision</i>	4 elements	4	Supported as a source operand type for certain instructions involving the <i>accumulator</i> state only	<i>f32v4acc</i>
<i>Half-precision</i>	2 elements	1	16-bit <i>Half-precision</i> values are packed into the upper and lower halves of each register	<i>f16v2add</i>
<i>Half-precision</i>	4 elements	2	16-bit <i>Half-precision</i> values are packed into the upper and lower halves of each register	<i>f16v4add</i>
<i>Half-precision</i>	8 elements	4	Supported as a source operand type for certain instructions involving the <i>accumulator</i> state only	<i>f16v8acc</i>

See *Floating-point operations x number format and vector length* for a table detailing which operations are available for each vector format.

Note: Support for half-precision scalars is provided via the 2-element half-precision vector operations, with the scalar input operands duplicated into the top-half of their 32-bit registers. The required duplication occurs automatically during 16-bit load and format conversions (see *ldb16* and *f32tof16* for examples)

Note: For vector formats that span multiple general purpose registers:

- The index of the first register in a vector group must be naturally aligned, based on the number of registers occupied per vector
- The indices of the registers within a single vector form a contiguous range

2.9.3 Control and Status Registers

The floating-point control and status registers reside within the Worker context, upper CSR address space (*Control and Status registers*). They are accessed using *uput* and *uget*.

Table 2.39: Floating-point control and status registers

Register	Description
<i>\$FP_CTL</i>	Floating point control register. Used to (un)mask floating-point exceptions, and set the rounding mode (including stochastic rounding).
<i>\$FP_STS</i>	Floating point status register. Provides floating-point exception status.
<i>\$FP_CLR</i>	Floating point <i>clear</i> register. Used to clear floating-point exception flags and to zero the accumulator state.

Note: The *Worker* context floating-point control register *\$FP_CTL* is initialised from the *Supervisor* control registers *\$FP_ICTL* during execution of the *run* and *runall* instructions. *run* and *runall* also clear all floating point exception flags in the target *Worker*.

2.9.4 General Accuracy

2.9.4.1 Single-precision

Tile implements strict *denorms-are-zero* and *flush-denorms-to-zero* policies for all operations involving *single-precision* values:

- All single-precision inputs within the denorm range are treated as ± 0.0 (preserving the sign of the input)
- *Tile* will never produce a single-precision result within the denorm range. If the operation would have produced a single-precision value within the denorm range, assuming the calculation was performed to infinite precision and rounded appropriately, *Tile* will produce ± 0.0 (preserving the sign of the result)

Denorm behaviour aside and subject to the supported rounding modes (see *Rounding modes*), all *single-precision* operations (i.e. those operating on and producing *Single-precision* values) conform to [IEEE754], unless explicitly stated in *IEEE 754-2008 caveats and differences*.

2.9.4.2 Half-precision

Operations on *half-precision* values are based on the guiding principals of [IEEE754], with some significant differences related to overflow behaviour and infinity values:

- *Tile* will never produce $\pm\infty$ as the *half-precision* result of an arithmetic operation.
- *Tile* treats $\pm\infty$ input values to arithmetic operations in the same way as signaling NaNs. That is:
 - When the operation yields a floating-point result, that result will always be a quiet NaN
 - The invalid operation flag, `$FP_STS.INV` will be set to 0b1.
- In default mode (`$FP_CTL.NANOO == 0b0`), an arithmetic operation that has overflowed the *half-precision* range will saturate to ± 65504 instead (and set the overflow flag `$FP_STS.OFLO`).
- In NAN On Overflow mode (`$FP_CTL.NANOO == 0b1`), an arithmetic operation that has overflowed the *half-precision* range will produce a quiet NaN and set both the overflow flag (`$FP_STS.OFLO`) and the invalid operation flag (`$FP_STS.INV`).

Conversions from a higher precision infinity to *half-precision*, whether explicit (using *f32tof16*) or implicit (using *f16v4hihoamp* for example) will produce a quiet NaN result and set the invalid operation flag.

Comparison and min/max type operations treat input $\pm\infty$ as per [IEEE754] `minNum`/`maxNum` (min, max and clamp can therefore produce infinity results, unlike arithmetic operations).

2.9.5 Rounding modes

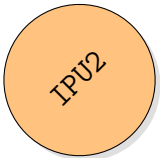
The *Tile* architecture defines the following rounding modes for floating-point operations. Not all rounding modes are supported by all implementations (see *Implementation Specifics*).

Rounding modes are configured using a combination of `$FP_CTL.RND` and `$FP_CTL.ESR`

Table 2.40: Floating-point rounding modes

Rounding mode	Implementation parameter	Description
Round to nearest, ties to even	TFPU_ROUND_EVEN_VALID	The floating-point number nearest to the infinitely precise result is returned; if the two nearest floating-point numbers are equally near, the one with zero as the least significant bit is returned.
Round to nearest, ties to away	TFPU_ROUND_AWAY_VALID	The floating-point number nearest to the infinitely precise result is returned; if the two nearest floating-point numbers are equally near, the one with the largest magnitude is returned.
Round toward positive infinity	TFPU_ROUND_POSINF_VALID	the result is the floating-point number closest to and no less than the infinitely precise result
Round toward negative infinity	TFPU_ROUND_NEGINF_VALID	the result is the floating-point number closest to and no greater than the infinitely precise result
Round toward Zero	TFPU_ROUND_ZERO_VALID	the result is the floating-point number closest to and no greater in magnitude than the infinitely precise result.
Stochastic rounding	TFPU_ROUND_STOCH_VALID	Stochastic rounding applies to a subset of floating-point instructions only. For those instructions, stochastic rounding is enabled/disabled via <code>\$FP_CTL.ESR</code> . When enabled, stochastic rounding is performed (by those instructions that support it) in preference to the rounding mode specified by <code>\$FP_CTL.RND</code> . See <i>Stochastic Rounding</i> for further details.

Implementation detail



Implements only Round to nearest, ties to even and Stochastic rounding modes.

2.9.6 Format Conversion and Transformations

Tile supports the following number format conversion and transformation operations:

Table 2.41: Format conversions

Source format	Result format	Instruction	Description
<i>Single-precision</i>	<i>Single-precision</i>	<code>f32int</code>	Round a <i>Single-precision</i> value to a single-precision integral. Rounding as per the mode specified by an instruction immediate (<code>\$FP_CTL.RND</code> has no effect on this instruction). Zero and ∞ operands are converted to zero/ ∞ results of the same sign.

Continued on next page

Table 2.41 – continued from previous page

Source format	Result format	Instruction	Description
<i>Single-precision</i>	Signed 32-bit integer	<i>f32toi32</i>	Convert a <i>Single-precision</i> value to a signed 32-bit integer. Rounding as per the current mode specified by <i>\$FP_CTL.RND</i> . NaN and ∞ operands will result in the setting of <i>\$FP_STS.FP_INV</i> . <i>\$FP_STS.FP_INV</i> will also be set if the source value is beyond the range of a signed 32-bit integer.
<i>Single-precision</i>	Unsigned 32-bit integer	<i>f32toui32</i>	Convert a <i>Single-precision</i> value to an unsigned 32-bit integer. Rounding as per the current mode specified by <i>\$FP_CTL.RND</i> . NaN and ∞ operands will result in the setting of <i>\$FP_STS.FP_INV</i> . <i>\$FP_STS.FP_INV</i> will also be set if the source value is beyond the range of an unsigned 32-bit integer.
Signed 32-bit integer	<i>Single-precision</i>	<i>f32fromi32</i>	Convert a signed 32-bit integer to <i>Single-precision</i> floating-point. Values in the range $[-2^{24}, 2^{24}]$ are converted exactly. For other values, rounding is applied as per <i>\$FP_CTL.RND</i> .
Unsigned 32-bit integer	<i>Single-precision</i>	<i>f32fromui32</i>	Convert an unsigned 32-bit integer to <i>Single-precision</i> floating-point. Values in the range $[0, 2^{24}]$ are converted exactly. For other values, rounding is applied as per <i>\$FP_CTL.RND</i> .
<i>Single-precision</i>	<i>Half-precision</i>	<i>f32tof16</i>	Convert a <i>Single-precision</i> value to <i>Half-precision</i> : <ul style="list-style-type: none"> • Signaling NaNs are converted to the <i>Tile fixed-value f16 quiet NaN</i> and set <i>\$FP_STS.INV</i> to 0b1 • Quiet NaNs are converted to the <i>Tile fixed-value f16 quiet NaN</i> (but don't set <i>\$FP_STS.INV</i>) • $\pm\infty$ are treated as per signaling NaNs • ± 0 map to ± 0 • Otherwise: <ul style="list-style-type: none"> – If enabled, stochastic rounding is applied to the single-precision input value (see <i>Stochastic Rounding</i>) – Otherwise, rounding is applied as per <i>\$FP_CTL.RND</i> – Rounded values with magnitude less than 2^{-24} map to ± 0 – Rounded values with magnitude in the range $[2^{-24}, 2^{-14} - 2^{-24}]$ map to <i>Half-precision</i> denorms (with rounded significand) – Rounded values with magnitude in the range $[2^{-14}, 65504]$ map to normalized <i>Half-precision</i> values (with rounded significand) – Rounded values with magnitude exceeding 65504 map to: <ul style="list-style-type: none"> * ± 65504 (the maximum representable value) and set <i>\$FP_STS.OFLO</i> to 0b1, if <i>\$FP_CTL.NANOO</i> is 0, * the <i>Tile fixed-value f16 quiet NaN</i> if <i>\$FP_CTL.NANOO</i> is 1. In this case, both <i>\$FP_STS.OFLO</i> and <i>\$FP_STS.INV</i> are set to 0b1

Continued on next page

Table 2.41 – continued from previous page

Source format	Result format	Instruction	Description
2-element <i>Single-precision</i> vector	2-element <i>Half-precision</i> vector	<i>f32v2tof16</i>	Vectorized version of <i>f32tof16</i>
4-element <i>Single-precision</i> vector	4-element <i>Half-precision</i> vector	<i>f32v4tof16</i>	Vectorized version of <i>f32tof16</i>
<i>Half-precision</i>	<i>Single-precision</i>	<i>f16tof32</i>	Convert a <i>Half-precision</i> value to <i>Single-precision</i> : <ul style="list-style-type: none"> • Signaling NaNs are converted to the <i>Tile</i> fixed-value quiet NaN (<i>TFPU_F32_QNaN</i>) and set <i>\$FP_STS.FP_INV</i> to 0b1 • Quiet NaNs are converted to the <i>Tile</i> fixed-value quiet NaN (<i>TFPU_F32_QNaN</i>) • $\pm\infty$ map to $\pm\infty$ • All other values can be represented exactly in <i>Single-precision</i>, so no rounding is performed
2-element <i>Half-precision</i> vector	2-element <i>Single-precision</i> vector	<i>f16v2tof32</i>	Vectorized version of <i>f16tof32</i> .
Unsigned 32-bit integer	<i>Single-precision</i>	<i>f32sufromui</i>	Convert an unsigned 32-bit integer to a single-precision floating-point value in the range $[-\frac{1}{2}, \frac{1}{2}]$. The conversion is symmetrical about 0 and guaranteed to never return exactly 0.0. This instruction is not affected by the <i>Tile</i> rounding mode.
2-element vector of 32-bit unsigned integers	2-element <i>Single-precision</i> vector	<i>f32v2sufromui</i>	Vectorized version of <i>f32sufromui</i>
2-element vector of 16-bit unsigned integers	2-element <i>Half-precision</i> vector	<i>f16v2sufromui</i>	Convert a pair of unsigned 16-bit integer values to a 2-element <i>Half-precision</i> vector where each floating-point value is in the range $[-\frac{1}{2}, \frac{1}{2}]$. The conversion is symmetrical about 0 and guaranteed to never return exactly 0.0. This instruction is not affected by the <i>Tile</i> rounding mode.
4-element vector of 16-bit unsigned integers	4-element <i>Half-precision</i> vector	<i>f16v4sufromui</i>	A 4-element variant of <i>f16v2sufromui</i> .

2.9.7 Floating point exceptions

Tile supports detection of the following classes of exception resulting from floating-point operations:

- *Invalid operation*
- *Divide-by-zero*
- *Overflow*

Tile does not support the following classes of [IEEE754] floating-point exceptions:

- Underflow
- Inexact

By default, floating-point exceptions are treated as *benign* and are *silently* flagged via the context specific status register `$FP_STS`. Floating-point exceptions may optionally be treated as *malign* and hence give rise to a FAULT Tile *exception event* (see `$FP_CTL`).

The floating-point exception flags within `$FP_STS` are *sticky* and can only be cleared via explicit writes to `$FP_CLR`.

2.9.7.1 Exception conditions

2.9.7.1.1 Invalid operation

The invalid operation exception flag (`$FP_STS.INV`) is raised as specified by [IEEE754]. In addition, when `$FP_CTL.NANOO` is 0b1, the invalid operation exception flag will be raised for any half-precision arithmetic instruction that experiences an overflow.

The instructions that can raise the invalid operation exception flag are *here*.

2.9.7.1.2 Divide-by-zero

The divide-by-zero exception flag (`$FP_STS.DIV0`) is raised as specified by [IEEE754].

The instructions that can raise the divide-by-zero exception flag are *here*.

2.9.7.1.3 Overflow

The overflow exception flag (`$FP_STS.OFLO`) is raised as specified by [IEEE754].

The instructions that can raise the overflow exception flag are *here*.

2.9.7.1.4 Underflow

Tile does not support the [IEEE754] underflow exception.

2.9.7.1.5 Inexact result

Tile does not support the [IEEE754] inexact exception.

2.9.8 Comparisons

Tile provides instructions for the following [IEEE754] unordered-signaling predicates (all raise the invalid operation flag for quiet NaN input operands). In the case of scalar values, the result is a single Boolean value. In the case of vectors, comparisons are performed element-wise and the result is a vector of Booleans.

The bit representation of false (true) is given by `TFPU_FP32_FALSE` (`TFPU_FP32_TRUE`) for comparisons of single-precision values and by `TFPU_FP16_FALSE` (`TFPU_FP16_TRUE`) for comparisons of half-precision values. These representations also allow the result of a floating-point comparison to be used as a mask (vector), avoiding the need for control code for certain conditional operations.

Table 2.42: Floating-point comparisons

Predicate name	Mnemonic	f32	f32v2	f16v2	f16v4
compareSignalingEqual	cmpeq	✓	✓	✓	✓
compareSignalingGreater	cmpgt	✓	✓	✓	✓
compareSignalingGreaterEqual	cmpge	✓	✓	✓	✓
compareSignalingLess	cmplt	✓	✓	✓	✓
compareSignalingLessEqual	cmple	✓	✓	✓	✓
compareSignalingNotEqual	cmpne	✓	✓	✓	✓

2.9.9 Accumulation

Accumulating operations are performed using internal accumulator state (see *Pipeline internal state*). The precise accuracy and format of the internal accumulator state is *implementation dependent* and so results of accumulating instructions may differ between different implementations of *Tile*.

Regardless of the internal accumulator format, accumulator state can be written to in a number of ways:

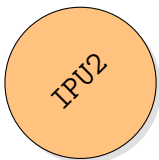
- Explicitly zeroed via `$FP_CLR.ZAACC`
- Initialised from the *ARF* using half-precision values via `f16v2gina`
- Initialised from the *ARF* using single-precision values via `f32v2gina`
- As unformatted, arbitrary data via `f16v4istacc`

The accumulator state may be read:

- as half-precision values via `f16v2gina`
- as single-precision values via `f32v2gina`
- as the half or single-precision results of certain accumulating instructions (see `f32sisoamp` for example)
- as (permuted) arbitrary data via `f16v4istacc` and `f16v4stacc`

In the cases where the accumulator state is read out as half-precision values, *stochastic rounding* may be applied.

Implementation detail



IPU2's floating-point accumulator format is strictly single-precision, with denorms flushed to zero.

2.9.9.1 Single-precision multiply-accumulate

Scalar and element-wise vector single-precision multiply-accumulate instructions (which implicitly use the accumulator state as the addend) round the intermediate multiplication result to the internal accumulator format prior to the addition (unlike *[IEEE754]*'s **fusedMultiplyAdd** operation which is computed as if with unbounded range and precision, with only a single round of the final result).

- `f32mac`
- `f32v2aop`
- `f32v2mac`
- `f32v4sqacc`

2.9.9.2 Half-precision multiply-accumulate

Element-wise vector half-precision multiply-accumulate instructions (which implicitly use the accumulator state as the addend) round the intermediate multiplication result to the internal accumulator format prior to the addition.

- `f16v8sqacc`

2.9.10 Dot-products

2.9.10.1 Half-precision vector dot-products

Certain instructions perform a vector dot-product operation between 2 and 4-element half-precision input vectors.

The result of the dot-product operation is a rounded single-precision value, which may differ from that calculated as if with unbounded range and precision for the intermediate results.

For dot-product operations on 4-element vectors, the second input vector (the weight vector) is provided by the *Common Compute Configuration State*, which is initialised by the Supervisor context and shared between all worker contexts.

For dot-product operations on 2-element vectors, the second input vector is provided either by the *ARF* (as in the case of *f16v4cmac*) or by the *\$TAS* CSR in the case of *f16v4mix*

- *f16v2cmac*
- *f16v4hihoslic*
- *f16v4mix*
- *f16v4sisoamp*
- *f16v4cmac*
- *f16v4hihov4amp*
- *f16v4sihoamp*
- *f16v4sisoslic*
- *f16v4hihoamp*
- *f16v4hihov4slic*
- *f16v4sihoslic*

2.9.11 AMP and SLIC instructions

The **Accumulating Matrix Product** and **Slim Convolution** instructions facilitate high performance multiply-accumulate sequences, for scenarios where weight sharing across Worker contexts is appropriate.

AMP and SLIC instructions are supported for both single-precision and half-precision number formats and operate in the same basic manner:

- The *Common Compute Configuration State* must first be initialised by the Supervisor context
- 2 *streams* of input data:
 - Partial-sum values specifying a starting value for a subsequent multiply-accumulate sequence (for convolutions these are the partially-computed output pixel/activations)
 - The input activation (pixel) data
- A single stream of output data: the resulting, accumulated partial-sum values
- Each partial-sum input value is subjected to a fixed-length sequence of multiply-accumulate operations, before the final partial-sum result is presented as an output.
- Internal accumulation is to the *implementation dependent* accumulator format, regardless of the input and output data formats.
- Many dot-product ; accumulate operations occur in parallel, performed by compute engines:
 - Each compute engine supports the following operations:
 - * 2 x 4-element f16 *dot-product* plus accumulate
 - * 2 x scalar f32 multiplication plus accumulatewhere the 1st multiplicand is provided by the input data stream and the 2nd by the *Common Compute Configuration State*.
- The compute engines are logically organised into sets
- The latency between the load of an input partial-sum and the updated partial-sum being presented as an output is such that the result can be written-back to the original memory location when operating at peak performance (assuming the output resides in a *memory region* with an interleave factor of at least 2).
- *Stochastic rounding* is available for the instruction variants which produce half-precision results.

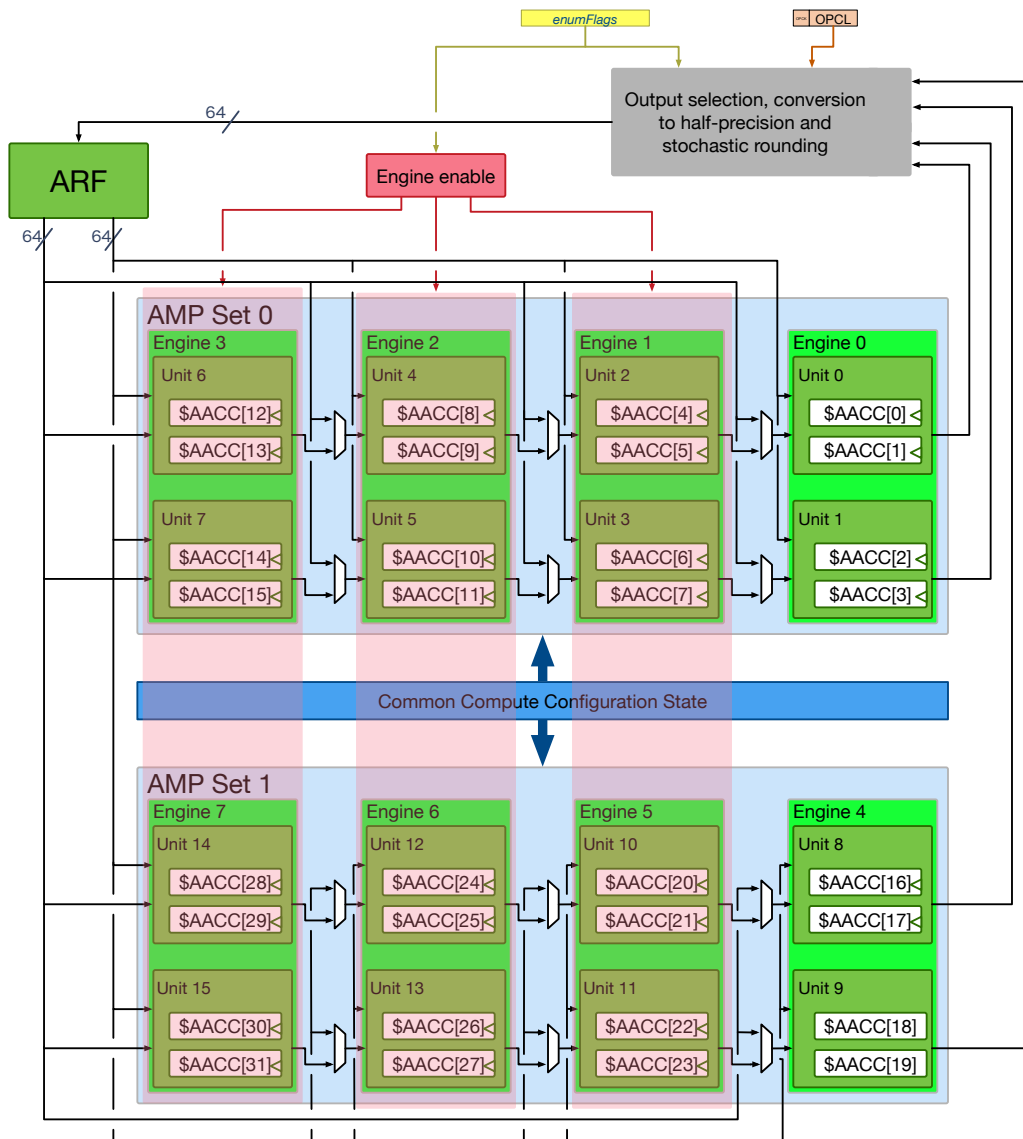


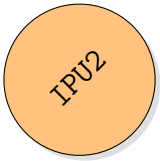
Fig. 2.31: AMP unit connectivity

See the individual instruction definitions for further details. The following table lists the AMP/SLIC instruction variants:

Table 2.43: AMP/SLIC instruction variants

Instruction	Input data format	Input partial format	Output partial format	Active AMP sets
<i>f16v4sisoamp</i>	Half-precision (v4)	Single-precision (v2)	Single-precision (v2)	1
<i>f16v4hihoamp</i>	Half-precision (v4)	Half-precision (v2)	Half-precision (v2)	1
<i>f16v4hihov4amp</i>	Half-precision (v4)	Half-precision (v4)	Half-precision (v4)	2
<i>f16v4sihoamp</i>	Half-precision (v4)	Single-precision (v2)	Half-precision (v2)	1
<i>f16v4sisoslic</i>	Half-precision (v4)	Single-precision (v2)	Single-precision (v2)	1
<i>f16v4hihoslic</i>	Half-precision (v4)	Half-precision (v2)	Half-precision (v2)	1
<i>f16v4hihov4slic</i>	Half-precision (v4)	Half-precision (v4)	Half-precision (v4)	2
<i>f16v4sihoslic</i>	Half-precision (v4)	Single-precision (v2)	Half-precision (v2)	1
<i>f32sisoamp</i>	Single-precision (scalar)	Single-precision (v2) ⁵	Single-precision (v2) ⁵	1
<i>f32sisov2amp</i>	Single-precision (scalar)	Single-precision (v2)	Single-precision (v2)	2
<i>f32sisoslic</i>	Single-precision (scalar)	Single-precision (v2) ⁵	Single-precision (v2) ⁵	1
<i>f32sisov2slic</i>	Single-precision (scalar)	Single-precision (v2)	Single-precision (v2)	2

Implementation detail



IPU2 includes 8 of the AMP/SLIC computation engines, arranged as 2 sets of 4 engines, providing a peak performance of:

- 64 half-precision fmac operations per cycle, or
- 16 single-precision fmac operations per cycle

2.9.12 Transcendental functions

The *Tile* architecture defines a number of instructions that perform long-latency Transcendental functions on scalar *Single-precision* values and *Half-precision* input vectors. The precise list of transcendental instructions supported is an *implementation dependent* subset of those listed below. Their accuracy, input domains and latency are also *implementation dependent*. Attempts to execute an unimplemented transcendental instruction will result in a `TEXCPT_INVALID_INSTR` exception.

⁵ New output produced every other instruction

Table 2.44: Transcendental functions

Function	Number formats	Typical Domain	Instruction(s)	Description
$\log_2(x)$	Single-precision, Half-precision	$x \in [0, +\infty]$	<i>f32log2</i> , <i>f16v2log2</i>	Base 2 logarithm.
$\ln(x)$	Single-precision, Half-precision	$x \in [0, +\infty]$	<i>f32ln</i> , <i>f16v2ln</i>	Natural logarithm.
2^x	Single-precision, Half-precision	$x \in [-\infty, +\infty]$	<i>f32exp2</i> , <i>f16v2exp2</i>	Base 2 exponential.
e^x	Single-precision, Half-precision	$x \in [-\infty, +\infty]$	<i>f32exp</i> , <i>f16v2exp</i>	Natural exponential.
<i>logistic</i> (x)	Single-precision, Half-precision	$x \in [-\infty, +\infty]$	<i>f32sigm</i> , <i>f16v2sigm</i>	$\frac{1}{1+e^{-x}}$
<i>tanh</i> (θ)	Single-precision, Half-precision	$\theta \in [-\infty, +\infty]$	<i>f32tanh</i> , <i>f16v2tanh</i>	Hyperbolic tangent

2.9.13 IEEE 754-2008 clarifications

2.9.13.1 NaN generation

- For formats which support NaNs, *Tile regenerates* rather than *propagates* quiet NaN input values. That is to say that those operations for which [IEEE754] specifies propagation of input quiet NaNs will produce the *Tile* fixed-value quiet NaN.
- Similarly, the quietening of signaling NaNs produces the *Tile* fixed-value quiet NaN.
- If both input operands for a floating-point min or max instruction are quiet NaNs, the result value will be the *Tile* fixed-value quiet NaN.
- Unlike min and max instructions, clamp instructions (such as *f16v4clamp*) will produce a quiet NaN if any of its inputs is a quiet NaN.
- For conversion operations where both the source and target formats support NaNs, signaling NaNs are quietened (converted to the fixed-value quiet NaN).
- For all other operations and inputs, regardless of whether one or more of the inputs is a quiet NaN, when a quiet NaN is delivered as the result, that NaN will be of a fixed value (*TFPU_F32_QNaN* in the case of single-precision and *TFPU_F32_QNaN* rounded to nearest in the case of half-precision).
- When *\$FP_CTL.NANOO* is 0b1, the fixed-value quiet NaN value will be returned by any instruction experiencing an overflow on a half-precision based calculation.

2.9.14 IEEE 754-2008 caveats and differences

- *f16 arithmetic saturates* (to ± 65504)
- *Single-precision denorms are treated as zero*
- *NaNs are regenerated (producing fixed-value NaNs), rather than propagated*
- *No support for the underflow or inexact exceptions*

2.9.14.1 Transcendentals

The accuracy of instructions that implement transcendental functions (*Transcendental functions*) are *implementation dependent* and may not produce result values equivalent to those calculated to infinite precision, rounded accordingly for all number formats.

2.10 Exception model

Note: The details provided by this section are not relevant to the *Tile Vertex ISA*.

2.10.1 Exception types

Tile recognises 2 types of *Exception*:

1. *Malign*

Such exceptions represent programming or unrecoverable hardware errors. All such exceptions result in the raising of a *FAULT Exception Event*.

2. *Benign*

Such exceptions indicate unusual execution conditions but are not considered programming errors. *Benign* exceptions will always be flagged in a manner visible to software but may not give rise to an *Exception Event*. When *Benign* exceptions do give rise to *Exception Events*, such events will always be of the *BREAK* type.

Note: Floating point calculation exceptions default to being of *Benign* type but can be re-classified as *Malign*. See *Floating point exceptions*.

2.10.2 Exception Events

Exception Events occur whenever it is deemed necessary to halt the execution of a *thread* that caused an *Exception*.

2.11 Debug

Note: The details provided by this section are not relevant to the *Tile Vertex ISA*.

2.12 Exchange interface

Note: The details provided by this section are not relevant to the *Tile Vertex ISA*.

2.13 Pseudorandom number generator

Tile includes a pseudorandom number generator (PRNG) circuit heavily inspired by the [xoroshiro128+](#) 128-bit full-period generator. The generator is referred to as *xoroshiro128aox* to signify the replacement of the 64-bit addition operation of *xoroshiro128+* with a function comprised of bitwise AND, OR and XOR operations (see [xoroshiro128aox](#)). The hardware allows for the generation of random values sampled from both the discrete uniform distribution and a quantized 12th degree [Irwin-Hall](#) distribution (an approximation to the Normal distribution).

The generated random values are used (by Worker contexts only) in three ways:

1. to explicitly generate random operand data by executing PRNG instructions
2. to *randomly mask* individual elements of floating-point vectors
3. to implement the *stochastic rounding* mode for a subset of floating-point instructions.

2.13.1 State

The *Tile* PRNG hardware uses the following, Worker context specific architectural state, which can be read and written using *uget* and *uput*:

- `$PRNG_0_0`
- `$PRNG_0_1`
- `$PRNG_1_0`
- `$PRNG_1_1`

Note: The four PRNG state registers can also be initialised from a single 32-bit value using the special alias CSR `$PRNG_SEED`.

2.13.2 Random Number Generation

The *Tile* PRNG hardware performs 2 steps of the *xoroshiro128aox* algorithm for every PRNG instruction executed, or when stochastic rounding is used. See [xoroshiro128aox](#).

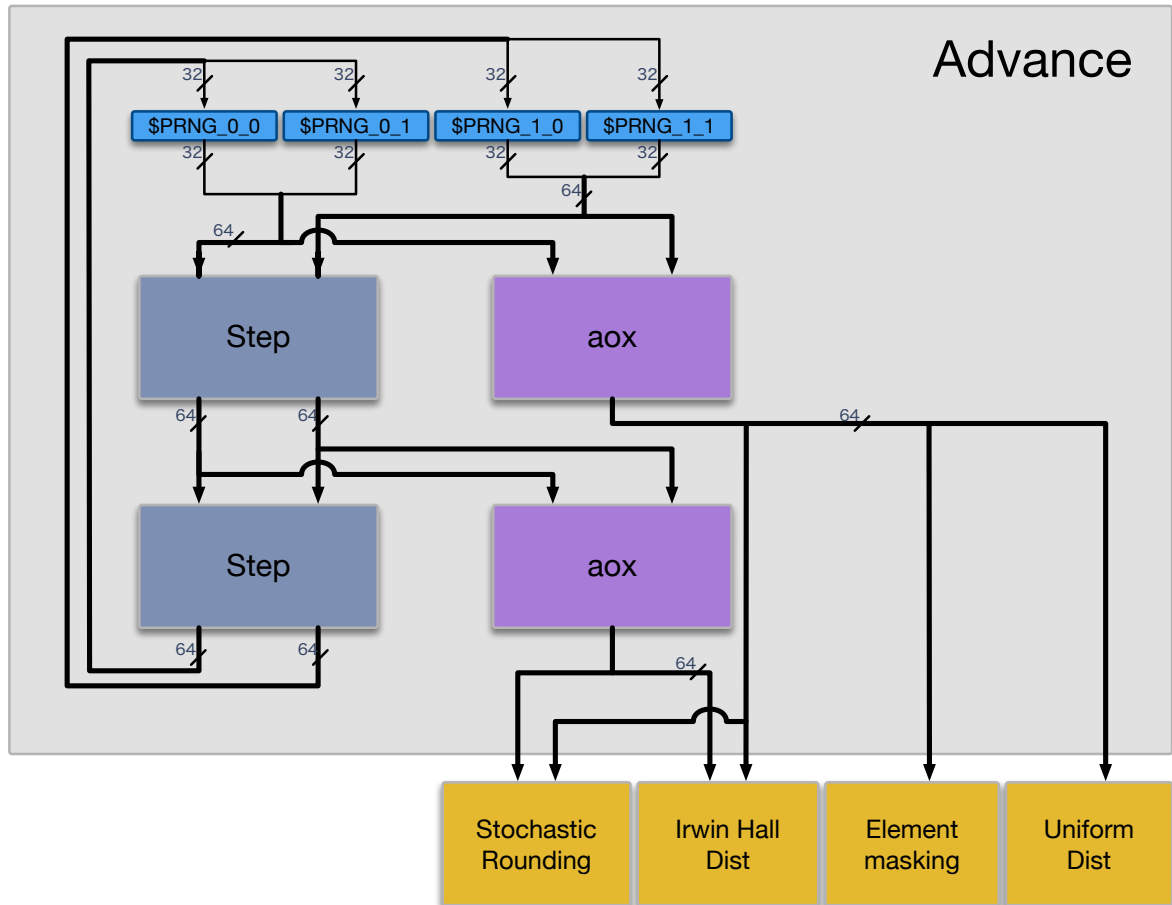


Fig. 2.32: PRNG overview

2.13.2.1 Quality

The following table compares the failures of *xoroshiro128aox* on TestU01's BigCrush test-suite, with that of *xoroshiro128+*. Four arrangements are separately tested:

- **std**: In the case of integer inputs, the upper and lower 32-bits of the generated 64-bit bit-pattern are each presented in their original bit-order. In the case of double-precision floating-point values, the bottom 52-bits of the generated 64-bit pattern are used as the mantissa.
- **rev**: In the case of integer inputs, the upper and lower 32-bits of the generated 64-bit bit-pattern are each presented in reverse bit-order. In the case of double-precision floating-point inputs, the bottom 52-bits of the reversed 64-bit pattern are used as the mantissa.
- **swap16-std**: In the case of integer based tests, the upper and lower 16-bits of each of the upper and lower 32-bits of the generated 64-bit bit-pattern are swapped. In the case of double-precision floating-point inputs, the bottom 52-bits of the swapped 64-bit pattern are used as the mantissa.
- **swap16-rev**: In the case of integer based tests, the upper and lower 16-bits of each of the upper and lower 32-bits of the reversed 64-bit bit-pattern are swapped. In the case of double-precision floating-point inputs, the bottom 52-bits of the swapped, bit-reversed 64-bit pattern are used as the mantissa.

Table 2.45: BigCrush failures

PRNG	Period	std	rev	swap16-std	swap16-rev	Overall
xoroshiro128aox	$2^{128} - 1$	30	30	37	31	128
xoroshiro128+	$2^{128} - 1$	31	27	134	129	321

Note:

- xoroshiro128+ std/rev BigCrush figures provided by <http://xoroshiro.di.unimi.it/>
 - swap16-std and swap16-rev results from internal testing
 - The randomness of the 16-bit sub-fields is particularly relevant for the *element-masking* instructions
 - The MatrixRank test with parameter L=5000 produces a systematic failure (i.e. fails for all tested seeds) using *xoroshiro128+*, with the swap16-std and swap16-rev bit orders. This highlights a weakness in bit 0 of numbers generated using *xoroshiro128+*.
-

Listing 2.9: xoroshiro128aox

```
typedef struct {
    uint64_t ires1; // Intermediate result, following single application of adapted xoroshiro128+
} tprngt_Internal;
```

Listing 2.10: xoroshiro128aox

```
// @brief Vanilla rotate-left
static uint64_t rotl(uint64_t x, int k) {
    return (x << k) | (x >> (64 - k));
}

// @brief perform the shifting/(x)or'ing as per xoroshiro128+
// @param s0/1 128-bits of input state
static void xoroshiro128_Step(uint64_t &s0, uint64_t &s1) {
    uint64_t t1 = s1 ^ s0;
    uint64_t t0 = rotl(s0, 55);

    s0 = t0 ^ t1 ^ (t1 << 14);
    s1 = rotl(t1, 36);
}

// @brief Alternative non-linear operation. Used in place of xorshiro128+'s 64-bit addition,
//         in particular to improve an observed linear artifact of bit 0.
//
//
static uint64_t aox(uint64_t s0, uint64_t s1) {
    uint64_t sum = s1 ^ s0;
    uint64_t carry = s1 & s0;
    return sum ^ (rotl(carry, 1) | rotl(carry, 2));
}

// @brief Advance the PRNG state, by performing 2 steps of xoroshiro128aox
// @param internal intermediate results are stored here
void TPRNG_Advance(Register &$PRNG_0_0,
                  Register &$PRNG_0_1,
                  Register &$PRNG_1_0,
                  Register &$PRNG_1_1,
                  std::array<uint64_t, 2> &randomBits) {
    uint64_t s1 = $PRNG_0_1.get() << 32 | $PRNG_0_0.get();
    uint64_t su = $PRNG_1_1.get() << 32 | $PRNG_1_0.get();

    randomBits[0] = aox(s1, su);

    // Perform single step of xoroshiro128aox and save the result
    xoroshiro128_Step(s1, su), randomBits[1] = aox(s1, su);

    // Perform final step of xoroshiro128aox and write the result to $PRNG_0/1
    xoroshiro128_Step(s1, su);

    $PRNG_0_0.set(s1, WriteSemantics::ForceWrite);
    $PRNG_0_1.set(s1 >> 32, WriteSemantics::ForceWrite);
    $PRNG_1_0.set(su, WriteSemantics::ForceWrite);
    $PRNG_1_1.set(su >> 32, WriteSemantics::ForceWrite);
}
```

2.13.3 Discrete Uniform Distribution

The *Tile* PRNG hardware supports the generation of 32-bit and 64-bit integer variables from the discrete uniform distribution.

In addition, uniformly distributed floating-point values within the range $[-\frac{1}{2}, \frac{1}{2}]$ can be obtained by combining the uniform-random integer instructions with the symmetric unbiased floating-point conversion instructions:

2.13.3.1 Integer Instructions

- *urand32*
- *urand64*.

2.13.3.2 Floating-point conversion instructions

- *f16v2sufromui*
- *f16v4sufromui*
- *f32sufromui*
- *f32v2sufromui*

2.13.4 Irwin-Hall Distribution

A quantized 12th degree Irwin-Hall distribution is used as an accurate approximation to the Standard normal distribution.

The *Tile* PRNG supports the generation of both *single-precision* and *half-precision* random variables from this distribution. In each case, the generated random number is the result of a 12-element addition of 5-bit fields extracted from the repeated application of xoroshiro128aox.

2.13.4.1 Properties

- The range of the generated floating-point values is $[-5\frac{13}{16}, 5\frac{13}{16}]$
- The quantized distribution provides 373 unique values within that range.
- The mean of the generated distribution is 0.
- The standard deviation of the generated distribution is $\sqrt{\frac{1023}{1024}}$ (~ 0.9995)⁶
- The kurtosis of the generated distribution is $\frac{5933}{2046}$ (~ 2.8998)⁷

⁶ The standard deviation of the Standard normal distribution is 1.0

⁷ The kurtosis of the Standard normal distribution is 3.0

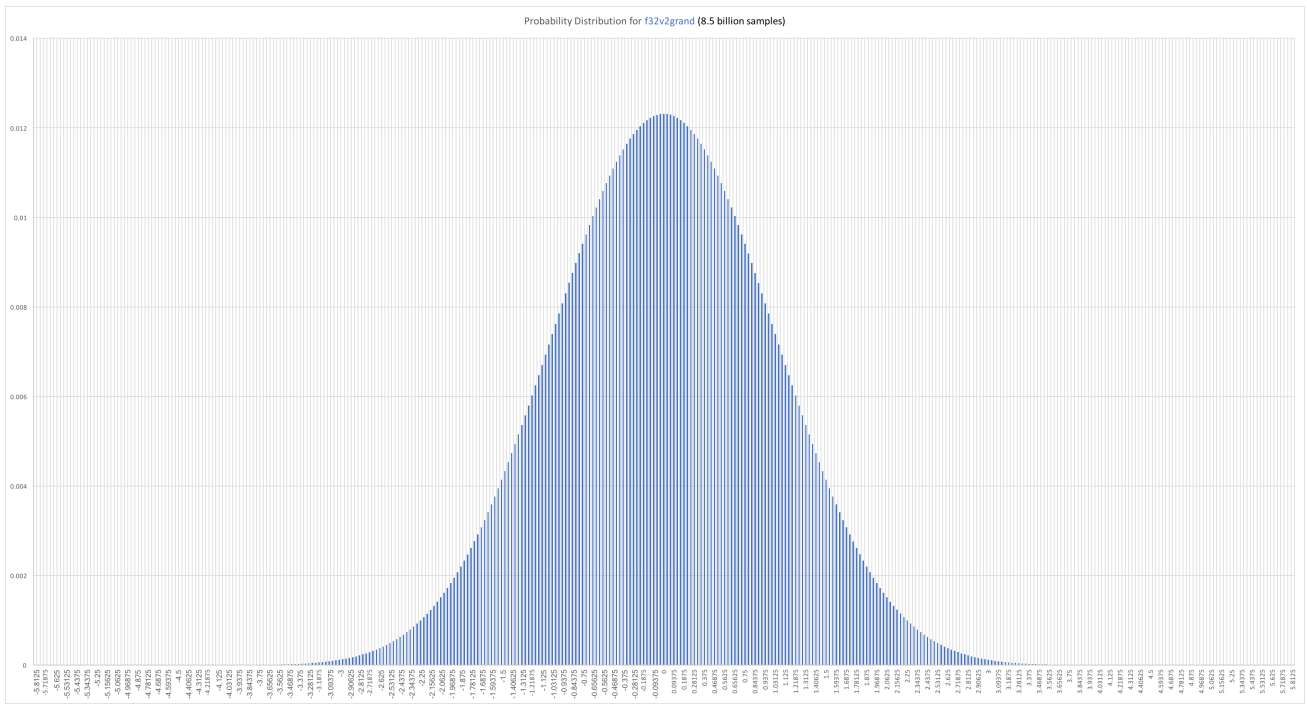


Fig. 2.33: f32v2grand probability distribution

2.13.4.2 Instructions

- *f32v2grand*
- *f16v2grand*

2.13.5 Element Masking

The *Tile* PRNG provides the ability to randomly mask individual members of both 4-element *half-precision* vectors and 2-element *single-precision* vectors. In both cases, each element is masked with a probability between 0 and 1, with the unmasking probability provided by a separate register value.

2.13.5.1 Instructions

- *f32v2rmask*
- *f16v4rmask*

2.13.6 Stochastic Rounding

Stochastic rounding mode applies to a subset of floating-point instructions, when those instructions are performing a down-conversion of a result, either explicitly, or implicitly (due to the extended accuracy of intermediate results).

When stochastic rounding mode is enabled (see *\$FP_CTL.ESR*), stochastic rounding is used in preference to the rounding mode specified by *\$FP_CTL.RND*, for those instructions listed below. All other instructions are unaffected by *\$FP_CTL.ESR*.

When stochastic rounding mode is enabled, those instructions that support it will first produce a full, single-precision intermediate result, using round to nearest, ties to even rounding. The PRNG hardware is then used to generate a uniformly distributed random bit pattern, which is masked and added to the mantissa of the rounded, single-precision intermediate result. The random bit pattern is masked such that most-significant-bit of the masked value lines up with mantissa bit 1 place below the rounding-point for the (smaller) target number format. The resulting floating-point mantissa is then truncated beneath the rounding-point as the final part of conversion. Since the random bit pattern is sampled from a uniform distribution, the probability of generating a carry into the lsb of the result mantissa is directly proportional to the absolute value of the mantissa bits below the lsb (with the mantissa bits below the lsb interpreted as an unsigned binary integer).

Stochastic rounding applies to any single-precision intermediate result value with an absolute value of at least 2^{-25} (i.e. half the size of the smallest half-precision denorm). Any intermediate result strictly less than this value will always be rounded down to 0.0.

The PRNG state is advanced by 2 steps of xoroshiro128aox for every instruction that performs stochastic rounding on its result (regardless of the format of the result).

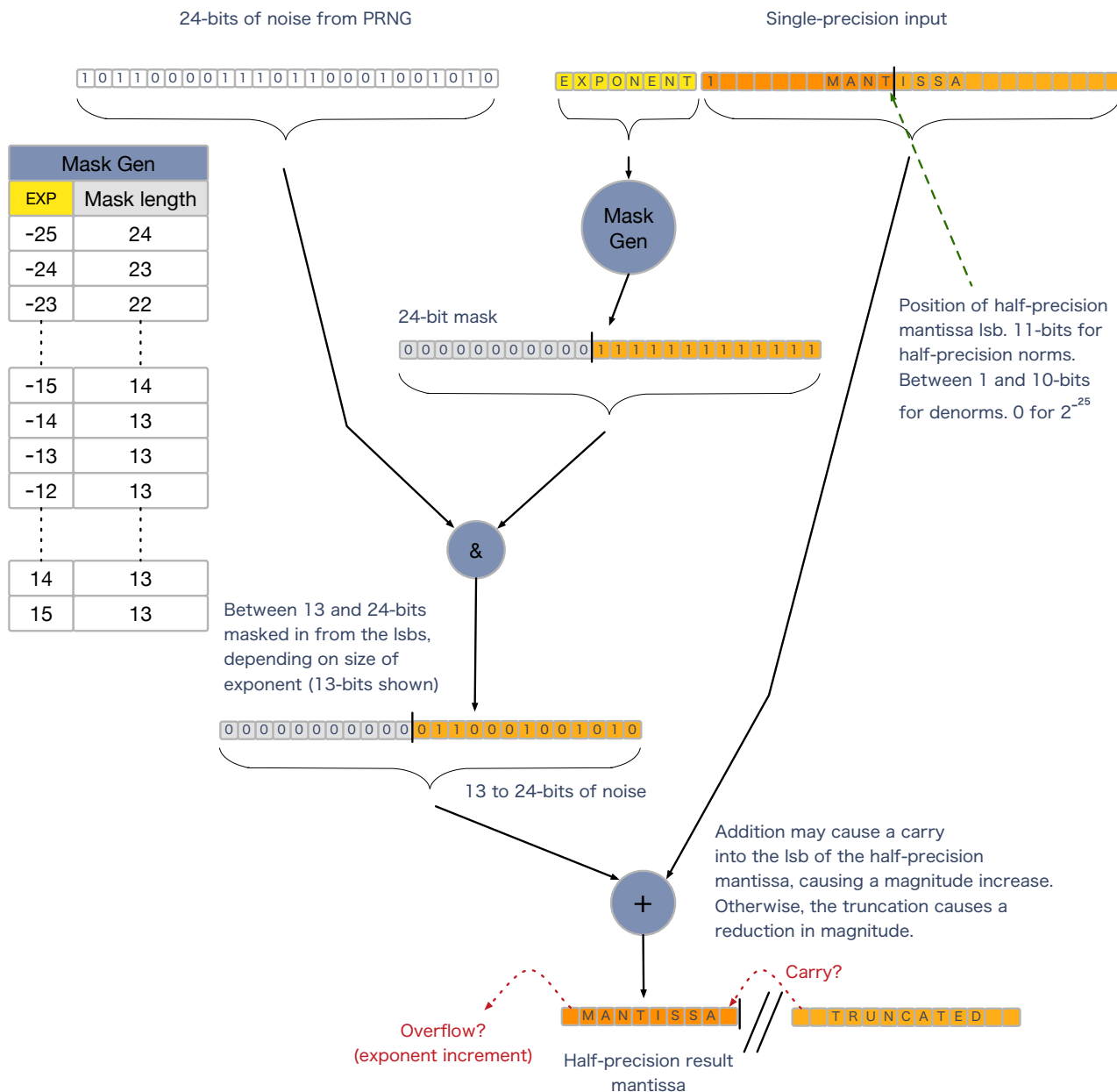


Fig. 2.34: Stochastic rounding of single to half-precision

Listing 2.11: Stochastic rounding f32 to f16

```
// @brief Perform stochastic rounding on a single-precision value,
//         producing a half-precision result.
// @param single the full single-precision value to be rounded
// @param bitpattern the random bit-pattern of extended mantissa bits
// @returns *single* (unmodified) if it is NaN or +/- infinity
//         +/- Zero if the rounded f32 result is less than 2^-24 (min f16 denorm)
//         +/- exp2f(16) if the rounded f32 result is greater than 65504 (max f16 norm)
//         Otherwise, a single-precision representation of *single* stochastically rounded
//         to half-precision (i.e. both the exponent range and mantissa accuracy)
```

```

//      are as per half-precision)
float TFPU_StochasticRound(float single, uint32_t bitpattern) {
    uint32_t sign, uexponent, mantissa;
    int sexponent, masklen;
    float result;

    if (std::isnan(single) || std::isinf(single)) {
        return single;
    }

    // Extract the exponent value from the single-precision float
    // and perform a range check
    sexponent = TFPU_F32_Exp(single);
    TFPU_F32_Decompose(single, &sign, &uexponent, &mantissa);

    if (sexponent >= TFPU_F16_MIN_NORM_EXP) {
        // f16 norm - use 13-bits of random mantissa
        // below the f16 lsb.
        masklen = TFPU_F32_M_SIZE - TFPU_F16_M_SIZE;
    } else if (sexponent >= (TFPU_F16_MIN_DENORM_EXP - 1)) {
        // f16 denorms (or at least half the size of the smallest f16 denorm)
        // - use between 14 and 24 bits of random mantissa bits below the lsb
        masklen = TFPU_F32_M_SIZE + (TFPU_F16_MIN_DENORM_EXP - sexponent);
    } else {
        masklen = TFPU_F32_M_SIZE + 1;
    }
    // Or-in the implicit 1 of the mantissa
    mantissa |= (1 << TFPU_F32_M_SIZE);

    // Mask the random bit-pattern and add it to the mantissa
    // of the input value.
    bitpattern &= ((1 << masklen) - 1);
    mantissa += bitpattern;

    // Did the addition overflow the mantissa?
    if ((mantissa >> (TFPU_F32_M_SIZE + 1)) & 1) {
        uexponent += 1;
        sexponent += 1;
        mantissa >>= 1; // Right shift not strictly necessary
                        // since all explicit significant bits are 0.
                        // Top mantissa bit is always implicitly 1.
    }

    // Convert the result to Half-precision accuracy by truncating the mantissa.
    // And remove the implicit 1
    mantissa >>= masklen;
    mantissa <<= (masklen + (32 - TFPU_F32_M_SIZE));
    mantissa >>= (32 - TFPU_F32_M_SIZE);

    // Range check result
    if (sexponent < TFPU_F16_MIN_DENORM_EXP) {
        result = 0.0;
    } else if (sexponent > TFPU_F16_MAX_NORM_EXP) {
        // Return a finite value above the f16 range.
        // Downstream conversion to f16 will then convert appropriately
        result = exp2f(16);
    } else {
        result = TFPU_F32FromBits((uexponent << TFPU_F32_E_OFFSET) | (mantissa << TFPU_F32_M_OFFSET));
    }

    return copysign(result, single);
}

```

Listing 2.12: Stochastic rounding f32 to f16

```

// @brief Apply stochastic rounding to a vector of single-precision values
void TFPU_ApplyStochasticRound(array<uint64_t, 2> &randm, vector<float> &values) {
    array<uint32_t, 4> randomBits;

    // Extract the (24-bit) random bit patterns from $PRNG_0/1 and
    // the intermediate values.

```

```

randomBits[0] = ((randm[0] >> 0) & 0xffff) | (((randm[1] >> 0) & 0xff) << 16);
randomBits[1] = ((randm[0] >> 16) & 0xffff) | (((randm[1] >> 8) & 0xff) << 16);
randomBits[2] = ((randm[0] >> 32) & 0xffff) | (((randm[1] >> 16) & 0xff) << 16);
randomBits[3] = ((randm[0] >> 48) & 0xffff) | (((randm[1] >> 24) & 0xff) << 16);

// Apply stochastic rounding to each of the result values
for (unsigned int i = 0; i < values.size(); i++) {
    values[i] = TFPU_StochasticRound(values[i], randomBits[i]);
}
}

```

The following instructions implement stochastic rounding:

- *f16v2absadd*
- *f16v2add*
- *f16v2gina*
- *f16v2mul*
- *f16v2sub*
- *f16v4absadd*
- *f16v4add*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4mix*
- *f16v4mul*
- *f16v4sihoamp*
- *f16v4sihoslic*
- *f16v4sub*
- *f32tof16*
- *f32v2tof16*
- *f32v4tof16*

INSTRUCTIONS

3.1 Instruction signatures

Tile instructions have one of the following parameter signatures:

Table 3.1: Instruction signatures

Signature	Example	Description
<i>instr src0</i>	<i>br</i>	The instruction has a single register source argument (\$m4 for example, see <i>Register specifiers</i>).
<i>instr imm0</i>	<i>bri</i>	The instruction has a single source argument, encoded as an immediate (simm6 for example).
<i>instr dst0</i>	<i>f16v2grand</i>	The instruction has a single explicit register destination argument only (\$a2:3 for example).
<i>instr dst0, src0</i>	<i>abs</i>	The instruction has a single register destination argument (which is modified at instruction retirement), \$m2 for example, and a single source register argument.
<i>instr src0, imm0</i>	<i>brneg</i>	The instruction has one source register argument and one immediate argument.
<i>instr dst0, imm0</i>	<i>call</i>	The instruction has one destination register argument (which is modified at instruction retirement) and one immediate argument.
<i>instr src0, src1</i>	<i>f16v2cmac</i>	The instruction has two source register arguments
<i>instr imm0, src0</i>	<i>put</i>	The instruction has one immediate argument and one source register argument.
<i>instr imm0, imm1</i>	<i>rpt</i>	The instruction has two arguments encoded as immediates.
<i>instr srcDst0, imm0</i>	<i>brnzdec</i>	The instruction has one source and destination register argument and one immediate argument. The register argument is modified at instruction retirement.
<i>instr dst0, src0, src1</i>	<i>add</i>	The instruction has one destination register argument and two source register arguments.
<i>instr dst0, src0, imm0</i>	<i>add</i>	The instruction has one destination register argument, one source register argument and one source argument encoded as an immediate.
<i>instr src0, src1, imm0</i>	<i>f32v2aop</i>	The instruction has two source register arguments and one source argument encoded as an immediate.

Continued on next page

Table 3.1 – continued from previous page

Signature	Example	Description
<i>instr src0, src1, src2</i>	<i>stm32</i>	The instruction has three source register arguments.
<i>instr dst0, imm0, src0</i>	<i>sub</i>	The instruction has one destination register argument, one immediate argument and one source register argument.
<i>instr srcDst0, src0, src1</i>	<i>movz</i>	The instruction has one source and destination register argument and two source register arguments. The source and destination register is modified at instruction retirement.
<i>instr src0, srcDst0+=, src1</i>	<i>stm32step</i>	The instruction has three source register arguments, the 2nd of which is post-incremented by an amount specified by the final source register
<i>instr dst0, src0, src1, imm0</i>	<i>f16v4hihoamp</i>	The instruction has one destination register argument, two source register arguments and one source argument encoded as an immediate.
<i>instr dst0, src0, src1, src2</i>	<i>ld128</i>	The instruction has one destination register argument and three source register arguments.
<i>instr src0, src1, src2, src3</i>	<i>st32</i>	The instruction has four source register arguments.
<i>instr src0, src1, src2, imm0</i>	<i>st32</i>	The instruction has three source register arguments and one immediate.
<i>instr dst0, src0, srcDst0+=, imm0</i>	<i>ld128step</i>	The instruction has one destination register argument, one source register argument, one source and destination register argument and one immediate. The source and destination register argument is post-incremented by the immediate value (possibly scaled).
<i>instr dst0, src0, srcDst0+=, src1</i>	<i>ld128step</i>	The instruction has one destination register argument, two source register arguments and one source and destination register argument. The source and destination register is post-incremented by the final source register (possibly scaled).
<i>instr dst0, srcDst0+=, src0, src1</i>	<i>ld64a32</i>	The instruction has one destination register argument, two source register arguments and one source and destination register argument. The source and destination register is post-incremented by a single atom (where the size of an atom is dependent on the instruction).
<i>instr src0, src1, srcDst0+=, imm0</i>	<i>st32step</i>	The instruction has two source register arguments, one source and destination register argument and one immediate. The source and destination register is post-incremented by the immediate value (possibly scaled).
<i>instr src0, src1, srcDst0+=, src2</i>	<i>st32step</i>	The instruction has three source register arguments and one source and destination register argument. The source and destination register is post-incremented by the final source register value (possibly scaled).
<i>instr src0, srcDst0+=, src1, imm0</i>	<i>st64pace</i>	The instruction has 3 source registers arguments and one immediate. The second source argument is post-modified in a manner specified by the third and forth arguments

Continued on next page

Table 3.1 – continued from previous page

Signature	Example	Description
<i>instr dst0, src0, srcDst0++, srcDst1</i>	<i>ldb16b16</i>	The instruction has one destination register argument, one source register argument and two source and destination register arguments. The first source and destination register is post-incremented by 1 atom. The second source and destination register is modified by a load into that register
<i>instr dst0, srcDst0++, src0, srcDst1@</i>	<i>ldd16a32</i>	The instruction has one destination register argument, one source register argument and two source and destination register arguments. The first source and destination register is post-incremented by 1 atom. The second source and destination register is modified by a load into that register.
<i>instr dst0, dst1, srcDst0+=, src0, imm0</i>	<i>ld2x64pace</i>	The instruction has two destination register arguments, one source register argument, one immediate argument and one source and destination register argument. The source and destination register is post-modified in a manner described by a combination of the source register and immediate value.
<i>instr dst0, src0, srcDst0+=, src1, imm0</i>	<i>ld2xst64pace</i>	The instruction has one destination register argument, two source register arguments, one immediate argument and one source and destination register argument. The source and destination register is post-modified in a manner described by a combination of the source register and immediate value.

3.2 Register specifiers

Tile instruction syntax register specifiers are composed of:

- A register file specifier:
 - \$m refers to the *MRF*
 - \$a refers to the *ARF*
- followed by one of:
 - a single register index in the range [0,15]. For example, \$m4 specifies register 4 of the *MRF*.
 - a 2 (pair) or 4 (quad) register index range. For example, \$a2:3 specifies the register pair composed of \$a2 and \$a3. When specifying a register index range, the base index (2 here) must be *naturally aligned* to the size of the range. If the base index isn't naturally aligned, the least significant bits of the index are ignored and assumed to be 0. See *ARF* for further details.
 - a single source register index, with a broadcast modifier. Such register specifiers result in a scalar value, from the given register being *broadcast* (duplicated) to form a vector of the appropriate size for the instruction. The resulting vector is then used as the indicated source operand for the operation. For example:
 - * \$a1:B specifies that a 32-bit value from ARF register 1 is to be broadcast (unmodified) to a 2-element vector
 - * \$a3:BL specifies that a 16-bit value from the lsbs of ARF register 3 is to be broadcast (unmodified) to a 2 or 4 element vector (whichever size is appropriate for the instruction)
 - * \$a5:BU specifies that a 16-bit value from the msbs of ARF register 5 is to be broadcast (unmodified) to a 2 or 4 element vector (whichever size is appropriate for the instruction)

In addition, this document uses / as a shorthand to represent an option between 2 specific registers. \$m14/15 for example states that either \$m14 or \$m15 can be used as the indicated operand.

The vast majority of instructions use strictly one of MRF or ARF for all source operands and write results (if any) back to the same register file. The following exceptions apply:

- *Tile Memory* addresses for load and store instructions are always formed from MRF register values, or MRF register values combined with instruction immediates.
- Some load instructions can only write to the ARF (*ld64* for example). Others can write to either the ARF or MRF (*ld32* for example). However, any attempt by the Supervisor context to execute a load instruction with an ARF destination operand will result in a `TEXCPT_INVALID_INSTR` *Exception*.
- Some store instructions can only store data from the ARF (*st64* for example). Others can store data from either the ARF or MRF (*st32* for example). However, any attempt by the Supervisor context to execute a store instruction with an ARF source operand will result in a `TEXCPT_INVALID_INSTR` *Exception*.
- *atom* takes its source operand from ARF and writes to the MRF.

Note: A register may not be used as a destination (or source-destination) operand more than once for any instruction. Attempt to use the same register for multiple destination (or source-destination) operands will result in a `TEXCPT_INVALID_OP` *exception event*.

3.3 Types of immediate

Table 3.2: Immediate types

Syntax	Description	Resulting value
<i>simmn</i>	An n-bit wide signed immediate. This operand is implicitly <i>sign extended</i> to <i>word</i> size.	$32'b\{32-n'b\text{simmn}[n-1], n'b\text{simmn}\}$
<i>zimmn</i>	An n-bit wide unsigned immediate. This operand is implicitly <i>zero extended</i> (using the most-significant-bits) to <i>word</i> size.	$32'b\{32-n'b0, n'b\text{zimmn}\}$
<i>immzn</i>	An n-bit wide unsigned immediate. This operand is implicitly <i>zero tailed</i> (using the least-significant-bits) to <i>word</i> size.	$32'b\{n'b\text{immzn}, 32-n'b0\}$
<i>Strimmnx2</i>	n distinct 2-bit address post-modification specifiers	
<i>enumFlags</i>	Flags to control the precise behaviour of an instruction	See instruction semantics

3.4 Memory addressing modes

3.4.1 Base address with delta and scaled unsigned register offset

Table 3.3: Base address with delta and scaled unsigned register offset

Syntax	$\$mBase, \$mDelta, \$mOff$
Effective address(es):	$EA_0 = \$mBase + \$mDelta + (\$mOff * \text{accessSizeInBytes})$
Register post-modification(s):	None
Example:	<i>ld32</i>

3.4.2 Base address with delta and scaled zero-extended immediate offset

Table 3.4: Base address with delta and scaled zero-extended immediate offset

Syntax	$\$mBase, \$mDelta, zimm12$
Effective address(es):	$EA_0 = \$mBase + \$mDelta + (zimm12 * accessSizeInBytes)$
Register post-modification(s):	None
Example:	<i>ld64</i>

3.4.3 Post-incrementing absolute address

Table 3.5: Post-incrementing absolute address

Syntax	$\$mAddr++$
Effective address(es):	$EA_0 = \$mAddr$
Register post-modification(s):	$\$mAddr += accessSizeInBytes$
Example:	<i>ld64a32</i>

3.4.4 Base address with post-incrementing delta and signed register stride

Table 3.6: Base address with post-incremented delta and signed register stride

Syntax	$\$mBase, \$mDelta +=, \$mStride$
Effective address(es):	$EA_0 = \$mBase + \$mDelta$
Register post-modification(s):	$\$mDelta += \$mStride * accessSizeInBytes$
Example:	<i>lds16step</i>

3.4.5 Base address with post-incrementing delta and signed immediate stride

Table 3.7: Base address with post-incremented delta and signed immediate stride

Syntax	$\$mBase, \$mDelta +=, simm8$
Effective address(es):	$EA_0 = \$mBase + \$mDelta$
Register post-modification(s):	$\$mDelta += simm8 * accessSizeInBytes$
Example:	<i>lds16step</i>

3.4.6 Base address with upper 16-bit delta

Table 3.8: Base address with upper 16-bit delta

Syntax	$\$mBase, \$mDelta$
Effective address(es):	$EA_0 = \$mBase + \$mDelta[31:16]$
Register post-modification(s):	None
Example:	<i>ldd16v2a32</i>

3.4.7 Base address with lower 16-bit delta, with simultaneous delta load from absolute, post-incrementing address

Table 3.9: Base address with lower 16-bit delta, with simultaneous delta load from absolute, post-incrementing address

Syntax	$\$mAddr++$, $\$mBase$, $\$mDelta@$
Effective address(es):	<ul style="list-style-type: none"> • $EA_0 = \\$mBase + \\$mDelta[15:0]$ • $EA_1 = \\$mAddr$
Register post-modification(s):	<ul style="list-style-type: none"> • $\\$mDelta = \text{loadHalfWord}(EA_1)$ • $\\$mAddr += 2$ (or 4)
Example(s):	<i>ldd16a32, ld64a32</i>

3.4.8 Base address with post-incrementing delta-pair

Table 3.10: Base address with post-incrementing delta-pair

Syntax	$\$mBase$, $\$mDelta++$, $\$mMiniD >>$
Effective address(es):	<ul style="list-style-type: none"> • $EA_0 = \\$mBase + \\$mDelta[15:0]$ • $EA_1 = \\$mBase + \\$mDelta[31:16]$
Register post-modification(s):	<ul style="list-style-type: none"> • $\\$mDelta = ((\\$mDelta[31:16] + 2) \ll 16) \mid ((\\$mDelta[15:0] + ((\\$mMiniD[3:0] + 1) \ll 1))$ • $\\$mMiniD >> = 4$ (least-significant 4 bits shifted out)
Example(s):	<i>ldb16b16</i>

3.4.9 Post-incrementing packed absolute addresses (with packed strides)

Table 3.11: Base address with post-incrementing delta-pair

Syntax	$\$mAddr0:1 +=$, $\$mStride$, $Strimmnx2$
Effective address(es):	<ul style="list-style-type: none"> • $EA_0 = \text{Tile_ExtractPackedAddress}(\\$mAddr0:1, 0)$ • $EA_1 = \text{Tile_ExtractPackedAddress}(\\$mAddr0:1, 1)$ • $EA_2 = \text{Tile_ExtractPackedAddress}(\\$mAddr0:1, 2)^1$
Register post-modification(s):	<ul style="list-style-type: none"> • $\text{stride}_0 = \text{Tile_ExtractPackedStride}(\\$mStride, Strimmnx2[1:0])$ • $\text{stride}_1 = \text{Tile_ExtractPackedStride}(\\$mStride, Strimmnx2[3:2])$ • $\text{stride}_2 = \text{Tile_ExtractPackedStride}(\\$mStride, Strimmnx2[5:4])^1$ • $\text{addrs}_0 = EA_0 + (\text{stride}_0 * 8)$ • $\text{addrs}_1 = EA_1 + (\text{stride}_1 * 8)$ • $\text{addrs}_2 = EA_2 + (\text{stride}_2 * 8)^1$ • $\\$mAddr0 = \text{Tile_TripleAddressPack_Lower}(\text{addrs})$ • $\\$mAddr1 = \text{Tile_TripleAddressPack_Upper}(\text{addrs})$
Example(s):	<i>ld2xst64pace, ld2x64pace, ldst64pace</i>

Function references: *Tile_ExtractPackedStride*, *Tile_ExtractPackedAddress*, *Tile_TripleAddressPack_Lower*, *Tile_TripleAddressPack_Upper*

¹ If applicable

3.5 Mnemonic conventions

3.5.1 Load/store instructions

Every load and store instruction mnemonic contains characters to indicate:

- The transfer direction(s):
 - ld: A load from *Tile Memory* to *Tile* register state
 - stm: A store to *Tile Memory* specifically from the *MRF*
 - st: A store to *Tile Memory* from *Tile* register state
- The access size of each transfer:
 - 8: A byte (8-bit) access
 - 16: A half-word (16-bit) access
 - 16v2: A *word* (32-bit access), with a specific vector format
 - 32: A *word* (32-bit) access
 - 64: A double-word (64-bit) access
 - 128: A quad-word (128-bit) access
- A zero, one or two letter modifier prefix to the access size:
 - No prefix indicates that the loaded value will be stored in the *Tile* register state unmodified.
 - zn: The *n*-bit wide loaded value will be *zero extended* to *word* size. 0s will be inserted into the most significant bits of the resulting *word* such that the original *n*-bit value resides in the least significant bits of the *word*.
 - sn: The *n*-bit wide loaded value will be *sign extended* to *word* size. The original sign-bit will be inserted into the most significant bits of the resulting *word* such that the newly formed *word* is numerically identical to the original.
 - bn: The *n*-bit wide loaded value will be broadcast (duplicated) to fill the target register subset.
 - an: The *n*-bit wide loaded value will be stored in the *Tile* register state unmodified. This prefix is only used as a delimiter between two distinct access sizes.
 - dn: The *n*-bit wide loaded value is implicitly treated as a delta-offset (replacing a value previously used as a delta-offset)
 - 2xn: This instruction will perform 2 independent accesses of size *n* (using identical addressing modes).
- In the case of instructions that perform a post-modification of the access address(es), a four letter suffix
 - step: The single access address is post-modified according to the value of an immediate or register
 - pace: The instruction uses triple-packed addresses, some or all of which will be post-modified according to a combination of immediate and register values.
- In the case of load instructions which don't target the *MRF* or *ARF* a suffix indicating the destination type:
 - putcs: The loaded value will be stored into the common compute configuration space

3.5.2 Floating-point instructions

Every floating-point instruction mnemonic:

- Starts with the character *f*
- which is followed by the width of the fundamental data type
 - 32: 32-bit, *single-precision*
 - 16: 16-bit, *half-precision*
- which is optionally followed by the character *v*, indicating a vector operation and if so:
 - the size of the vector:

- * 2: A 2-element vector
- * 4: A 4-element vector
- * 8: An 8-element vector

For example:

- *f32mul* is a floating-point instruction, which operates on *single-precision* scalar values.
- *f16v2sub* is a floating-point instruction, which operates on 2-element *half-precision* vectors.

3.6 Instruction execution semantics

The functional execution of a *Tile* instruction (phase 3 of an instructions *lifespan*), in terms of the operations performed and their effect on architectural state, is split into 6 sub-phases. Not all instructions perform operations in every sub-phase.

Table 3.12: Instruction execution sub-phases

Execution phase	sub-	Description
<i>Prepare</i>		Register and immediate operands are prepared with the necessary shifting, sign extension and conversions. The operations and architectural state changes performed in this phase are enacted and committed, regardless of the (subsequent) detection of exceptions.
<i>Except In</i>		Checks are performed on architectural state, instruction operands and on intermediate values calculated during the <i>Prepare</i> phase. The detection of a <i>precise</i> exception during this phase will result in the architectural updates defined by the <i>Commit</i> phase being squashed and a precise exception event will be raised.
<i>Compute</i>		The core of the operation is performed and the results computed.
<i>Memory</i>		Tile Memory transactions are performed using effective addresses calculated in previous phases.
<i>Except Out</i>		The results of the operation are checked for exceptions like result overflow. As with ‘Except In’, the detection of a <i>precise</i> exception during this phase will result in the architectural updates defined by the <i>Commit</i> phase being squashed and a precise exception being raised.
<i>Commit</i>		The architectural state changes resulting from the operations in this phase are only committed in the absence of precise exceptions detected during the <i>Exception In/Out</i> phases. Note that the architectural state changes will be committed if an <i>imprecise</i> exception is detected (in the absence of any precise exceptions).

The flow chart in *Instruction lifespan* and pseudo-code in *Instruction lifespan (part 1 of 3)* and *Instruction lifespan (part 2 of 3)* serve to illustrate the complete lifespan of an instruction:

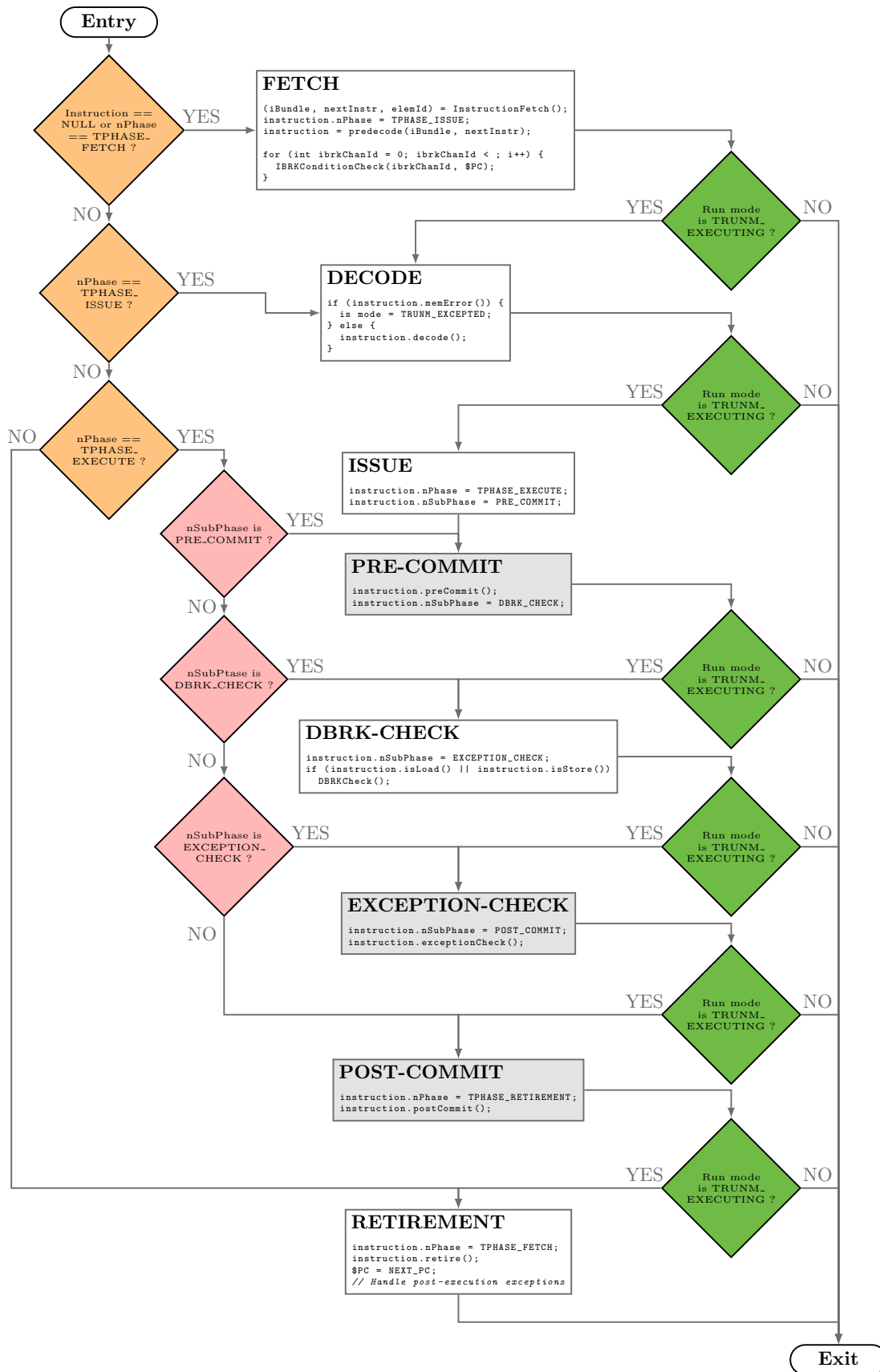


Fig. 3.1: Instruction lifespan

Listing 3.1: Instruction lifespan (part 1 of 3)

```

void context::RunMode_Executing() {
  assert(runMode == TRUNM_EXECUTING);
}
  
```

```

// Note that in the absence of exceptions, all instructions that do not perform
// a run-mode transition will progress through their entire lifespan in a single
// invocation of this method.

// In the case of exceptions and changes to run-mode, this method will need to be
// re-invoked once the run-mode has transitioned back to TRUNM_EXECUTING.

if ((instruction == NULL) || (instruction.nPhase == TPHASE_FETCH)) {
    /*-----
    INSTRUCTION FETCH PHASE
    -----*/

    if (!instruction.wasInjected()) {
        if ($TXT_STS.MERR) {
            // Memory parity/ecc error detected by another context
            EXCEPT(TEXCPT_MEMERR);
            return;
        }
    }

    // Perform instruction fetch from $PC
    (iBundle, nextInstr, elemId) = InstructionFetch();

    // No exception can be raised by pre-decode
    instruction = predecode(iBundle, nextInstr);
    instruction.nPhase = TPHASE_ISSUE;

    if (!instruction.wasInjected()) {
        // Check $PC against all enabled IBRK channels.
        // A match will cause a run-mode transition to TRUNM_EXCEPTED
        for (int ibrkChanId = 0; ibrkChanId < ; i++) {
            IBRKConditionCheck(ibrkChanId, $PC);
        }
        if (runMode != TRUNM_EXECUTING) {
            return;
        }
    }
}

// No TEXCPT_IBRK exception raised, or IBRK exception now cleared
// Run-mode is TRUNM_EXECUTING
if (instruction.nPhase == TPHASE_ISSUE) {
    /*-----
    INSTRUCTION DECODE/ISSUE PHASE
    -----*/

    // Check for memory parity/ECC error from instruction fetch
    if (instruction.memError()) {
        $TXT_STS.MERR = 0b1;
        runMode = TRUNM_EXCEPTED;
        eType = TEXCPT_MEMERR;
    } else {
        instruction.decode();
    }

    // Instruction decode may raise an exception
    if (runMode != TRUNM_EXECUTING) {
        instruction = NULL;
        return;
    }
    instruction.nPhase = TPHASE_EXECUTE;
    instruction.nSubPhase = PRE_COMMIT;
}

```

Listing 3.2: Instruction lifespan (part 2 of 3)

```

if (instruction.nPhase == TPHASE_EXECUTE) {
    /*-----
    INSTRUCTION EXECUTION PHASE
    -----*/

    if (instruction.nSubPhase == PRE_COMMIT) {
        instruction.nSubPhase = DBRK_CHECK;
        instruction.preCommit();
    }
}

```

```

// Supervisor sync instruction may have transitioned run-mode to TRUNM_WAIT_WORKERS.
// Continue here once run-mode transitions back to TRUNM_EXECUTING.
if (runMode != TRUNM_EXECUTING) {
    return;
}
}

if (instruction.nSubPhase == DBRK_CHECK) {
    instruction.nSubPhase = EXCEPTION_CHECK;

    // DBRK check following effective address calculations
    if (!instruction.wasInjected() && (instruction.isLoad() || instruction.isStore())) {
        DBRKCheck();

        if (runMode != TRUNM_EXECUTING) {
            return;
        }
    }
}
// No TEXCPT_DBRK exception raised, or DBRK exception now cleared

if (instruction.nSubPhase == EXCEPTION_CHECK) {
    instruction.nSubPhase = POST_COMMIT;
    instruction.exceptionCheck();

    // If a non-debug exception was raised during exceptionCheck(),
    // instruction terminates here and doesn't enter post-commit.
    if (runMode != TRUNM_EXECUTING) {
        return;
    }
}

// If a debug exception was raised during exceptionCheck(), instruction execution will
// resume here once the exception is cleared.
if (instruction.nSubPhase == POST_COMMIT) {
    instruction.nPhase = TPHASE_RETIRE;
    instruction.postCommit();

    if (runMode != TRUNM_EXECUTING) {
        return;
    }
}
}
// A run-mode transition may have occurred during postCommit();
// Instruction execution will resume here once
// run-mode transitions back to TRUNM_EXECUTING.

if (instruction.nPhase == TPHASE_RETIRE) {
    /*-----
    INSTRUCTION RETIREMENT PHASE
    -----*/

    instruction.nPhase = TPHASE_FETCH;
    instruction.retire();

    if (takenBranchInsn) {
        // Either a taken solo branch instruction, an
        // Execution Bundle containing a taken branch
        // or sendpicp
        $PC = TARGET_PC;
    } else if (!instruction.wasInjected()) {
        // $PC := $PC + 4 for a solo instruction
        // $PC := ($PC + 4) + 4 for Execution Bundle
        $PC = $PC + 4;
    }
}

```

Listing 3.3: Instruction lifespan (part 3 of 3)

```

/*-----
INSTRUCTION RETIREMENT PHASE (cont'd)
-----*/

if (!instruction.wasInjected()) {

```

```

// Handle post-execution exceptions
// Check for asynchronous exception events
if (($TDI_CTL.SEPEX == 0) && $CTXT_STS.EERR) {
    EXCEPT(TEXCPT_EXERR);
    return;
}

// If there are active worker contexts when runall is executed
// the Supervisor and those active contexts will except.
if ($SSR.RAERR) {
    EXCEPT(TEXCPT_INVALID_INSTR);
    return;
}

if ($CTXT_STS.ERERR) {
    EXCEPT(($CTXT_STS.ERERR < TEXCH_RERR_ADDR) ? TEXCPT_CONFLICT : TEXCPT_EXCONF);
    return;
}

// RBRK check
if (instruction.canRaiseRBRK()
    && (((DBG_RBRK.VM_EN == 0) && DBG_RBRK.EN[ctxtId])
        || ((DBG_RBRK.VM_EN != 0) && (DBG_RBRK_VERT == VERTEX_BASE)))) {
    EXCEPT(TEXCPT_RBRK);
    return;
}

// nextRunMode set by exit instructions
if (nextRunMode != runMode) {
    setRunMode(nextRunMode);
}
}
}
}
}

```

Table 3.13: Common functions/methods/syntax

Function/Syntax	Arguments	Description
SignExtend32	<i>value</i> , <i>signBit</i>	Sign extend <i>value</i> to 32-bits by replicating the sign-bit at bit-position <i>signBit</i> in <i>value</i> into the most significant bits of the 32-bit result.
getGlobalConsensus	None	Returns current value on global consensus input wires
bit16	<i>infinityBehav</i> = <i>TFPU_4CE_INF_NANOO</i>	Returns the raw (16-bit) bit-pattern of a half-precision floating-point value. <i>infinityBehav</i> specifies how to treat infinity values (see bitz32())
bitz32	<i>infinityBehav</i> = <i>TFPU_4CE_INF_NANOO</i>	Returns the raw bit-pattern of a half-precision value, <i>zero extended</i> to 32-bits. <i>infinityBehav</i> specifies how to treat infinity values: <ul style="list-style-type: none"> • <i>TFPU_4CE_INF_NANOO</i>: Saturate or generate qNaN as specified by <i>\$FP_CTL.NANOO</i> • <i>TFPU_4CE_INF_PROP</i>: Return unmodified infinity value • <i>TFPU_4CE_INF_SAT</i>: infinities map to +-65504 • <i>TFPU_4CE_INF_QNAN</i>: infinities convert to quiet NaN
floatAsUint32		Returns the raw 32-bit bit-pattern of a single-precision value.
uint32AsFloat		Returns the single-precision value represented by a raw 32-bit bit-pattern.
pickHalf	<i>regIndex</i> , <i>zeroOrOne</i> , <i>propagateInf</i> =false	<ul style="list-style-type: none"> • If <i>zeroOrOne</i> is 0, returns the half-precision floating-point value residing in the bottom 16-bits of the register at index <i>regIndex</i>. • Otherwise returns the half-precision floating-point value residing in the upper 16-bits of the register at index <i>regIndex</i> • When <i>propagateInf</i> is true, infinity values are left unmodified. When <i>propagateInf</i> is false (the default), infinities are saturated or converted to signalling NaNs depending on the setting of <i>\$FP_CTL.NANOO</i>.
EXCEPT	<i>exceptionId</i>	Begins process of exception handling, using the exception Id provided. Note that any exception raised when <i>\$REPEAT_COUNT</i> is non-zero will automatically set <i>\$WSR.ERPT</i> to 0b1 and result in a <i>malign</i> exception launch. See Exception model
isValidAddress	<i>address</i>	Returns True if and only if <i>address</i> is a valid <i>Tile Memory</i> address
hadMemoryConflict	<i>address</i>	Returns True if and only if there is a <i>memory-element conflict</i> involving this memory access.
isValidState	<i>id</i>	Returns True if and only if <i>id</i> is a valid <i>CSR</i> identifier.
readState	<i>csrIndex</i>	Returns the current value of the <i>CSR</i> at index <i>csrIndex</i>
writeState	<i>csrIndex</i> , <i>value</i>	Writes value to <i>CSR</i> at index <i>csrIndex</i>
setExitValue	<i>exitBool</i>	<ul style="list-style-type: none"> • Set <i>Worker</i> exit status to <i>exitBool</i>. • <i>Supervisor \$SSR.LC</i> &= <i>exitBool</i>
<(op0, op1)>		The source or destination operand used is dependent on the format of the instruction being executed.

3.7 Instructions by class

3.7.1 Bit

Table 3.14: bit instructions summary

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>and</i>	✓	✓	✓	✓	✓	32-bit bitwise logical AND
<i>and64</i>	✓	✗	✓	✗	✓	64-bit bitwise logical AND
<i>andc</i>	✓	✓	✓	✓	✓	32-bit bitwise logical AND Complement
<i>andc64</i>	✓	✗	✓	✗	✓	64-bit bitwise logical AND Complement
<i>bitrev8</i>	✓	✓	✓	✓	✗	Byte-wise bit order reversal
<i>clz</i>	✓	✓	✓	✓	✗	Count leading zero bits
<i>cms</i>	✓	✓	✓	✓	✗	Count matching sign bits
<i>not</i>	✓	✗	✓	✗	✓	32-bit bitwise logical NOT
<i>not64</i>	✓	✗	✓	✗	✓	64-bit bitwise logical NOT
<i>or</i>	✓	✓	✓	✓	✓	Bitwise OR
<i>or64</i>	✓	✗	✓	✗	✓	64-bit bitwise logical OR
<i>popc</i>	✓	✓	✓	✓	✗	Population count
<i>roll16</i>	✓	✓	✓	✓	✓	roll16 SIMD permutation
<i>roll32</i>	✓	✗	✓	✗	✓	roll32 SIMD permutation
<i>roll8l</i>	✓	✓	✓	✓	✗	roll8-left SIMD permutation
<i>roll8r</i>	✓	✓	✓	✓	✗	roll8-right SIMD permutation
<i>setzi</i>	✓	✓	✓	✓	✓	Register set from immediate
<i>shuf8x8hi</i>	✓	✓	✓	✓	✗	8 x 8-bit SIMD permutation
<i>shuf8x8lo</i>	✓	✓	✓	✓	✗	8 x 8-bit SIMD permutation
<i>sort4x16hi</i>	✓	✓	✓	✓	✓	4 x 16-bit SIMD permutation
<i>sort4x16lo</i>	✓	✓	✓	✓	✓	4 x 16-bit SIMD permutation
<i>sort4x32hi</i>	✓	✗	✓	✗	✓	4 x 32-bit SIMD permutation
<i>sort4x32lo</i>	✓	✗	✓	✗	✓	4 x 32-bit SIMD permutation
<i>sort8</i>	✓	✓	✓	✓	✗	4 x 8-bit SIMD permutation
<i>sort8x8hi</i>	✓	✓	✓	✓	✗	8 x 8-bit SIMD permutation
<i>sort8x8lo</i>	✓	✓	✓	✓	✗	8 x 8-bit SIMD permutation
<i>swap8</i>	✓	✓	✓	✓	✗	4 x 8-bit SIMD permutation
<i>xnor</i>	✓	✓	✓	✓	✗	Bitwise NOT XOR
<i>xor</i>	✓	✓	✓	✓	✗	Bitwise XOR

3.7.1.1 and

Bitwise logical AND of two source register values, or 1 source register and 1 *zero extended/zero tailed* immediate.

Table 3.15: *and* instruction definition

<i>and</i>		both	main
Syntax	<code>and \$mDst0, \$mSrc0, \$mSrc1</code> <code>and \$mDst0, \$mSrc0, zimm12</code>		
<i>and</i>		worker	aux
Syntax	<code>and \$aDst0, \$aSrc0, \$aSrc1</code> <code>and \$aDst0, \$aSrc0, immz12</code> <code>and \$aDst0, \$aSrc0, zimm12</code>		
Semantics	Prepare	<code>DataWord op1 = <(\$mSrc0, \$aSrc0)>;</code> <code>DataWord op2 = <(\$mSrc1, zimm12, \$aSrc1, (immz12 << 20))>;</code>	
	Compute	<code>DataWord result = op1 & op2;</code>	
	Commit	<code><(\$mDst0, \$aDst0)> = result;</code>	

3.7.1.2 *and64*

64-bit bitwise logical AND of two *ARF* source register-pairs.

Table 3.16: *and64* instruction definition

<i>and64</i>		worker	aux
Syntax	<code>and64 \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>		
Semantics	Prepare	<code>array<DataWord,2> op1 = {</code> <code> \$aSrc0:Src0+1[0],</code> <code> \$aSrc0:Src0+1[1]</code> <code>};</code> <code>array<DataWord,2> op2 = {</code> <code> \$aSrc1:Src1+1[0],</code> <code> \$aSrc1:Src1+1[1]</code> <code>};</code> <code>array<DataWord,2> result;</code>	
	Compute	<code>result[0] = op1[0] & op2[0];</code> <code>result[1] = op1[1] & op2[1];</code>	
	Commit	<code>\$aDst0:Dst0+1 = {</code> <code> result[0],</code> <code> result[1]</code> <code>};</code>	

3.7.1.3 *andc*

Bitwise logical AND of first source register value with the bitwise negated value of a second source register value or *zero extended/zero tailed* immediate.

Table 3.17: *andc* instruction definition

<i>andc</i>		both	main
Syntax	<pre>andc \$mDst0, \$mSrc0, \$mSrc1 andc \$mDst0, \$mSrc0, zimm12</pre>		
<i>andc</i>		worker	aux
Syntax	<pre>andc \$aDst0, \$aSrc0, \$aSrc1 andc \$aDst0, \$aSrc0, immz12 andc \$aDst0, \$aSrc0, zimm12</pre>		
Semantics	Prepare	<pre>DataWord op1 = <(\$mSrc0, \$aSrc0)>; DataWord op2 = <(\$mSrc1, zimm12, \$aSrc1, (immz12 << 20))>;</pre>	
	Compute	<pre>DataWord result = op1 & ~op2;</pre>	
	Commit	<pre><(\$mDst0, \$aDst0)> = result;</pre>	

3.7.1.4 *andc64*

64-bit bitwise logical AND of first *ARF* source register-pair with the bitwise negated value of a second *ARF* source register-pair.

Table 3.18: *andc64* instruction definition

<i>andc64</i>		worker	aux
Syntax	<pre>andc64 \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</pre>		
Semantics	Prepare	<pre>array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,2> op2 = { \$aSrc1:Src1+1[0], \$aSrc1:Src1+1[1] }; array<DataWord,2> result;</pre>	
	Compute	<pre>result[0] = op1[0] & ~op2[0]; result[1] = op1[1] & ~op2[1];</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { result[0], result[1] };</pre>	

3.7.1.5 *bitrev8*

Reverse the bit order of each bit inside each byte of a register.

Table 3.19: *bitrev8* instruction definition

<i>bitrev8</i>		both	main
Syntax	bitrev8 \$mDst0, \$mSrc0		
Semantics	Prepare	DataWord op1 = \$mSrc0;	
	Compute	<pre>DataWord result = 0; for (i = 0; i < 32; i++) { if (op1 & (1 << i)) { result = 1 << ((i & 24) (7-(i & 7))); } }</pre>	
	Commit	\$mDst0 = result;	

3.7.1.6 clz

Establishes the number of higher order bits that are zero. An unsigned interpretation of the source value can be stored in (32 - result) bits without loss.

Table 3.20: *clz* instruction definition

<i>clz</i>		both	main
Syntax	clz \$mDst0, \$mSrc0		
Semantics	Prepare	DataWord op1 = \$mSrc0;	
	Compute	<pre>int i = 31; while (i >= 0 && ((op1 >> i) & 1) == 0) { i = i - 1; } SignedDataWord result = 31 - i;</pre>	
	Commit	\$mDst0 = result;	

3.7.1.7 cms

Establishes the number of higher order bits that match the sign-bit (bit 31). Result is always in the range [0..31].

Table 3.21: *cms* instruction definition

<i>cms</i>		both	main
Syntax	cms \$mDst0, \$mSrc0		
Semantics	Prepare	SignedDataWord op1 = \$mSrc0;	
	Compute	<pre>unsigned signBit = ((op1 >> 31) & 1); int i = 31; while ((i > 0) && (signBit == ((op1 >> (i - 1)) & 1))) { i = i - 1; } SignedDataWord result = 31 - i;</pre>	
	Commit	\$mDst0 = result;	

3.7.1.8 not

Compute the bitwise logical NOT of a single 32-bit *ARF* register.

Table 3.22: *not* instruction definition

<i>not</i>		worker	aux
Syntax	not <i>\$aDst0</i> , <i>\$aSrc0</i>		
Semantics	Prepare	DataWord op1 = <i>\$aSrc0</i> ;	
	Compute	DataWord result = ~op1;	
	Commit	<i>\$aDst0</i> = result;	

3.7.1.9 not64

Compute the bitwise logical NOT of a 64-bit *ARF* register-pair.

Table 3.23: *not64* instruction definition

<i>not64</i>		worker	aux
Syntax	not64 <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0:Src0+1</i>		
Semantics	Prepare	array<DataWord,2> op1 = { <i>\$aSrc0:Src0+1</i> [0], <i>\$aSrc0:Src0+1</i> [1] }; array<DataWord,2> result;	
	Compute	result[0] = ~op1[0]; result[1] = ~op1[1];	
	Commit	<i>\$aDst0:Dst0+1</i> = { result[0], result[1] };	

3.7.1.10 or

Compute the bitwise-OR of 1 32-bit register source value with 1 32-bit register or 1 *zero extended/zero tailed* 12-bit immediate value.

Table 3.24: *or* instruction definition

<i>or</i>		both	main
Syntax	or <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> or <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>immz12</i> or <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>zimm12</i>		
<i>or</i>		worker	aux
Syntax	or <i>\$aDst0</i> , <i>\$aSrc0</i> , <i>\$aSrc1</i> or <i>\$aDst0</i> , <i>\$aSrc0</i> , <i>immz12</i> or <i>\$aDst0</i> , <i>\$aSrc0</i> , <i>zimm12</i>		
Semantics	Prepare	DataWord op1 = <(<i>\$mSrc0</i> , <i>\$aSrc0</i>)>; DataWord op2 = <(<i>\$mSrc1</i> , <i>zimm12</i> , (<i>immz12</i> << 20), <i>\$aSrc1</i>)>;	
	Compute	DataWord result = op1 op2;	
	Commit	<(<i>\$mDst0</i> , <i>\$aDst0</i>)> = result;	

or occurs in the following code examples:

- *ldb16b16 example*

3.7.1.11 or64

Compute the bitwise logical OR of 1 *ARF* register-pair source value with a 2nd *ARF* register-pair.

Table 3.25: *or64* instruction definition

<i>or64</i>		worker	aux
Syntax	or64 \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1		
Semantics	Prepare	<pre>array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,2> op2 = { \$aSrc1:Src1+1[0], \$aSrc1:Src1+1[1] }; array<DataWord,2> result;</pre>	
	Compute	<pre>result[0] = op1[0] op2[0]; result[1] = op1[1] op2[1];</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { result[0], result[1] };</pre>	

3.7.1.12 popc

Establishes the number of set bits in a 32-bit register source value.

Table 3.26: *popc* instruction definition

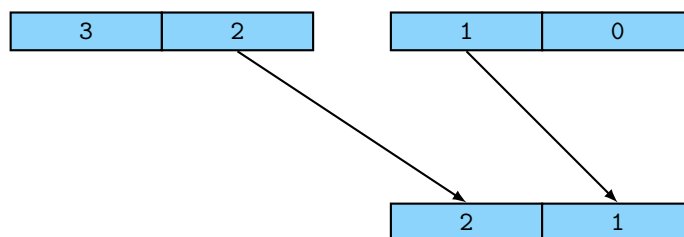
<i>popc</i>		both	main
Syntax	popc \$mDst0, \$mSrc0		
Semantics	Prepare	DataWord op1 = \$mSrc0;	
	Compute	<pre>DataWord count = 0; for (i = 0; i < 32; i++) { count += (op1 >> i) & 1; }</pre>	
	Commit	\$mDst0 = count;	

3.7.1.13 roll16

Perform a SIMD *roll* permutation on the 4 x 16-bit values across 2 source registers. Equivalent to a SIMD *roll* operation on 8 x 8-bit values.

Table 3.27: *roll16* instruction definition

<i>roll16</i>	both	main
Syntax	<code>roll16 \$mDst0, \$mSrc0, \$mSrc1</code>	
<i>roll16</i>	worker	aux
Syntax	<code>roll16 \$aDst0, \$aSrc0, \$aSrc1</code>	
Semantics	<p>Prepare</p> <pre> array<HalfDataWord,2> op0; array<HalfDataWord,2> op1 = { ((<\$mSrc0, \$aSrc0)>[0] >> 0) & 0xffff), ((<\$mSrc0, \$aSrc0)>[0] >> 16) & 0xffff) }; array<HalfDataWord,2> op2 = { ((<\$mSrc1, \$aSrc1)>[0] >> 0) & 0xffff), ((<\$mSrc1, \$aSrc1)>[0] >> 16) & 0xffff) }; </pre> <p>Compute</p> <pre>op0 = { op1[1], op2[0] };</pre> <p>Commit</p> <pre> <(\$mDst0, \$aDst0)> = { ((op0[0] & 0xffff) << 0) ((op0[1] & 0xffff) << 16) }; </pre>	

Fig. 3.2: *roll16*

3.7.1.14 *roll32*

Perform a SIMD *roll* permutation on the 4 x 32-bit values across 2 source registers-pairs.

Table 3.28: *roll32* instruction definition

<i>roll32</i>	worker	aux
Syntax	<code>roll32 \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>	
Semantics	<p>Prepare</p> <pre> array<DataWord,2> op0; array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,2> op2 = { \$aSrc1:Src1+1[0], \$aSrc1:Src1+1[1] }; </pre> <p>Compute</p> <pre>op0 = { op1[1], op2[0] };</pre> <p>Commit</p> <pre> \$aDst0:Dst0+1 = { op0[0], op0[1] }; </pre>	

3.7.1.15 roll8l

Perform a SIMD *roll-left* permutation on the 8 x 8-bit values across 2 source registers.

Table 3.29: *roll8l* instruction definition

<i>roll8l</i>	both	main
Syntax	<code>roll8l \$mDst0, \$mSrc0, \$mSrc1</code>	
Semantics Prepare	<pre> array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$Src0[0] >> 0) & 0xff), ((\$Src0[0] >> 8) & 0xff), ((\$Src0[0] >> 16) & 0xff), ((\$Src0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$Src1[0] >> 0) & 0xff), ((\$Src1[0] >> 8) & 0xff), ((\$Src1[0] >> 16) & 0xff), ((\$Src1[0] >> 24) & 0xff) }; </pre>	
Compute	<code>op0 = { op1[3], op2[0], op2[1], op2[2] };</code>	
Commit	<pre> \$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) }; </pre>	

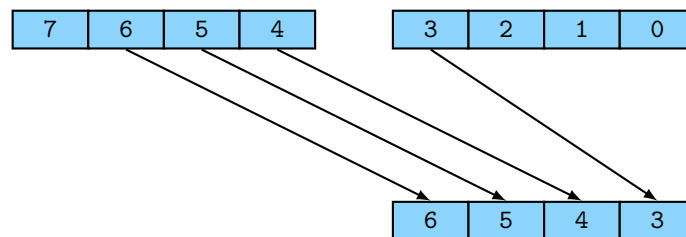


Fig. 3.3: roll8l

3.7.1.16 roll8r

Perform a SIMD *roll-right* permutation on the 8 x 8-bit values across 2 source registers.

Table 3.30: *roll8r* instruction definition

<i>roll8r</i>		both	main
Syntax	<code>roll8r \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	<pre> array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$mSrc1[0] >> 0) & 0xff), ((\$mSrc1[0] >> 8) & 0xff), ((\$mSrc1[0] >> 16) & 0xff), ((\$mSrc1[0] >> 24) & 0xff) }; </pre>	
	Compute	<code>op0 = { op1[1], op1[2], op1[3], op2[0] };</code>	
	Commit	<pre> \$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) }; </pre>	

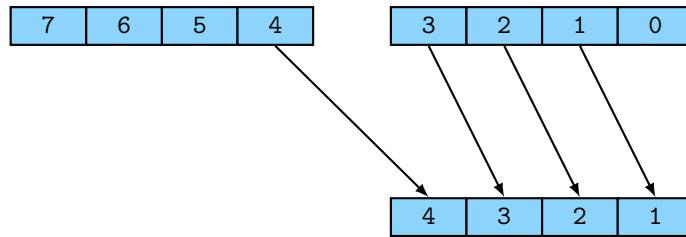


Fig. 3.4: *roll8r*

3.7.1.17 *setzi*

Set register to *zero extended* 20-bit immediate value.

Table 3.31: *setzi* instruction definition

<i>setzi</i>		both	main
Syntax	<code>setzi \$mDst0, zimm20</code>		
<i>setzi</i>		worker	aux
Syntax	<code>setzi \$aDst0, zimm20</code>		
Semantics	Prepare	<code>DataWord op1 = zimm20;</code>	
	Commit	<code><(\$mDst0, \$aDst0)> = op1;</code>	

setzi occurs in the following code examples:

- *f16v4stacc example*
- *ldb16b16 example*

3.7.1.18 *shuf8x8hi*

Perform SIMD *shuffle* permutation on 8 x 8-bit values, across 2 source registers, returning the upper *word* of the result. See *shuf8x8lo* for lower *word*.

Table 3.32: *shuf8x8hi* instruction definition

<i>shuf8x8hi</i>		both	main
Syntax	<code>shuf8x8hi \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	<pre> array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$mSrc1[0] >> 0) & 0xff), ((\$mSrc1[0] >> 8) & 0xff), ((\$mSrc1[0] >> 16) & 0xff), ((\$mSrc1[0] >> 24) & 0xff) }; </pre>	
	Compute	<pre> op0 = { op1[2], op2[2], op1[3], op2[3] }; </pre>	
	Commit	<pre> \$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) }; </pre>	

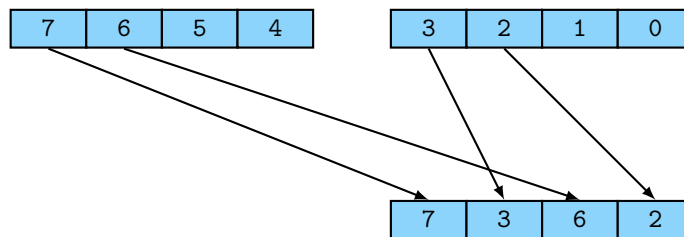


Fig. 3.5: *shuf8x8hi*

3.7.1.19 *shuf8x8lo*

Perform SIMD *shuffle* permutation on 8 x 8-bit values, across 2 source registers, returning the lower *word* of the result. See *shuf8x8hi* for upper *word*.

Table 3.33: *shuf8x8lo* instruction definition

<i>shuf8x8lo</i>		both	main
Syntax	<code>shuf8x8lo \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	<pre> array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$Src0[0] >> 0) & 0xff), ((\$Src0[0] >> 8) & 0xff), ((\$Src0[0] >> 16) & 0xff), ((\$Src0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$Src1[0] >> 0) & 0xff), ((\$Src1[0] >> 8) & 0xff), ((\$Src1[0] >> 16) & 0xff), ((\$Src1[0] >> 24) & 0xff) }; </pre>	
	Compute	<code>op0 = { op1[0], op2[0], op1[1], op2[1] };</code>	
	Commit	<pre> \$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) }; </pre>	

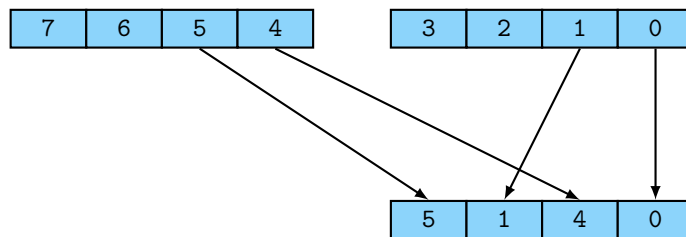


Fig. 3.6: *shuf8x8lo*

3.7.1.20 *sort4x16hi*

Perform SIMD *sort* permutation on 4 x 16-bit values, across 2 source registers, producing a 2 x 16-bit result.

Table 3.34: *sort4x16hi* instruction definition

<i>sort4x16hi</i>		both	main
Syntax	sort4x16hi \$mDst0, \$mSrc0, \$mSrc1		
<i>sort4x16hi</i>		worker	aux
Syntax	sort4x16hi \$aDst0, \$aSrc0:BL, \$aSrc1 sort4x16hi \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	<pre> array<HalfDataWord,2> op0; array<HalfDataWord,2> op1 = { ((<(\$mSrc0, \$aSrc0, \$aSrc0:BL)>[0] >> 0) & 0xffff), ((<(\$mSrc0, \$aSrc0, \$aSrc0:BL)>[0] >> 16) & 0xffff) }; array<HalfDataWord,2> op2 = { ((<(\$mSrc1, \$aSrc1)>[0] >> 0) & 0xffff), ((<(\$mSrc1, \$aSrc1)>[0] >> 16) & 0xffff) }; </pre>	
	Compute	op0 = { op1[1], op2[1] };	
	Commit	<pre> <(\$mDst0, \$aDst0)> = { ((op0[0] & 0xffff) << 0) ((op0[1] & 0xffff) << 16) }; </pre>	

3.7.1.21 *sort4x16lo*

Perform SIMD *sort* permutation on 4 x 16-bit values, across 2 source registers, producing a 2 x 16-bit result.

Table 3.35: *sort4x16lo* instruction definition

<i>sort4x16lo</i>		both	main
Syntax	sort4x16lo \$mDst0, \$mSrc0, \$mSrc1		
<i>sort4x16lo</i>		worker	aux
Syntax	sort4x16lo \$aDst0, \$aSrc0, \$aSrc1 sort4x16lo \$aDst0, \$aSrc0:BU, \$aSrc1		
Semantics	Prepare	<pre> array<HalfDataWord,2> op0; array<HalfDataWord,2> op1 = { ((<(\$mSrc0, \$aSrc0, \$aSrc0:BU)>[0] >> 0) & 0xffff), ((<(\$mSrc0, \$aSrc0, \$aSrc0:BU)>[0] >> 16) & 0xffff) }; array<HalfDataWord,2> op2 = { ((<(\$mSrc1, \$aSrc1)>[0] >> 0) & 0xffff), ((<(\$mSrc1, \$aSrc1)>[0] >> 16) & 0xffff) }; </pre>	
	Compute	op0 = { op1[0], op2[0] };	
	Commit	<pre> <(\$mDst0, \$aDst0)> = { ((op0[0] & 0xffff) << 0) ((op0[1] & 0xffff) << 16) }; </pre>	

3.7.1.22 *sort4x32hi*

Perform SIMD *sort* permutation on 4 x 32-bit values, across 2 source register-pairs, producing a 2 x 32-bit result.

Table 3.36: *sort4x32hi* instruction definition

<i>sort4x32hi</i>		worker	aux
Syntax	<i>sort4x32hi</i> $\$aDst0:Dst0+1$, $\$aSrc0:Src0+1$, $\$aSrc1:Src1+1$		
Semantics	Prepare	<pre> array<DataWord,2> op0; array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,2> op2 = { \$aSrc1:Src1+1[0], \$aSrc1:Src1+1[1] }; </pre>	
	Compute	<pre> op0 = { op1[1], op2[1] }; </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { op0[0], op0[1] }; </pre>	

3.7.1.23 *sort4x32lo*

Perform SIMD *sort* permutation on 4 x 32-bit values, across 2 source register-pairs, producing a 2 x 32-bit result.

Table 3.37: *sort4x32lo* instruction definition

<i>sort4x32lo</i>		worker	aux
Syntax	<i>sort4x32lo</i> $\$aDst0:Dst0+1$, $\$aSrc0:Src0+1$, $\$aSrc1:Src1+1$		
Semantics	Prepare	<pre> array<DataWord,2> op0; array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,2> op2 = { \$aSrc1:Src1+1[0], \$aSrc1:Src1+1[1] }; </pre>	
	Compute	<pre> op0 = { op1[0], op2[0] }; </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { op0[0], op0[1] }; </pre>	

3.7.1.24 *sort8*

Perform SIMD *sort8* permutation on 4 x 8-bit values.

Table 3.38: *sort8* instruction definition

<i>sort8</i>		both	main
Syntax	<code>sort8 \$mDst0, \$mSrc0</code>		
Semantics	Prepare	<pre>array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) };</pre>	
	Compute	<pre>op0 = { op1[0], op1[2], op1[1], op1[3] };</pre>	
	Commit	<pre>\$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

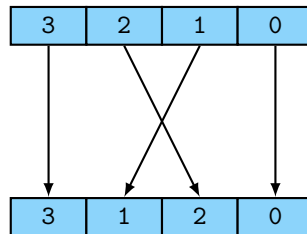


Fig. 3.7: *sort8*

3.7.1.25 *sort8x8hi*

Perform SIMD *sort* permutation on 8 x 8-bit values, across 2 source registers, returning the upper *word* of the result. See *sort8x8lo* for lower *word*.

Table 3.39: *sort8x8hi* instruction definition

<i>sort8x8hi</i>		both	main
Syntax	<code>sort8x8hi \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	<pre> array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$mSrc1[0] >> 0) & 0xff), ((\$mSrc1[0] >> 8) & 0xff), ((\$mSrc1[0] >> 16) & 0xff), ((\$mSrc1[0] >> 24) & 0xff) }; </pre>	
	Compute	<pre> op0 = { op1[1], op1[3], op2[1], op2[3] }; </pre>	
	Commit	<pre> \$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) }; </pre>	

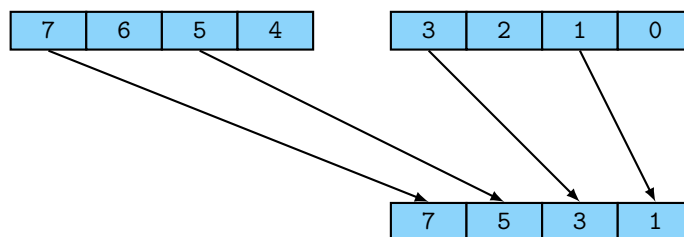


Fig. 3.8: *sort8x8hi*

3.7.1.26 *sort8x8lo*

Perform SIMD *sort* permutation on 8 x 8-bit values, across 2 source registers, returning the lower *word* of the result. See *sort8x8hi* for upper *word*.

Table 3.40: *sort8x8lo* instruction definition

<i>sort8x8lo</i>		both	main
Syntax	<code>sort8x8lo \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	<pre>array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) }; array<QuarterDataWord,4> op2 = { ((\$mSrc1[0] >> 0) & 0xff), ((\$mSrc1[0] >> 8) & 0xff), ((\$mSrc1[0] >> 16) & 0xff), ((\$mSrc1[0] >> 24) & 0xff) };</pre>	
	Compute	<code>op0 = { op1[0], op1[2], op2[0], op2[2] };</code>	
	Commit	<pre>\$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

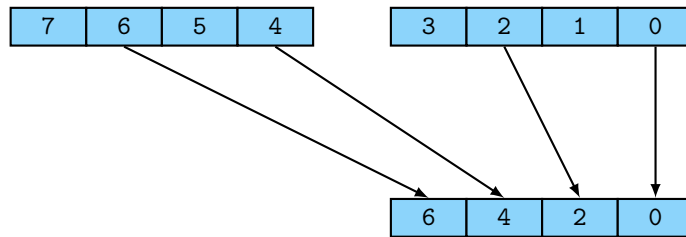


Fig. 3.9: *sort8x8lo*

3.7.1.27 *swap8*

Perform *swap* SIMD permutation on 4 x 8-bit values.

Table 3.41: *swap8* instruction definition

<i>swap8</i>		both	main
Syntax	<code>swap8 \$mDst0, \$mSrc0</code>		
Semantics	Prepare	<pre>array<QuarterDataWord,4> op0; array<QuarterDataWord,4> op1 = { ((\$mSrc0[0] >> 0) & 0xff), ((\$mSrc0[0] >> 8) & 0xff), ((\$mSrc0[0] >> 16) & 0xff), ((\$mSrc0[0] >> 24) & 0xff) };</pre>	
	Compute	<code>op0 = { op1[1], op1[0], op1[3], op1[2] };</code>	
	Commit	<pre>\$mDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

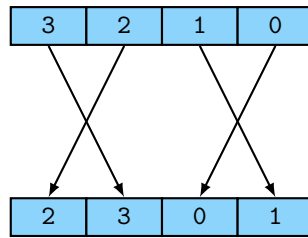


Fig. 3.10: swap8

3.7.1.28 xnor

The complement of the bitwise-XOR of two register values.

Table 3.42: *xnor* instruction definition

<i>xnor</i>		both	main
Syntax	<code>xnor \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	DataWord op1 = \$mSrc0; DataWord op2 = \$mSrc1;	
	Compute	DataWord result = ~(op1 ^ op2);	
	Commit	\$mDst0 = result;	

3.7.1.29 xor

Bitwise-XOR of two register values.

Table 3.43: *xor* instruction definition

<i>xor</i>		both	main
Syntax	<code>xor \$mDst0, \$mSrc0, \$mSrc1</code>		
Semantics	Prepare	DataWord op1 = \$mSrc0; DataWord op2 = \$mSrc1;	
	Compute	DataWord result = op1 ^ op2;	
	Commit	\$mDst0 = result;	

3.7.2 Control

Table 3.44: control instructions summary

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>br</i>	✓	✓	✓	✓	✗	Unconditional absolute branch to register target
<i>bri</i>	✓	✓	✓	✓	✗	Unconditional absolute branch to immediate target
<i>brneg</i>	✓	✓	✓	✓	✗	Branch if negative
<i>brnz</i>	✓	✓	✓	✓	✗	Branch if not zero
<i>brnzdec</i>	✓	✓	✓	✓	✗	Branch if not zero, with counter decrement
<i>brpos</i>	✓	✓	✓	✓	✗	Branch if positive
<i>brz</i>	✓	✓	✓	✓	✗	Branch if zero
<i>call</i>	✓	✓	✓	✓	✗	Function call
<i>exitneg</i>	✓	✗	✓	✓	✗	Worker thread termination
<i>exitnz</i>	✓	✗	✓	✓	✗	Worker thread termination
<i>exitpos</i>	✓	✗	✓	✓	✗	Worker thread termination
<i>exitz</i>	✓	✗	✓	✓	✗	Worker thread termination
<i>rpt</i>	✓	✗	✓	✓	✗	Repeat a sequence of <i>Execution Bundles</i>

3.7.2.1 br

Unconditional absolute branch to register target address.

Table 3.45: *br* instruction definition

<i>br</i>	both	main
Syntax	br \$mSrc0	
Semantics	Prepare DataWord op0 = \$mSrc0; DataWord target = op0;	
Except In	<pre> if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (target & 0x3) { // Misaligned target \$PC EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); } </pre>	
Commit	TARGET_PC = target;	

Architectural state references: *\$REPEAT_COUNT* , *\$PC*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.2 bri

Unconditional branch to absolute address. Immediate provides word-addressed absolute destination address.

Table 3.46: *bri* instruction definition

<i>bri</i>		both	main
Syntax	<i>bri zimm19</i>		
Semantics	Prepare	<pre>DataWord op0 = (zimm19 << 2); DataWord target = op0;</pre>	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); }</pre>	
	Commit	<pre>TARGET_PC = target;</pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.3 brneg

Conditional branch to absolute address. Branch taken if and only if register value is **negative**. Immediate provides word-addressed absolute destination address.

Table 3.47: *brneg* instruction definition

<i>brneg</i>		both	main
Syntax	<i>brneg \$mSrc0, zimm19</i>		
Semantics	Prepare	<pre>DataWord op0 = \$mSrc0; DataWord op1 = (zimm19 << 2); DataWord target = op1;</pre>	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); }</pre>	
	Commit	<pre>if ((op0 >> 31) & 1) { TARGET_PC = target; }</pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.4 brnz

Conditional branch to absolute address. Branch taken if and only if register value **is not 0**. Immediate provides word-addressed absolute destination address.

Note: This instruction considers the floating-point *single-precision* value -0.0 to not be zero (+0.0)

Table 3.48: *brnz* instruction definition

<i>brnz</i>		both	main
Syntax	brnz \$mSrc0, zimm19		
Semantics	Prepare	DataWord op0 = \$mSrc0; DataWord op1 = (zimm19 << 2); DataWord target = op1;	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); }</pre>	
	Commit	<pre>if (op0 != 0) { TARGET_PC = target; }</pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.5 brnzdec

Conditional branch to absolute address with counter decrement. Branch taken and counter value decremented by 1 if and only if counter register value **is not 0**. Immediate provides word-addressed absolute destination address.

Note: This instruction considers the floating-point *single-precision* value -0.0 to not be equal to zero (+0.0)

Table 3.49: *brnzdec* instruction definition

<i>brnzdec</i>		both	main
Syntax	brnzdec <i>\$mSrcDst0</i> , <i>zimm19</i>		
Semantics	Prepare	<pre>DataWord op0 = \$mSrcDst0; DataWord op1 = (zimm19 << 2); DataWord target = op1;</pre>	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); }</pre>	
	Commit	<pre>if (op0 != 0) { TARGET_PC = target; } \$mSrcDst0 = op0 - 1;</pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress*, *TMem_AddressIsExecutable*

brnzdec occurs in the following code examples:

- *f16v4stacc example*

3.7.2.6 brpos

Conditional branch to absolute address. Branch taken if and only if register value is **positive**. Immediate provides word-addressed absolute destination address.

Table 3.50: *brpos* instruction definition

<i>brpos</i>		both	main
Syntax	<i>brpos</i> \$mSrc0, <i>zimm19</i>		
Semantics	Prepare	DataWord op0 = \$mSrc0; DataWord op1 = (<i>zimm19</i> << 2); DataWord target = op1;	
	Except In	<pre> if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); } </pre>	
	Commit	<pre> if ((op0 >> 31) & 1) == 0) { TARGET_PC = target; } </pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress*, *TMem_AddressIsExecutable*

3.7.2.7 *brz*

Conditional branch to absolute address. Branch taken if and only if register value is 0. Immediate provides word-addressed absolute destination address.

Note: This instruction considers the floating-point *single-precision* value -0.0 to not be zero (+0.0)

Table 3.51: *brz* instruction definition

<i>brz</i>		both	main
Syntax	<i>brz</i> \$mSrc0, <i>zimm19</i>		
Semantics	Prepare	DataWord op0 = \$mSrc0; DataWord op1 = (<i>zimm19</i> << 2); DataWord target = op1;	
	Except In	<pre> if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); } </pre>	
	Commit	<pre> if (op0 == 0) { TARGET_PC = target; } </pre>	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.8 call

Unconditional absolute branch and link. Save the next value of **\$PC** into a general purpose register and perform an unconditional branch. Immediate provides word-addressed absolute destination address.

Note: Bit 19 of the immediate is ignored

Table 3.52: *call* instruction definition

<i>call</i>	both	main
Syntax	call \$mDst0, zimm20	
Semantics	Prepare	<pre>DataWord op1 = (zimm20 << 2); DataWord target = (op1 & TMEM_FULL_ADDRESS_MASK);</pre>
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else if (!TMem_IsValidAddress(target)) { // Target isn't within valid memory range EXCEPT(TEXCPT_INVALID_PC); } else if (!TMem_AddressIsExecutable(target)) { // Target isn't within an executable memory region EXCEPT(TEXCPT_INVALID_PC); }</pre>
	Commit	<pre>// Store return address in op0 TARGET_PC = target; \$mDst0 = COISSUE ? \$PC + 8 : \$PC + 4;</pre>

Architectural state references: *\$REPEAT_COUNT* , *\$PC*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.2.9 exitneg

Terminate current execution of a *Worker* thread and return a Boolean exit status to the *Supervisor* thread. This instruction passes control from a *Worker* thread to the *Supervisor* thread. The currently allocated thread execution slot is returned to the *Supervisor*, which may reassign the execution slot to another task.

Table 3.53: *exitneg* instruction definition

<i>exitneg</i>	worker	main
Syntax	exitneg \$mSrc0	
Semantics	Prepare	SignedDataWord op0 = \$mSrc0;
	Except In	<pre> if (0 != \$REPEAT_COUNT) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } </pre>
	Commit	<pre> setExitValue(op0 < 0); // run mode will transition to INACTIVE at instruction retirement setNextRunState(TRUNM_INACTIVE); </pre>

Architectural state references: *\$REPEAT_COUNT*

3.7.2.10 *exitnz*

Terminate current execution of a *Worker* thread and return a Boolean exit status to the *Supervisor* thread. This instruction passes control from a *Worker* thread to the *Supervisor* thread. The currently allocated thread execution slot is returned to the *Supervisor*, which may reassign the execution slot to another task.

Note: This instruction considers the floating-point *single-precision* value -0.0 to not be zero (+0.0)

Table 3.54: *exitnz* instruction definition

<i>exitnz</i>	worker	main
Syntax	exitnz \$mSrc0	
Semantics	Prepare	DataWord op0 = \$mSrc0;
	Except In	<pre> if (0 != \$REPEAT_COUNT) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } </pre>
	Commit	<pre> setExitValue(op0 != 0); // run mode will transition to INACTIVE at instruction retirement setNextRunState(TRUNM_INACTIVE); </pre>

Architectural state references: *\$REPEAT_COUNT*

3.7.2.11 *exitpos*

Terminate current execution of a *Worker* thread and return a Boolean exit status to the *Supervisor* thread. This instruction passes control from a *Worker* thread to the *Supervisor* thread. The currently allocated thread execution slot is returned to the *Supervisor*, which may reassign the execution slot to another task.

Table 3.55: *exitpos* instruction definition

<i>exitpos</i>		worker	main
Syntax	exitpos \$mSrc0		
Semantics	Prepare	SignedDataWord op0 = \$mSrc0;	
	Except In	<pre>if (0 != \$REPEAT_COUNT) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); }</pre>	
	Commit	<pre>setExitValue(op0 >= 0); // run mode will transition to INACTIVE at instruction retirement setNextRunState(TRUNM_INACTIVE);</pre>	

Architectural state references: *\$REPEAT_COUNT*

3.7.2.12 *exitz*

Terminate current execution of a *Worker* thread and return a Boolean exit status to the *Supervisor* thread. This instruction passes control from a *Worker* thread to the *Supervisor* thread. The currently allocated thread execution slot is returned to the *Supervisor*, which may reassign the execution slot to another task.

Note: This instruction considers the floating-point *single-precision* value -0.0 to not be zero (+0.0)

Table 3.56: *exitz* instruction definition

<i>exitz</i>		worker	main
Syntax	exitz \$mSrc0		
Semantics	Prepare	DataWord op0 = \$mSrc0;	
	Except In	<pre>if (0 != \$REPEAT_COUNT) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); }</pre>	
	Commit	<pre>setExitValue(op0 == 0); // run mode will transition to INACTIVE at instruction retirement setNextRunState(TRUNM_INACTIVE);</pre>	

Architectural state references: *\$REPEAT_COUNT*

3.7.2.13 *rpt*

rpt provides a zero-overhead loop facility, causing the subsequent sequence of *Execution Bundles* (the repeat-body) to be executed repeatedly. The repeat-count can be provided as an immediate or as an unsigned register source value. The size of the repeat-body is expressed in whole *Execution Bundles* and provided by an immediate (with the repeat-body size being (immediate + 1) *Execution Bundles*). Note that it is not possible to execute solo instructions within a repeat-body. A *TEXCPT_INVALID_INSTR* exception will be raised in an attempt to execute a solo instruction.

If the repeat-count is zero initially, *rpt* will act as a branch over the repeat-body. Otherwise, the subsequent repeat-body *Execution Bundles* will be executed repeat-count times.

Any instruction co-issued with *rpt* is executed only once, and is not part of the repeat-body.

Control and *System* instructions cannot be executed within the repeat-body. A *TEXCPT_INVALID_INSTR* exception will be raised in an attempt to execute any such instruction within the body of *rpt*.

Exceptions raised during the execution of the repeat-body will always be treated as *malign*, regardless of the underlying exception type (including Debug exceptions). When such exceptions arise, `$WSR.ERPT` is set to 0b1 to indicate that the event is unrecoverable.

Table 3.57: *rpt* instruction definition

<i>rpt</i>	worker	main
Syntax	<code>rpt \$mSrc0, zimm8</code> <code>rpt zimm12, zimm8</code>	
Semantics	Prepare	<pre>DataWord op0 = <(\$mSrc0, zimm12)>; DataWord op1 = (zimm8 << 3); DataWord nextPC = COISSUE ? \$PC + 8 : \$PC + 4; DataWord pcAfterRptBody = nextPC + (op1 + 8);</pre>
	Except In	<pre>if (0 != \$REPEAT_COUNT) { // rpt cannot appear within the body of another rpt EXCEPT(TEXCPT_INVALID_INSTR); } if (nextPC & 0x7) { // Body must be 8-byte aligned EXCEPT(TEXCPT_INVALID_OP); } if (!TMem_IsValidAddress(nextPC)) { // Body isn't within valid memory range EXCEPT(TEXCPT_INVALID_OP); } if (!TMem_IsValidAddress(pcAfterRptBody)) { // Final target \$PC isn't within valid memory range EXCEPT(TEXCPT_INVALID_OP); } if (!TMem_AddressIsExecutable(nextPC)) { // Body isn't within an executable memory region EXCEPT(TEXCPT_INVALID_OP); } if (!TMem_AddressIsExecutable(pcAfterRptBody)) { // Final target \$PC isn't within an executable memory region EXCEPT(TEXCPT_INVALID_OP); }</pre>
	Commit	<pre>uint32_t count = op0 & CSR_W_REPEAT_COUNT__VALUE__MASK; if (count != 0) { \$REPEAT_COUNT = count; \$REPEAT_FIRST = nextPC; \$REPEAT_END = pcAfterRptBody; } else { // Simply branch over the repeat-body TARGET_PC = pcAfterRptBody; }</pre>

Architectural state references: `$PC`, `$REPEAT_COUNT`, `$REPEAT_FIRST`, `$REPEAT_END`

Function references: `TMem_IsValidAddress`, `TMem_AddressIsExecutable`

rpt occurs in the following code examples:

- *f16v4cmac* example
- *f16v4hihoamp* example
- *f16v4sisoamp* example
- *f16v4sisoslic* example part 1
- *f16v4sisoslic* example part 2
- *f16v4stacc* example

-
- *f32mac example*
 - *f32sisoamp example*

- *f32sisoslic example*
- *ldb16b16 example*

3.7.3 Float

Floating-point instructions summary table

3.7.3.1 Format conversion

3.7.3.1.1 f16tof32

Convert a *half-precision* floating-point value to *single-precision*.

Table 3.58: *f16tof32* instruction definition

<i>f16tof32</i>		worker	aux
Syntax	f16tof32 \$aDst0, \$aSrc0		
Semantics	Prepare	Half op1 = pickHalf(\$aSrc0, 0); Single result;	
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; if (op1.isNaN()) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Compute	if (op1.isNaN()) { result = TFPU_F32_QuietenNan((Single)op1); } else { result = (Single)op1; }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_F32_QuietenNan*, *TFPU_BitsFromF32*

3.7.3.1.2 f16v2sufromui

Symmetric, unbiased conversion from 2-element vector of unsigned 16-bit integers to 2-element *half-precision* vector.

Each of the *half-precision* results lies within the range $[-\frac{1}{2}, \frac{1}{2}]$ but can never be exactly 0. The minimum result magnitude is $\frac{1}{2^{17}}$ (and therefore results can lie within the denormalised number range for *half-precision*).

Note that this instruction can be combined with *urand32/urand64* to produce random, uniformly distributed floating point values.

Table 3.59: *f16v2sufromui* instruction definition

<i>f16v2sufromui</i>		worker	aux
Syntax	f16v2sufromui \$aDst0, \$aSrc0		
Semantics	Prepare	<pre> array<HalfDataWord,2> op1 = { ((\$aSrc0[0] >> 0) & 0xffff), ((\$aSrc0[0] >> 16) & 0xffff) }; bool nanoo = \$FP_CTL.NANOO; array<Single,2> fval; </pre>	
	Compute	<pre> for (i = 0; i < 2; i++) { // Double the magnitude of the input value // and shift the result to produce a symmetric // distribution centred on 0 (resulting range here is [-65535, 65535]) int32_t valp = (2 * op1[i]) - ((1 << 16) - 1); // Scale the result so that the resulting output range is [-0.5, 0.5] fval[i] = valp / exp2(17); } </pre>	
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(fval[0]).bitz32(smode) (Half(fval[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*

3.7.3.1.3 f16v2tof32

Half-precision floating-point pair to *single-precision* conversion

Table 3.60: *f16v2tof32* instruction definition

<i>f16v2tof32</i>		worker	aux
Syntax	f16v2tof32 \$aDst0:Dst0+1, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; if (op1[0].issNaN() op1[1].issNaN()) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>if (op1[0].isNan()) { result[0] = TFPU_F32_QuietenNan((Single)op1[0]); } else { result[0] = (Single)op1[0]; } if (op1[1].isNan()) { result[1] = TFPU_F32_QuietenNan((Single)op1[1]); } else { result[1] = (Single)op1[1]; }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) };</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_F32_QuietenNan*, *TFPU_BitsFromF32*

3.7.3.1.4 f16v4sufromui

Symmetric, unbiased conversion from 4-element vector of unsigned 16-bit integers to 4-element *half-precision* vector.

Each of the *half-precision* results lies within the range $[-\frac{1}{2}, \frac{1}{2}]$ but can never be exactly 0. The minimum result magnitude is $\frac{1}{2^{17}}$ (and therefore results can lie within the denormalised number range for *half-precision*).

Note that this instruction can be combined with *urand32/urand64* to produce random, uniformly distributed floating point values.

Table 3.61: *f16v4sufromui* instruction definition

<i>f16v4sufromui</i>		worker	aux
Syntax	f16v4sufromui \$aDst0:Dst0+1, \$aSrc0:Src0+1		
Semantics	Prepare	<pre>array<HalfDataWord,4> op1 = { ((\$aSrc0:Src0+1[0] >> 0) & 0xffff), ((\$aSrc0:Src0+1[0] >> 16) & 0xffff), ((\$aSrc0:Src0+1[1] >> 0) & 0xffff), ((\$aSrc0:Src0+1[1] >> 16) & 0xffff) }; bool nanoo = \$FP_CTL.NANOO; array<Single,4> fval;</pre>	
	Compute	<pre>for (i = 0; i < 4; i++) { // Double the magnitude of the input value // and shift the result to produce a symmetric // distribution centred on 0 (resulting range here is [-65535, 65535]) int32_t valp = (2 * op1[i]) - ((1 << 16) - 1); // Scale the result so that the resulting output range is [-0.5, 0.5] fval[i] = valp / exp2(17); }</pre>	
	Commit	<pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(fval[0]).bitz32(smode) (Half(fval[1]).bitz32(smode) << 16), Half(fval[2]).bitz32(smode) (Half(fval[3]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL*

3.7.3.1.5 f32fromi32

Convert a signed integer to a *single-precision* floating-point value.

Table 3.62: *f32fromi32* instruction definition

<i>f32fromi32</i>		worker	aux
Syntax	f32fromi32 \$aDst0, \$aSrc0		
Semantics	Prepare	SignedDataWord op1 = \$aSrc0;	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; Double accurate = op1; Single result = TFPU_RoundFP64ToFmt(accurate, TFPU_FP32, rmode);</pre>	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL*

Function references: *TFPU_RoundFP64ToFmt*, *TFPU_BitsFromF32*

3.7.3.1.6 f32fromui32

Convert an unsigned integer to a *single-precision* floating-point value.

Table 3.63: *f32fromui32* instruction definition

<i>f32fromui32</i>	worker	aux
Syntax	<code>f32fromui32 \$aDst0, \$aSrc0</code>	
Semantics	Prepare	<code>DataWord op1 = \$aSrc0;</code>
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; Double accurate = op1; Single result = TFPU_RoundFP64ToFmt(accurate, TFPU_FP32, rmode);</pre>
	Commit	<code>\$aDst0 = TFPU_BitsFromF32(result);</code>

Architectural state references: *\$FP_CTL*

Function references: *TFPU_RoundFP64ToFmt*, *TFPU_BitsFromF32*

f32fromui32 occurs in the following code examples:

- *ldb16b16 example*

3.7.3.1.7 *f32sufromui*

Symmetric, unbiased conversion from an unsigned 32-bit integer to *single-precision* floating-point.

The *single-precision* result lies within the range $[-\frac{1}{2}, \frac{1}{2}]$ but can never be exactly 0. The result will also have a magnitude of at least $\frac{1}{2^{33}}$ (and therefore results will never be inside the denormalised number range for *single-precision*).

Note that this instruction can be combined with *urand32/urand64* to produce a random, uniformly distributed floating point value.

Table 3.64: *f32sufromui* instruction definition

<i>f32sufromui</i>	worker	aux
Syntax	<code>f32sufromui \$aDst0, \$aSrc0</code>	
Semantics	Prepare	<code>DataWord op1 = \$aSrc0;</code>
	Compute	<pre>// Obtain unsigned 32-bit input value uint64_t uival = op1; // Double the magnitude of the input value // and shift the result to produce a symmetric // distribution centred on 0 (resulting range here is [-2^32-1, 2^32-1]) int64_t valp = (2 * uival) - ((1ULL << 32) - 1); // Scale the result so that the resulting output range is [-0.5, 0.5] Double res = valp / exp2(33); // Round to nearest single-precision value, ties to even Single result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);</pre>
	Commit	<code>\$aDst0 = TFPU_BitsFromF32(result);</code>

Function references: *TFPU_RoundFP64ToFmt*, *TFPU_BitsFromF32*

3.7.3.1.8 *f32tof16*

Convert a *single-precision* value to *half-precision*, using the rounding mode as specified by *\$FP_CTL.RND/ESR*. Supports stochastic rounding. See *Format Conversion and Transformations*.

The 16-bit result of the conversion is broadcast to (duplicated into) a single ARF register, producing a 2-element vector of identical *half-precision* values.

Table 3.65: *f32tof16* instruction definition

<i>f32tof16</i>		worker	aux
Syntax	f32tof16 \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>Single op1 = TFPU_F32FromBits(\$aSrc0); bool nanoo = \$FP_CTL.NANOO; vector<Single> result = { op1 };</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; // Floating-point exception check if (\$FP_CTL.ESR) { array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); TFPU_ApplyStochasticRound(randomBits, result); } if (TFPU_F32_IsSNan(op1)) { fpExcpt = TFPEXCPT_INV; } else if (TFPU_F32_IsQNaN(op1)) { // No exception } else if (isinf(op1)) { fpExcpt = TFPEXCPT_INV; } else { fpExcpt = TFPU_GenOFLOCheck(result[0], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>if (TFPU_F32_IsSNan(op1) TFPU_F32_IsQNaN(op1) isinf(op1)) { result[0] = TFPU_F32_QNaN(); } else { result[0] = result[0]; }</pre>	
	Commit	<pre>// Value returned in both halves of the result HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[0]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_ApplyStochasticRound*, *TFPU_F32_IsSNan*, *TFPU_F32_IsQNaN*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_F32_QNaN*

3.7.3.1.9 f32toi32

Convert a *single-precision* floating-point value to a signed integer, rounding as per *\$FP_CTL.RND*.

Table 3.66: *f32toi32* instruction definition

<i>f32toi32</i>		worker	aux
Syntax	f32toi32 \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); SignedDataWord result;	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; TileRoundMode_t mode = \$FP_CTL.RND; Single rounded = TFPU_RoundFP32ToIntegral(op1, mode); if (isnan(op1) isinf(op1)) { // IEEE 754-2008: 5.8 fpExcpt = TFPEXCPT_INV; } else { // IEEE 754-2008: 5.8 // Source operand is not a NaN or infinity. Check for out-of-range. if ((static_cast<int64_t>(rounded) > INT32_MAX) (static_cast<int64_t>(rounded) < INT32_MIN)) { // IEEE 754-2008: 5.8 fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> if (isnan(op1) isinf(op1)) { // IEEE 754-2008: 5.8 (result is implementation defined) result = INT32_MIN; } else if (fabs(op1) == 0.0) { // -0.0 -> 0 result = 0; } else { // Source operand is not a NaN, infinity, or zero. if ((static_cast<int64_t>(rounded) > INT32_MAX) (static_cast<int64_t>(rounded) < INT32_MIN)) { // IEEE 754-2008: 5.8 (result is implementation defined) result = INT32_MIN; } else { // Source operand is not a NaN, infinity or zero and is within // uint32 range. result = static_cast<int32_t>(rounded); } } </pre>	
	Commit	\$aDst0 = result;	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_RoundFP32ToIntegral* , *TFPU_IsMalign*

3.7.3.1.10 f32toi32

Convert a *single-precision* floating-point value to a unsigned integer, rounding as per *\$FP_CTL.RND*.

Table 3.67: *f32toi32* instruction definition

<i>f32toi32</i>		worker	aux
Syntax	f32toi32 \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); DataWord result;	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; TileRoundMode_t mode = \$FP_CTL.RND; Single rounded = TFPU_RoundFP32ToIntegral(op1, mode); if (isnan(op1) isinf(op1)) { // IEEE 754-2008: 5.8 fpExcpt = TFPEXCPT_INV; } else { // IEEE 754-2008: 5.8 // Source operand is not a NaN or infinity. Check for out-of-range. if ((static_cast<int64_t>(rounded) > UINT32_MAX) (static_cast<int64_t>(rounded) < 0)) { // IEEE 754-2008: 5.8 fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> if (isnan(op1) isinf(op1)) { // IEEE 754-2008: 5.8 (result is implementation defined) result = 0; } else if (fabs(op1) == 0.0) { // -0.0 -> 0 result = 0; } else { // Source operand is not a NaN, infinity, or zero. if ((static_cast<int64_t>(rounded) > UINT32_MAX) (static_cast<int64_t>(rounded) < 0)) { // IEEE 754-2008: 5.8 (result is implementation defined) result = 0; } else { // Source operand is not a NaN, infinity or zero and is within // uint32 range. result = static_cast<uint32_t>(rounded); } } </pre>	
	Commit	\$aDst0 = result;	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_RoundFP32ToIntegral* , *TFPU_IsMalign*

3.7.3.1.11 f32v2sufromui

Symmetric, unbiased conversion from 2-element vector of unsigned 32-bit integers to 2-element *single-precision* vector.

Each of the *single-precision* results lies within the range $[-\frac{1}{2}, \frac{1}{2}]$ but can never be exactly 0. All results will have a magnitude of at least $\frac{1}{2^{33}}$ (and therefore results will never be inside the denormalised number range for *single-precision*).

Note that this instruction can be combined with *urand32/urand64* to produce random, uniformly distributed floating point values.

Table 3.68: *f32v2sufromui* instruction definition

<i>f32v2sufromui</i>		worker	aux
Syntax	f32v2sufromui \$aDst0:Dst0+1, \$aSrc0:Src0+1		
Semantics	Prepare	<pre>array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<Single,2> result;</pre>	
	Compute	<pre>for (i = 0; i < 2; i++) { uint64_t uival = op1[i]; // Double the magnitude of the input value // and shift the result to produce a symmetric // distribution centred on 0 (resulting range here is [-2^32-1, 2^32-1]) int64_t valp = (2 * uival) - ((1ULL << 32) - 1); // Scale the result so that the resulting output range is [-0.5, 0.5] Double res = valp / exp2(33); // Round to nearest single-precision value, ties to even result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) };</pre>	

Function references: *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

3.7.3.1.12 f32v2tof16

Single-precision floating-point pair to *half-precision* conversion

Table 3.69: *f32v2tof16* instruction definition

<i>f32v2tof16</i>	worker	aux
Syntax	<i>f32v2tof16</i> <i>\$aDst0</i> , <i>\$aSrc0:Src0+1</i>	
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; bool nanoo = \$FP_CTL.NANOO; vector<Single> result = { op1[0], op1[1] }; </pre>
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; // Floating-point exception check if (\$FP_CTL.ESR) { array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 2; i++) { if (TFPU_F32_IsSNan(op1[i])) { fpExcpt = TFPEXCPT_INV; } else if (TFPU_F32_IsQNaN(op1[i])) { // No exception } else if (isinf(op1[i])) { fpExcpt = TFPEXCPT_INV; } else { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>
	Compute	<pre> for (i = 0; i < 2; i++) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsQNaN(op1[i]) isinf(op1[i])) { result[i] = TFPU_F32_QNaN(); } else { result[i] = result[i]; } } </pre>
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_ApplyStochasticRound*, *TFPU_F32_IsSNan*, *TFPU_F32_IsQNaN*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_F32_QNaN*

3.7.3.1.13 *f32v4tof16*

Single-precision floating-point 4-element vector to *half-precision* 4-element vector conversion.

Table 3.70: *f32v4tof16* instruction definition

<i>f32v4tof16</i>		worker	aux
Syntax	f32v4tof16 \$aDst0:Dst0+1, \$aSrc0:Src0+3		
Semantics	Prepare	<pre> array<Single,4> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+3[0]), TFPU_F32FromBits(\$aSrc0:Src0+3[1]), TFPU_F32FromBits(\$aSrc0:Src0+3[2]), TFPU_F32FromBits(\$aSrc0:Src0+3[3]) }; bool nanoo = \$FP_CTL.NANOO; vector<Single> result = { op1[0], op1[1], op1[2], op1[3] }; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; // Floating-point exception check if (\$FP_CTL.ESR) { array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 4; i++) { if (TFPU_F32_IsSNan(op1[i])) { fpExcpt = TFPEXCPT_INV; } else if (TFPU_F32_IsQNaN(op1[i])) { // No exception } else if (isinf(op1[i])) { fpExcpt = TFPEXCPT_INV; } else { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> for (i = 0; i < 4; i++) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsQNaN(op1[i]) isinf(op1[i])) { result[i] = TFPU_F32_QNaN(); } else { result[i] = result[i]; } } </pre>	
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_ApplyStochasticRound*, *TFPU_F32_IsSNan*, *TFPU_F32_IsQNaN*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_F32_QNaN*

3.7.3.2 f16 2-element vector

3.7.3.2.1 f16v2absadd

Half-precision 2-element vector element-wise addition of absolute values.

Table 3.71: *f16v2absadd* instruction definition

<i>f16v2absadd</i>		worker	aux
Syntax	f16v2absadd \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,2> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; } </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(2); for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { op1[i] = fabs(op1[i]); op2[i] = fabs(op2[i]); double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } } </pre>	
	Except Out	<pre> if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 2; ++i) { fpExcpt = TFPU_GenOFLCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAddPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_ApplyStochasticRound* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign*

3.7.3.2.2 f16v2absmax

Half-precision floating point vector max of absolute values

Table 3.72: *f16v2absmax* instruction definition

<i>f16v2absmax</i>		worker	aux
Syntax	f16v2absmax \$aDst0, \$aSrc1, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<Half,2> z;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>z[0] = TFPU_Max(fabs(op1[0]), fabs(op2[0])); z[1] = TFPU_Max(fabs(op1[1]), fabs(op2[1]));</pre>	
	Commit	<pre>\$aDst0 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_IsMalign* , *TFPU_Max*

3.7.3.2.3 f16v2add

Half-precision floating point vector add on two register source values.

Table 3.73: *f16v2add* instruction definition

<i>f16v2add</i>		worker	aux
Syntax	f16v2add \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2add \$aDst0, \$aSrc0, \$aSrc1 f16v2add \$aDst0, \$aSrc0:BU, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,2> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; } </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(2); for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } } </pre>	
	Except Out	<pre> if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 2; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: [TFPU_DoAddPreExecute](#) , [TFPU_RoundFP64ToFmt](#) , [TFPU_ApplyStochasticRound](#) , [TFPU_GenOFLOCheck](#) , [TFPU_IsMalign](#)

3.7.3.2.4 f16v2clamp

Half-precision floating-point vector min-of-maximum

Table 3.74: *f16v2clamp* instruction definition

<i>f16v2clamp</i>	worker	aux
Syntax	f16v2clamp \$aDst0, \$aSrc0, \$aSrc1	
Semantics Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<Half,2> result;</pre>	
Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
Compute	<pre>Half lower = op2[0]; Half upper = op2[1]; // Unlike min/max, clamp explicitly propagates (regenerates) // NaN inputs if (lower.isNaN() upper.isNaN()) { for (i = 0; i < 2; i++) { result[i] = TFPU_F32_QNan(); } } else { for (i = 0; i < 2; i++) { if (op1[i].isNaN()) { result[i] = TFPU_F32_QNan(); } else if (op1[i] > upper) { result[i] = upper; } else { result[i] = TFPU_Max(op1[i], lower); } } }</pre>	
Commit	<pre>\$aDst0 = { Half(result[0]).bitz32(PROP_INF) Half(result[1]).bitz32(PROP_INF) << 16 };</pre>	

Architectural state references: [\\$FP_STS](#) , [\\$FP_CTL](#)

Function references: [TFPU_IsMalign](#) , [TFPU_F32_QNan](#) , [TFPU_Max](#)

3.7.3.2.5 f16v2class

Half-precision floating-point vector classifier. IEEE 754-2008: 5.7.2

Table 3.75: *f16v2class* instruction definition

<i>f16v2class</i>		worker	aux
Syntax	f16v2class \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<QuarterDataWord,4> op0; array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) };</pre>	
	Compute	<pre>op0 = { 0, 0, 0, 0 }; for (i = 0; i < 2; i++) { DataWord clss; bool sign = op1[i].sign(); if (op1[i].issNaN()) { clss = TFPU_CLASS_SNAN; } else if (op1[i].isqNaN()) { clss = TFPU_CLASS_QNAN; } else if (op1[i].isInf()) { clss = (sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF); } else if (op1[i].isZero()) { clss = (sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO); } else if (op1[i].isDenorm()) { clss = (sign ? TFPU_CLASS_NEG_DENORM : TFPU_CLASS_POS_DENORM); } else { clss = (sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM); } op0[i] = clss; }</pre>	
	Commit	<pre>\$aDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

3.7.3.2.6 f16v2cmac

Half-precision floating-point vector element-wise multiply with *single-precision* lateral sum and accumulate.

Table 3.76: *f16v2cmac* instruction definition

<i>f16v2cmac</i>		worker	aux
Syntax	<code>f16v2cmac \$aSrc0, \$aSrc1</code>		
Semantics	<p>Prepare</p> <pre>array<Half,2> op0 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op1 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; Single result;</pre> <p>Except In</p> <pre>bool specialCase = false; uint32_t fpExcpt = TFPEXCPT_NONE; if (op0[0].isNaN() op1[0].isNaN() op0[1].isNaN() op1[1].isNaN()) { specialCase = true; result = TFPU_F32_QNan(); fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute if (!specialCase) { vector<Single> xv = { op0[0], op0[1] }; vector<Single> yv = { op1[0], op1[1] }; result = TFPU_F16DotProduct(xv, yv); result = TFPU_Add(\$AACC[0], result, TFPU_FP32); } Commit \$AACC[0] = result;</pre>		

Architectural state references: *\$FP_STS*, *\$FP_CTL*, *\$AACC*

Function references: *TFPU_F32_QNan*, *TFPU_IsMalign*, *TFPU_F16DotProduct*, *TFPU_Add*

f16v2cmac occurs in the following code examples:

- *ldb16b16 example*

3.7.3.2.7 *f16v2cmpeq*

Half-precision 2-element vector equality test

Table 3.77: *f16v2cmpeq* instruction definition

<i>f16v2cmpeq</i>		worker	aux
Syntax	f16v2cmpeq \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmpeq \$aDst0, \$aSrc0, \$aSrc1 f16v2cmpeq \$aDst0, \$aSrc0:BU, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } </pre>	
	Except In	<pre> // IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> result[0] = (r1[0] == TFPU_RELATION_EQ); result[1] = (r1[1] == TFPU_RELATION_EQ); </pre>	
	Commit	<pre> \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) }; </pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.2.8 f16v2cmpge

Half-precision floating point vector greater-than-or-equal-to test

Table 3.78: *f16v2cmpge* instruction definition

<i>f16v2cmpge</i>	worker	aux
Syntax	f16v2cmpge \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmpge \$aDst0, \$aSrc0, \$aSrc1 f16v2cmpge \$aDst0, \$aSrc0:BU, \$aSrc1	
Semantics	<pre> Prepare array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = ((r1[0] == TFPU_RELATION_EQ) (r1[0] == TFPU_RELATION_GT)); result[1] = ((r1[1] == TFPU_RELATION_EQ) (r1[1] == TFPU_RELATION_GT)); Commit \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.2.9 f16v2cmpgt

Half-precision floating point vector greater-than test

Table 3.79: *f16v2cmpgt* instruction definition

<i>f16v2cmpgt</i>	worker	aux
Syntax	f16v2cmpgt \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmpgt \$aDst0, \$aSrc0, \$aSrc1 f16v2cmpgt \$aDst0, \$aSrc0:BU, \$aSrc1	
Semantics	<pre> Prepare array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] == TFPU_RELATION_GT); result[1] = (r1[1] == TFPU_RELATION_GT); Commit \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.2.10 *f16v2cmple*

Half-precision floating point vector less-than test

Table 3.80: *f16v2cmple* instruction definition

<i>f16v2cmple</i>	worker	aux
Syntax	<pre>f16v2cmple \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmple \$aDst0, \$aSrc0, \$aSrc1 f16v2cmple \$aDst0, \$aSrc0:BU, \$aSrc1</pre>	
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except In // IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = ((r1[0] == TFPU_RELATION_EQ) (r1[0] == TFPU_RELATION_LT)); result[1] = ((r1[1] == TFPU_RELATION_EQ) (r1[1] == TFPU_RELATION_LT)); Commit \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.2.11 *f16v2cmplt*

Half-precision floating point vector less-than test

Table 3.81: *f16v2cmplt* instruction definition

<i>f16v2cmplt</i>	worker	aux
Syntax	<pre>f16v2cmplt \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmplt \$aDst0, \$aSrc0, \$aSrc1 f16v2cmplt \$aDst0, \$aSrc0:BU, \$aSrc1</pre>	
Semantics	<pre>Prepare array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] == TFPU_RELATION_LT); result[1] = (r1[1] == TFPU_RELATION_LT); Commit \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.2.12 *f16v2cmpne*

Half-precision floating point vector equality test

Table 3.82: *f16v2cmpne* instruction definition

<i>f16v2cmpne</i>		worker	aux
Syntax	f16v2cmpne \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2cmpne \$aDst0, \$aSrc0, \$aSrc1 f16v2cmpne \$aDst0, \$aSrc0:BU, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } </pre>	
	Except In	<pre> // IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> result[0] = (r1[0] != TFPU_RELATION_EQ); result[1] = (r1[1] != TFPU_RELATION_EQ); </pre>	
	Commit	<pre> \$aDst0 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.2.13 f16v2exp

Table 3.83: *f16v2exp* instruction definition

<i>f16v2exp</i>		worker	aux
Syntax	f16v2exp \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoExpPreExecute(op1[i], &fpExcpt, &result[i]); }</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { result[i] = TFPU_F16Exp(op1[i], TFPU_BASE_E, rmode); fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } }</pre>	
	Except Out	<pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	<pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_DoExpPreExecute* , *TFPU_F16Exp* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign*

3.7.3.2.14 f16v2exp2

Table 3.84: *f16v2exp2* instruction definition

<i>f16v2exp2</i>		worker	aux
Syntax	f16v2exp2 \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoExpPreExecute(op1[i], &fpExcpt, &result[i]); }</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { result[i] = TFPU_F16Exp(op1[i], TFPU_BASE_2, rmode); fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } }</pre>	
	Except Out	<pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	<pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_DoExpPreExecute* , *TFPU_F16Exp* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign*

3.7.3.2.15 f16v2gina

Get and initialise accumulators.

- Read a pair of internal accumulator values as *half-precision* values. Stochastic rounding applies as configured by *\$FP_CTL.ESR*.
- Convert 2-element vector of *half-precision* input values to single precision and write to internal accumulator state.
- The instruction immediate specifies which pair of accumulator registers are to be read and written:
 1. Read *\$AACC[0]* and *\$AACC[2]*, write *\$AACC[12]* and *\$AACC[14]*
 2. Read *\$AACC[1]* and *\$AACC[3]*, write *\$AACC[13]* and *\$AACC[15]*
and if and only if the platform supports 2 AMP sets:
 3. Read *\$AACC[16]* and *\$AACC[18]*, write *\$AACC[28]* and *\$AACC[30]*
 4. Read *\$AACC[17]* and *\$AACC[19]*, write *\$AACC[29]* and *\$AACC[31]*
- Propagate internal accumulator state such that all accumulator registers may be read and written via a sequence of this instruction.

zimm12 immediate format:

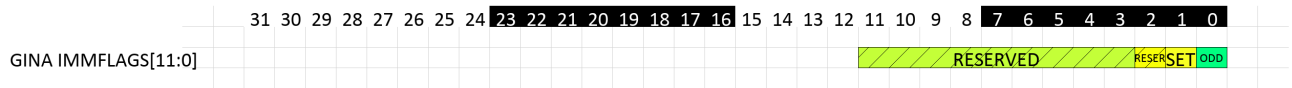


Fig. 3.11: *f16v2gina* immediate format

Table 3.85: *f16v2gina* instruction definition

<i>f16v2gina</i>	worker	aux
Syntax	f16v2gina \$aDst0, \$aSrc0, zimm12	
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; DataWord op2 = zimm12; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; vector<Single> result; unsigned set = GINA_IMMFLAGS__SET__GET(op2); // base accumulator Id for set unsigned b = set * TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; </pre>
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; // Input exception check if (op1[0].isNaN() op1[1].isNaN()) { fpExcpt = TFPEXCPT_INV; } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
	Compute	<pre> if (GINA_IMMFLAGS__ODD__GET(op2) == 0) { result = { \$AACC[b+0], \$AACC[b+2] }; } else { result = { \$AACC[b+1], \$AACC[b+3] }; } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, result); } </pre>
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(result, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>
	Commit	<pre> Single x0 = op1[0]; Single x1 = op1[1]; if (GINA_IMMFLAGS__ODD__GET(op2) == 0) { // Propagate internal accumulator state \$AACC[b+0] = \$AACC[b+4]; \$AACC[b+4] = \$AACC[b+8]; \$AACC[b+8] = \$AACC[b+12]; \$AACC[b+2] = \$AACC[b+6]; \$AACC[b+6] = \$AACC[b+10]; \$AACC[b+10] = \$AACC[b+14]; // Commit 2 input values \$AACC[b+12] = isnan(x0) ? TFPU_F32_QuietenNan(x0) : x0; \$AACC[b+14] = isnan(x1) ? TFPU_F32_QuietenNan(x1) : x1; } else { // Propagate internal accumulator state \$AACC[b+1] = \$AACC[b+5]; \$AACC[b+5] = \$AACC[b+9]; \$AACC[b+9] = \$AACC[b+13]; \$AACC[b+3] = \$AACC[b+7]; \$AACC[b+7] = \$AACC[b+11]; \$AACC[b+11] = \$AACC[b+15]; } </pre>

Table 3.85: *f16v2gina* instruction definition (continued)

<i>f16v2gina</i>	worker	aux
Syntax	f16v2gina \$aDst0, \$aSrc0, zimm12	
Semantics	Commit cont'd	<pre> // Commit 2 input values \$AACC[b+13] = isnan(x0) ? TFPU_F32_QuietenNan(x0) : x0; \$AACC[b+15] = isnan(x1) ? TFPU_F32_QuietenNan(x1) : x1; } for (i = 0; i < 2; i++) { if (isinf(result[i]) isnan(result[i])) { result[i] = (Half)TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>

Architectural state references: \$FP_CTL, \$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, \$AACC, \$FP_STS

Function references: TFPU_ApplyStochasticRound, TFPU_AACCReadFlags, TFPU_IsMalign, TFPU_F32_QuietenNan, TFPU_F32_QNan

3.7.3.2.16 f16v2grand

Gaussian distribution, 2-element half-precision random vector

Table 3.86: *f16v2grand* instruction definition

<i>f16v2grand</i>	worker	aux
Syntax	f16v2grand \$aDst0	
Semantics	Prepare	<pre> bool nanoo = \$FP_CTL.NANOO; array<Single,2> result; </pre>
	Compute	<pre> array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); // Calculate overall summation based on outputs from successive applications // of xoroshiro128aox int16_t rsum[2] = { 0, 0 }; for (i = 0; i < 12; i++) { rsum[0] += ((randomBits[0] >> (i * 5)) & 0x1f); rsum[1] += ((randomBits[1] >> (i * 5)) & 0x1f); } result[0] = (rsum[0] - 186) / 32.0; result[1] = (rsum[1] - 186) / 32.0; </pre>
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>

Architectural state references: \$FP_CTL, \$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1

3.7.3.2.17 f16v2ln

Table 3.87: *f16v2ln* instruction definition

<i>f16v2ln</i>	worker	aux
Syntax	f16v2ln \$aDst0, \$aSrc0	
Semantics	<p>Prepare</p> <pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result;</pre> <p>Except In</p> <pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoLogPreExecute(op1[i], &fpExcpt, &result[i]); }</pre> <p>Compute</p> <pre>TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { result[i] = TFPU_F16Log(op1[i], TFPU_BASE_E, rmode); } }</pre> <p>Except Out</p> <pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre> <p>Commit</p> <pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_DoLogPreExecute*, *TFPU_F16Log*, *TFPU_IsMalign*

3.7.3.2.18 f16v2log2

Table 3.88: *f16v2log2* instruction definition

<i>f16v2log2</i>		worker	aux
Syntax	f16v2log2 \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoLogPreExecute(op1[i], &fpExcpt, &result[i]); }</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { result[i] = TFPU_F16Log(op1[i], TFPU_BASE_2, rmode); } }</pre>	
	Except Out	<pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	<pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_DoLogPreExecute*, *TFPU_F16Log*, *TFPU_IsMalign*

3.7.3.2.19 f16v2max

Half-precision floating point vector max

Table 3.89: *f16v2max* instruction definition

<i>f16v2max</i>		worker	aux
Syntax	f16v2max \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<Half,2> z;</pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre> z[0] = TFPU_Max(op1[0], op2[0]); z[1] = TFPU_Max(op1[1], op2[1]);</pre>	
	Commit	<pre> \$aDst0 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16) };</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_Max*

3.7.3.2.20 f16v2maxc

Half-precision 2-element vector lateral maximum.

Table 3.90: *f16v2maxc* instruction definition

<i>f16v2maxc</i>		worker	aux
Syntax	<code>f16v2maxc \$aDst0, \$aSrc0</code>		
Semantics	<p>Prepare</p> <pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) };</pre> <p>Except</p> <p>In <pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (op1[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre> <p>Compute</p> <pre>Half max = TFPU_Max(op1[0], op1[1]);</pre> <p>Commit</p> <pre>\$aDst0 = Half(max).bitz32(PROP_INF);</pre> </p>		

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_IsMalign* , *TFPU_Max*

3.7.3.2.21 *f16v2min*

Half-precision floating point vector element-wise minimum

Table 3.91: *f16v2min* instruction definition

<i>f16v2min</i>		worker	aux
Syntax	f16v2min \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<Half,2> z;</pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre> z[0] = TFPU_Min(op1[0], op2[0]); z[1] = TFPU_Min(op1[1], op2[1]);</pre>	
	Commit	<pre> \$aDst0 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16) };</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_Min*

3.7.3.2.22 f16v2mul

Half-precision 2-element vector, Hadamard product

Table 3.92: *f16v2mul* instruction definition

<i>f16v2mul</i>	worker	aux
Syntax	f16v2mul <i>\$aDst0</i> , <i>\$aSrc0:BL</i> , <i>\$aSrc1</i> f16v2mul <i>\$aDst0</i> , <i>\$aSrc0</i> , <i>\$aSrc1</i> f16v2mul <i>\$aDst0</i> , <i>\$aSrc0:BU</i> , <i>\$aSrc1</i>	
Semantics	Prepare <pre> array<Half,2> op1 = { pickHalf(<(<\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(<\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; TileRoundMode_t rmode; vector<Single> result; array<bool,2> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; rmode = TFPU_ROUND_EVEN; } else { enableStochasticRounding = false; rmode = \$FP_CTL.RND; } Except In uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(2); for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoMulPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } Compute for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) * static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, rmode); } } Except Out if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 2; ++i) { fpExcpt = TFPU_GenOFLCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: [TFPU_DoMulPreExecute](#) , [TFPU_RoundFP64ToFmt](#) , [TFPU_ApplyStochasticRound](#) , [TFPU_GenOFLOCheck](#) , [TFPU_IsMalign](#)

3.7.3.2.23 f16v2sigm

Table 3.93: *f16v2sigm* instruction definition

<i>f16v2sigm</i>		worker	aux
Syntax	f16v2sigm \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoSigmoidPreExecute(op1[i], &fpExcpt, &result[i]); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { result[i] = specialCase[i] ? result[i] : TFPU_F16Sigmoid(op1[i], rmode); }</pre>	
	Commit	<pre>HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) };</pre>	

Architectural state references: [\\$FP_CTL](#) , [\\$FP_STS](#)

Function references: [TFPU_DoSigmoidPreExecute](#) , [TFPU_IsMalign](#) , [TFPU_F16Sigmoid](#)

3.7.3.2.24 f16v2sub

Half-precision floating-point vector subtract

Table 3.94: *f16v2sub* instruction definition

<i>f16v2sub</i>	worker	aux
Syntax	f16v2sub \$aDst0, \$aSrc0:BL, \$aSrc1 f16v2sub \$aDst0, \$aSrc0, \$aSrc1 f16v2sub \$aDst0, \$aSrc0:BU, \$aSrc1	
Semantics	<pre> Prepare array<Half,2> op1 = { pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0, \$aSrc0:BL, \$aSrc0:BU)>[0], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,2> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; } Except In uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(2); for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } Compute for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) - static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } } Except Out if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 2; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAddPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_ApplyStochasticRound* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign*

3.7.3.2.25 f16v2sum

Half-precision 2-element vector lateral summation to *single-precision*.

Table 3.95: *f16v2sum* instruction definition

<i>f16v2sum</i>		worker	aux
Syntax	f16v2sum \$aDst0, \$aSrc0		
Semantics	Prepare	<pre>array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; Single result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoAddPreExecute(op1[0], op1[1], &fpExcpt, &result); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>if (!specialCase) { Double res = op1[0] + op1[1]; result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); }</pre>	
	Commit	<pre>\$aDst0 = TFPU_BitsFromF32(result);</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_DoAddPreExecute* , *TFPU_IsMalign* , *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

3.7.3.2.26 f16v2tanh

Table 3.96: *f16v2tanh* instruction definition

<i>f16v2tanh</i>		worker	aux
Syntax	f16v2tanh <i>\$aDst0</i> , <i>\$aSrc0</i>		
Semantics	Prepare	<pre> array<Half,2> op1 = { pickHalf(\$aSrc0[0], 0), pickHalf(\$aSrc0[0], 1) }; bool nanoo = \$FP_CTL.NANOO; array<bool,2> specialCase; array<Single,2> result; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoTanhPreExecute(op1[i], &fpExcpt, &result[i]); } </pre>	
	Compute	<pre> TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { result[i] = TFPU_F16Tanh(op1[i], rmode); } } </pre>	
	Except Out	<pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_DoTanhPreExecute*, *TFPU_F16Tanh*, *TFPU_IsMalign*

3.7.3.3 f16 4-element vector

3.7.3.3.1 f16v4absacc

Half-precision 4-element vector accumulation of absolute values to *single-precision*.

Table 3.97: *f16v4absacc* instruction definition

<i>f16v4absacc</i>		worker	aux
Syntax	f16v4absacc \$aSrc0:Src0+1		
Semantics	Prepare	<pre>array<Half,4> op0 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,4> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op0[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>result[0] = TFPU_Add(op0[0], \$AACC[0], TFPU_FP32, TFPU_ABS); result[1] = TFPU_Add(op0[1], \$AACC[2], TFPU_FP32, TFPU_ABS); result[2] = TFPU_Add(op0[2], \$AACC[4], TFPU_FP32, TFPU_ABS); result[3] = TFPU_Add(op0[3], \$AACC[6], TFPU_FP32, TFPU_ABS);</pre>	
	Commit	<pre>\$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3];</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*, *\$AACC*

Function references: *TFPU_IsMalign*, *TFPU_Add*

3.7.3.3.2 f16v4absadd

Half-precision 4-element vector element-wise addition of absolute values.

Table 3.98: *f16v4absadd* instruction definition

<i>f16v4absadd</i>		worker	aux
Syntax	<code>f16v4absadd \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>		
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,4> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; } </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(4); for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	
	Compute	<pre> for (i = 0; i < 4; ++i) { if (!specialCase[i]) { op1[i] = fabs(op1[i]); op2[i] = fabs(op2[i]); double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } } </pre>	
	Except Out	<pre> if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 4; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	

Continued on next page

Table 3.98: *f16v4absadd* instruction definition (continued)

<i>f16v4absadd</i>		worker	aux
Syntax	f16v4absadd \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1		
Semantics	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_ApplyStochasticRound*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*

3.7.3.3.3 f16v4absmax

Half-precision 4-element vector element-wise max of absolute values

Table 3.99: *f16v4absmax* instruction definition

<i>f16v4absmax</i>		worker	aux
Syntax	f16v4absmax \$aDst0:Dst0+1, \$aSrc1:Src1+1, \$aSrc0:Src0+1		
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<Half,4> z; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> z[0] = TFPU_Max(fabs(op1[0]), fabs(op2[0])); z[1] = TFPU_Max(fabs(op1[1]), fabs(op2[1])); z[2] = TFPU_Max(fabs(op1[2]), fabs(op2[2])); z[3] = TFPU_Max(fabs(op1[3]), fabs(op2[3])); </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16), Half(z[2]).bitz32(PROP_INF) (Half(z[3]).bitz32(PROP_INF) << 16) }; </pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_IsMalign* , *TFPU_Max*

3.7.3.3.4 f16v4acc

Half-precision 4-element vector accumulation to *single-precision*.

Table 3.100: *f16v4acc* instruction definition

<i>f16v4acc</i>	worker	aux
Syntax	f16v4acc \$aSrc0:Src0+1	
Semantics	Prepare	<pre>array<Half,4> op0 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,4> result;</pre>
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op0[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>
	Compute	<pre>result[0] = TFPU_Add(op0[0], \$AACC[0], TFPU_FP32); result[1] = TFPU_Add(op0[1], \$AACC[2], TFPU_FP32); result[2] = TFPU_Add(op0[2], \$AACC[4], TFPU_FP32); result[3] = TFPU_Add(op0[3], \$AACC[6], TFPU_FP32);</pre>
	Commit	<pre>\$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3];</pre>

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_IsMalign* , *TFPU_Add*

3.7.3.3.5 f16v4add

Half-precision 4-element vector element-wise addition.

Table 3.101: *f16v4add* instruction definition

<i>f16v4add</i>		worker	aux
Syntax	<pre>f16v4add \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4add \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4add \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>		
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,4> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; }</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(4); for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); }</pre>	
	Compute	<pre>for (i = 0; i < 4; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } }</pre>	
	Except Out	<pre>if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 4; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	

Continued on next page

Table 3.101: *f16v4add* instruction definition (continued)

<i>f16v4add</i>		worker	aux
Syntax	f16v4add \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4add \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4add \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1		
Semantics	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_ApplyStochasticRound*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*

3.7.3.3.6 f16v4clamp

Half-precision floating-point vector min-of-maximum

Table 3.102: *f16v4clamp* instruction definition

<i>f16v4clamp</i>	worker	aux
Syntax	f16v4clamp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; array<Half,4> result; Except In Single ops[6] = { op1[0], op1[1], op1[2], op1[3], op2[0], op2[1] }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 6); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute Half lower = op2[0]; Half upper = op2[1]; // Unlike min/max, clamp explicitly propagates (regenerates) // NaN inputs if (lower.isNaN() upper.isNaN()) { for (i = 0; i < 4; i++) { result[i] = TFPU_F32_QNan(); } } else { for (i = 0; i < 4; i++) { if (op1[i].isNaN()) { result[i] = TFPU_F32_QNan(); } else if (op1[i] > upper) { result[i] = upper; } else { result[i] = TFPU_Max(op1[i], lower); } } } Commit \$aDst0:Dst0+1 = { Half(result[0]).bitz32(PROP_INF) (Half(result[1]).bitz32(PROP_INF) << 16), Half(result[2]).bitz32(PROP_INF) (Half(result[3]).bitz32(PROP_INF) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_F32_QNan*, *TFPU_Max*

3.7.3.3.7 f16v4class

Half-precision floating-point vector classifier. IEEE 754-2008: 5.7.2

Table 3.103: *f16v4class* instruction definition

<i>f16v4class</i>	worker	aux
Syntax	f16v4class \$aDst0, \$aSrc0:Src0+1	
Semantics	<p>Prepare</p> <pre>array<QuarterDataWord,4> op0; array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) };</pre> <p>Compute</p> <pre>for (i = 0; i < 4; i++) { DataWord clss; bool sign = op1[i].sign(); if (op1[i].issNaN()) { clss = TFPU_CLASS_SNAN; } else if (op1[i].isqNaN()) { clss = TFPU_CLASS_QNAN; } else if (op1[i].isInf()) { clss = (sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF); } else if (op1[i].isZero()) { clss = (sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO); } else if (op1[i].isDenorm()) { clss = (sign ? TFPU_CLASS_NEG_DENORM : TFPU_CLASS_POS_DENORM); } else { clss = (sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM); } op0[i] = clss; } Commit \$aDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

3.7.3.3.8 f16v4cmac

Half-precision floating-point vector element-wise multiply with *single-precision* 2x2 lateral sum and accumulate.

Table 3.104: *f16v4cmac* instruction definition

<i>f16v4cmac</i>	worker	aux
Syntax	f16v4cmac \$aSrc0:Src0+1, \$aSrc1:Src1+1	
Semantics	Prepare	<pre> array<Half,4> op0 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op1 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<Single,2> result; </pre>
	Except In	<pre> array<bool,2> specialCase = { false, false }; uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; i+=2) { if (op0[i].isNaN() op1[i].isNaN() op0[i+1].isNaN() op1[i+1].isNaN()) { specialCase[i/2] = true; result[i/2] = TFPU_F32_QNan(); fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>
	Compute	<pre> TileRoundMode_t rmode = \$FP_CTL.RND; for (int i = 0; i < 4; i+=2) { if (!specialCase[i/2]) { vector<Single> xv = { op0[i], op0[i+1] }; vector<Single> yv = { op1[i], op1[i+1] }; result[i/2] = TFPU_F16DotProduct(xv, yv); result[i/2] = TFPU_Add(\$AACC[i], result[i/2], TFPU_FP32); } } </pre>
	Commit	<pre> \$AACC[0] = result[0]; \$AACC[2] = result[1]; </pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*, *\$AACC*

Function references: *TFPU_F32_QNan*, *TFPU_IsMalign*, *TFPU_F16DotProduct*, *TFPU_Add*

f16v4cmac occurs in the following code examples:

- *f16v4cmac* example

Listing 3.4: *f16v4cmac* example

```

// Pack the addresses into the register pair
tapack      $striPtr, $sinPtr, $weightPtr, $mzero

ld2x64pace $inputs, $weights, $striPtr+8, $mzero, 0

.align 8
{
    rpt      $numRepeats, ((_loop_end - _loop_start) / 8) - 1
    fnop
}

```

```
_loop_start:
{
  // Performance is 4 half-precision fmacs (8 flops) per tick, on average
  ld2x64pace  $inputs, $weights, $striPtr+=, $mzero, 0
  f16v4cmac   $inputs, $weights
}
_loop_end:

// Final fmac
f16v4cmac    $inputs, $weights

// Final reduction - read out the pair of accumulator values and sum
f32v2gina    $a0:1, $zeros, 0
f32add       $a0, $a0, $a1
```

3.7.3.3.9 f16v4cmpeq

Half-precision 4-element vector equality test

Table 3.105: *f16v4cmpeq* instruction definition

<i>f16v4cmpeq</i>	worker	aux
Syntax	<pre>f16v4cmpeq \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmpeq \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmpeq \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] == TFPU_RELATION_EQ); result[1] = (r1[1] == TFPU_RELATION_EQ); result[2] = (r1[2] == TFPU_RELATION_EQ); result[3] = (r1[3] == TFPU_RELATION_EQ); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.10 *f16v4cmpge*

Half-precision 4-element vector greater-than or equal-to test

Table 3.106: *f16v4cmpge* instruction definition

<i>f16v4cmpge</i>	worker	aux
Syntax	<pre>f16v4cmpge \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmpge \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmpge \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = ((r1[0] == TFPU_RELATION_EQ) (r1[0] == TFPU_RELATION_GT)); result[1] = ((r1[1] == TFPU_RELATION_EQ) (r1[1] == TFPU_RELATION_GT)); result[2] = ((r1[2] == TFPU_RELATION_EQ) (r1[2] == TFPU_RELATION_GT)); result[3] = ((r1[3] == TFPU_RELATION_EQ) (r1[3] == TFPU_RELATION_GT)); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.11 *f16v4cmpgt*

Half-precision 4-element vector greater-than test

Table 3.107: *f16v4cmpgt* instruction definition

<i>f16v4cmpgt</i>	worker	aux
Syntax	<pre>f16v4cmpgt \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmpgt \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmpgt \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] == TFPU_RELATION_GT); result[1] = (r1[1] == TFPU_RELATION_GT); result[2] = (r1[2] == TFPU_RELATION_GT); result[3] = (r1[3] == TFPU_RELATION_GT); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.12 *f16v4cmple*

Half-precision 4-element vector less-than or equal-to test

Table 3.108: *f16v4cmple* instruction definition

<i>f16v4cmple</i>	worker	aux
Syntax	<pre>f16v4cmple \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmple \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmple \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = ((r1[0] == TFPU_RELATION_EQ) (r1[0] == TFPU_RELATION_LT)); result[1] = ((r1[1] == TFPU_RELATION_EQ) (r1[1] == TFPU_RELATION_LT)); result[2] = ((r1[2] == TFPU_RELATION_EQ) (r1[2] == TFPU_RELATION_LT)); result[3] = ((r1[3] == TFPU_RELATION_EQ) (r1[3] == TFPU_RELATION_LT)); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.13 *f16v4cmplt*

Half-precision 4-element vector less-than test

Table 3.109: *f16v4cmplt* instruction definition

<i>f16v4cmplt</i>	worker	aux
Syntax	<pre>f16v4cmplt \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmplt \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmplt \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] == TFPU_RELATION_LT); result[1] = (r1[1] == TFPU_RELATION_LT); result[2] = (r1[2] == TFPU_RELATION_LT); result[3] = (r1[3] == TFPU_RELATION_LT); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.14 *f16v4cmpne*

Half-precision 4-element vector inequality test

Table 3.110: *f16v4cmpne* instruction definition

<i>f16v4cmpne</i>	worker	aux
Syntax	<pre>f16v4cmpne \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4cmpne \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4cmpne \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<TileFPRelation_t,4> r1; array<bool,4> result; for (i = 0; i < 4; ++i) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute result[0] = (r1[0] != TFPU_RELATION_EQ); result[1] = (r1[1] != TFPU_RELATION_EQ); result[2] = (r1[2] != TFPU_RELATION_EQ); result[3] = (r1[3] != TFPU_RELATION_EQ); Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffff : 0x0000) ((result[1] ? 0xffff : 0x0000) << 16), (result[2] ? 0xffff : 0x0000) ((result[3] ? 0xffff : 0x0000) << 16) };</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.3.15 *f16v4gacc*

Get accumulators.

Read 4 internal *accumulator* values as a *half-precision* vector.

Table 3.111: *f16v4gacc* instruction definition

<i>f16v4gacc</i>		worker	aux
Syntax	<code>f16v4gacc</code>	<code>\$aDst0:Dst0+1</code>	
Semantics	Prepare	<pre>bool nanoo = \$FP_CTL.NANOO; vector<Single> result = { \$AACC[0], \$AACC[2], \$AACC[4], \$AACC[6] };</pre>	
	Compute	<pre>// Output overflow check if (\$FP_CTL.ESR) { array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); TFPU_ApplyStochasticRound(randomBits, result); }</pre>	
	Except Out	<pre>// Floating-point result exception check uint32_t fpExcpt = TFPU_AACCReadFlags(result, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	<pre>for (i = 0; i < 4; i++) { if (isinf(result[i]) isnan(result[i])) { result[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) };</pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_ApplyStochasticRound*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QNan*

3.7.3.3.16 *f16v4hihoamp*

Half-precision floating point vector accumulating matrix-vector product. Input partial-sum and result values are *half-precision*.

Table 3.112: f16v4hihoamp 8x1x1x16 example sequence

$\$aSrc0$ I	$\$AACC[14]$	$\$AACC[12]$	$\$AACC[10]$	$\$AACC[8]$	$\$AACC[6]$	$\$AACC[4]$	$\$AACC[2]$	$\$AACC[0]$	$\$aDst0$
0^2 P_0, P_1	-	-	-	-	-	-	-	-	-
0 P_2, P_3	-	-	-	[WARM-UP	PERIOD]	-	-	-	-
0 P_4, P_5	-	-	-	-	-	-	-	-	-
0 P_6, P_7	-	-	-	-	-	-	-	-	-
x_0 P_8, P_9	$R_7 = x_0 \cdot CW_{7,0} + P_7$	$R_6 = x_0 \cdot CW_{6,0} + P_6$	$R_5 = x_0 \cdot CW_{5,0} + P_5$	$R_4 = x_0 \cdot CW_{4,0} + P_4$	$R_3 = x_0 \cdot CW_{3,0} + P_3$	$R_2 = x_0 \cdot CW_{2,0} + P_2$	$R_1 = x_0 \cdot CW_{1,0} + P_1$	$R_0 = x_0 \cdot CW_{0,0} + P_0$	-
x_1 P_{10}, P_{11}	$R_7 + = x_1 \cdot CW_{7,1}$	$R_6 + = x_1 \cdot CW_{6,1}$	$R_5 + = x_1 \cdot CW_{5,1}$	$R_4 + = x_1 \cdot CW_{4,1}$	$R_3 + = x_1 \cdot CW_{3,1}$	$R_2 + = x_1 \cdot CW_{2,1}$	$R_1 + = x_1 \cdot CW_{1,1}$	$R_0 + = x_1 \cdot CW_{0,1}$	-
x_2 P_{12}, P_{13}	$R_7 + = x_2 \cdot CW_{7,2}$	$R_6 + = x_2 \cdot CW_{6,2}$	$R_5 + = x_2 \cdot CW_{5,2}$	$R_4 + = x_2 \cdot CW_{4,2}$	$R_3 + = x_2 \cdot CW_{3,2}$	$R_2 + = x_2 \cdot CW_{2,2}$	$R_1 + = x_2 \cdot CW_{1,2}$	$R_0 + = x_2 \cdot CW_{0,2}$	-
x_3 P_{14}, P_{15}	$R_7 + = x_3 \cdot CW_{7,3}$	$R_6 + = x_3 \cdot CW_{6,3}$	$R_5 + = x_3 \cdot CW_{5,3}$	$R_4 + = x_3 \cdot CW_{4,3}$	$R_3 + = x_3 \cdot CW_{3,3}$	$R_2 + = x_3 \cdot CW_{2,3}$	$R_1 + = x_3 \cdot CW_{1,3}$	$R_0 + = x_3 \cdot CW_{0,3}$	-
x_4 P_{16}, P_{17}	$R_{15} = x_4 \cdot CW_{7,0} + P_{15}$	$R_{14} = x_4 \cdot CW_{6,0} + P_{14}$	$R_{13} = x_4 \cdot CW_{5,0} + P_{13}$	$R_{12} = x_4 \cdot CW_{4,0} + P_{12}$	$R_{11} = x_4 \cdot CW_{3,0} + P_{11}$	$R_{10} = x_4 \cdot CW_{2,0} + P_{10}$	$R_9 = x_4 \cdot CW_{1,0} + P_9$	$R_8 = x_4 \cdot CW_{0,0} + P_8$	R_0, R_1
x_5 P_{18}, P_{19}	$R_{15} + = x_5 \cdot CW_{7,1}$	$R_{14} + = x_5 \cdot CW_{6,1}$	$R_{13} + = x_5 \cdot CW_{5,1}$	$R_{12} + = x_5 \cdot CW_{4,1}$	$R_{11} + = x_5 \cdot CW_{3,1}$	$R_{10} + = x_5 \cdot CW_{2,1}$	$R_9 + = x_5 \cdot CW_{1,1}$	$R_8 + = x_5 \cdot CW_{0,1}$	R_2, R_3
x_6 P_{20}, P_{21}	$R_{15} + = x_6 \cdot CW_{7,2}$	$R_{14} + = x_6 \cdot CW_{6,2}$	$R_{13} + = x_6 \cdot CW_{5,2}$	$R_{12} + = x_6 \cdot CW_{4,2}$	$R_{11} + = x_6 \cdot CW_{3,2}$	$R_{10} + = x_6 \cdot CW_{2,2}$	$R_9 + = x_6 \cdot CW_{1,2}$	$R_8 + = x_6 \cdot CW_{0,2}$	R_4, R_5
x_7 P_{22}, P_{23}	$R_{15} + = x_7 \cdot CW_{7,3}$	$R_{14} + = x_7 \cdot CW_{6,3}$	$R_{13} + = x_7 \cdot CW_{5,3}$	$R_{12} + = x_7 \cdot CW_{4,3}$	$R_{11} + = x_7 \cdot CW_{3,3}$	$R_{10} + = x_7 \cdot CW_{2,3}$	$R_9 + = x_7 \cdot CW_{1,3}$	$R_8 + = x_7 \cdot CW_{0,3}$	R_6, R_7
x_8 P_{24}, P_{25}	$R_{23} = x_8 \cdot CW_{7,0} + P_{23}$	$R_{22} = x_8 \cdot CW_{6,0} + P_{22}$	$R_{21} = x_8 \cdot CW_{5,0} + P_{21}$	$R_{20} = x_8 \cdot CW_{4,0} + P_{20}$	$R_{19} = x_8 \cdot CW_{3,0} + P_{19}$	$R_{18} = x_8 \cdot CW_{2,0} + P_{18}$	$R_{17} = x_8 \cdot CW_{1,0} + P_{17}$	$R_{16} = x_8 \cdot CW_{0,0} + P_{16}$	R_8, R_9
x_9 P_{26}, P_{27}	$R_{23} + = x_9 \cdot CW_{7,1}$	$R_{22} + = x_9 \cdot CW_{6,1}$	$R_{21} + = x_9 \cdot CW_{5,1}$	$R_{20} + = x_9 \cdot CW_{4,1}$	$R_{19} + = x_9 \cdot CW_{3,1}$	$R_{18} + = x_9 \cdot CW_{2,1}$	$R_{17} + = x_9 \cdot CW_{1,1}$	$R_{16} + = x_9 \cdot CW_{0,1}$	R_{10}, R_{11}
x_{10} P_{28}, P_{29}	$R_{23} + = x_{10} \cdot CW_{7,2}$	$R_{22} + = x_{10} \cdot CW_{6,2}$	$R_{21} + = x_{10} \cdot CW_{5,2}$	$R_{20} + = x_{10} \cdot CW_{4,2}$	$R_{19} + = x_{10} \cdot CW_{3,2}$	$R_{18} + = x_{10} \cdot CW_{2,2}$	$R_{17} + = x_{10} \cdot CW_{1,2}$	$R_{16} + = x_{10} \cdot CW_{0,2}$	R_{12}, R_{13}
x_{11} P_{30}, P_{31}	$R_{23} + = x_{11} \cdot CW_{7,3}$	$R_{22} + = x_{11} \cdot CW_{6,3}$	$R_{21} + = x_{11} \cdot CW_{5,3}$	$R_{20} + = x_{11} \cdot CW_{4,3}$	$R_{19} + = x_{11} \cdot CW_{3,3}$	$R_{18} + = x_{11} \cdot CW_{2,3}$	$R_{17} + = x_{11} \cdot CW_{1,3}$	$R_{16} + = x_{11} \cdot CW_{0,3}$	R_{14}, R_{15}
x_{12} P_{32}, P_{33}	$R_{31} = x_{12} \cdot CW_{7,0} + P_{31}$	$R_{30} = x_{12} \cdot CW_{6,0} + P_{30}$	$R_{29} = x_{12} \cdot CW_{5,0} + P_{29}$	$R_{28} = x_{12} \cdot CW_{4,0} + P_{28}$	$R_{27} = x_{12} \cdot CW_{3,0} + P_{27}$	$R_{26} = x_{12} \cdot CW_{2,0} + P_{26}$	$R_{25} = x_{12} \cdot CW_{1,0} + P_{25}$	$R_{24} = x_{12} \cdot CW_{0,0} + P_{24}$	R_{16}, R_{17}

P_n is *half-precision* input partial-sum n

x_n is an *f16v4* input vector

$CW_{m,n}$ is the common weight state $\$CWEI_{m,n}$

R_n is the final *half-precision* result of successive dot-product accumulations that began with P_n

² 0 input used to fill AMP pipeline during warm-up period

Table 3.113: *f16v4hihoamp* instruction definition

<i>f16v4hihoamp</i>	worker	aux
Syntax	f16v4hihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1, enumFlags	
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; vector<Single> out; array<vector<Single>,8> weights; array<Single,8> resultEven; array<Single,8> resultOdd; // Phase selection unsigned phase = F16AMP_ENUMFLAGS__PH__GET(op3); // Engine enables uint32_t ee = F16AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain if (0 == phase) { // Output from even accumulators out = { \$AACC[0], \$AACC[2] }; } else { // Output from odd accumulators out = { \$AACC[1], \$AACC[3] }; } vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight operands for (int k = 0; k < 8; k++) { if (engineEnable[k >> 1]) { // Weight selection - all fp16 uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = &fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
Except In		<pre> // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight operands for (int k = 0; k < 8; k++) { if (engineEnable[k >> 1]) { // Weight selection - all fp16 uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = &fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>

Continued on next page

Table 3.113: *f16v4hihoamp* instruction definition (continued)

<i>f16v4hihoamp</i>	worker	aux
Syntax	f16v4hihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1, enumFlags	
Semantics	<pre> Compute for (int k = 0; k < 8; k++) { unsigned engine = k >> 1; if (engineEnable[engine]) { // 4-element dot-product resultEven[k] = TFPU_F16DotProduct(weights[k], inputs); Single incomingp; if (0 == phase) { // Even accumulators - // combine incoming partial-sum (currently stored in our // odd-accumulator) with dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2) + 1], resultEven[k], TFPU_FP32); // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * 2]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs Half partialIn = op2[k & 1]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = (Single)partialIn; } } } else { // phase != 0 // Even accumulators - // accumulate dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2)], resultEven[k], TFPU_FP32); // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled incomingp = \$AACC[((k + 2) * 2) + 1]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs Half partialIn = op2[k & 1]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = (Single)partialIn; } } } resultOdd[k] = incomingp; } } </pre>	

Continued on next page

Table 3.113: *f16v4hihoamp* instruction definition (continued)

<i>f16v4hihoamp</i>		worker	aux
Syntax	f16v4hihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1, enumFlags		
Semantics	Compute cont'd	<pre> if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCRoadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> for (int k = 0; k < 8; k++) { unsigned engine = k >> 1; if (engineEnable[engine]) { \$AACC[(k * 2)] = resultEven[k]; \$AACC[(k * 2) + 1] = resultOdd[k]; } } for (i = 0; i < 2; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_GenSNanCheck*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_F32_QNan*, *TFPU_ApplyStochasticRound*, *TFPU_AACCRoadFlags*, *TFPU_IsMalign*

f16v4f16/f32amp calculation (4 phases)

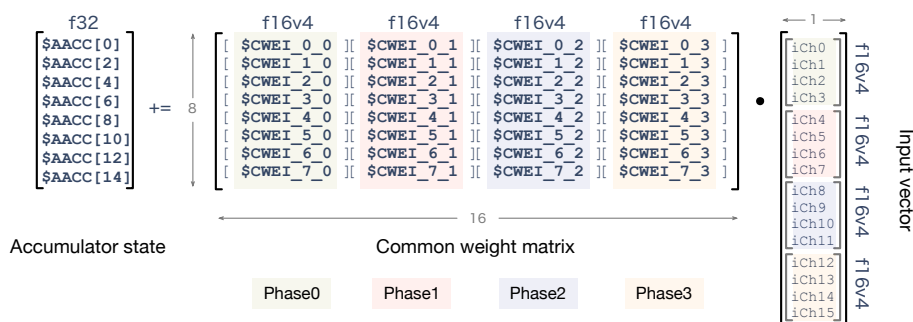


Fig. 3.12: f16v4hihoamp

f16v4f16amp convolution operation

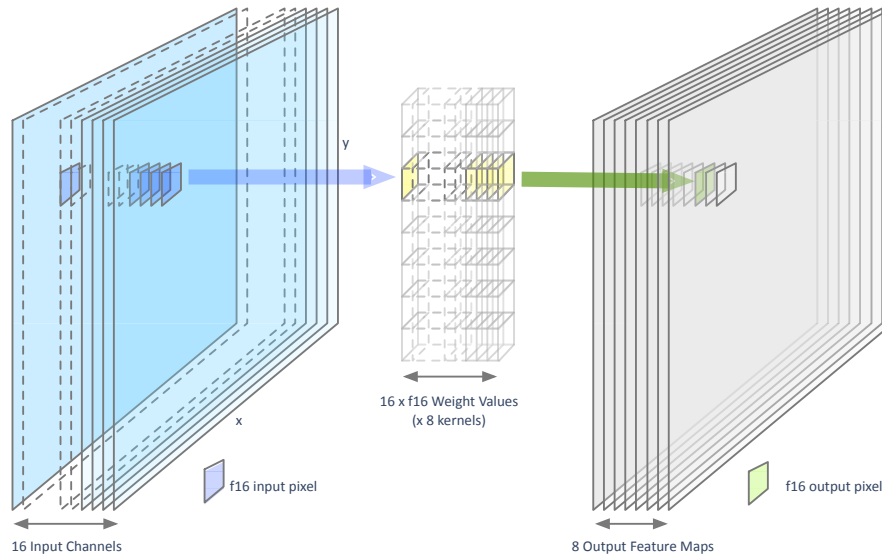


Fig. 3.13: f16v4hihoamp

f16v4hihoamp occurs in the following code examples:

- *f16v4hihoamp* example

Listing 3.5: f16v4hihoamp example

```
.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
_loop_start:
{
  ldst64pace  $inData, $outPartials, $striPtr+=", $mzero, 0
  f16v4hihoamp $outPartial0, $inData, $inPartial0, TAMP_F16V4_E4_P0
}
{
  ld2x64pace  $inData, $inPartials, $striPtr+=", $mzero, 0
  f16v4hihoamp $outPartial1, $inData, $inPartial1, TAMP_F16V4_E4_P1
}
{
  ldst64pace  $inData, $outPartials, $striPtr+=", $mzero, 0
  f16v4hihoamp $outPartial0, $inData, $inPartial0, TAMP_F16V4_E4_P2
}
{
  ld2x64pace  $inData, $inPartials, $striPtr+=", $mzero, 0
  f16v4hihoamp $outPartial1, $inData, $inPartial1, TAMP_F16V4_E4_P3
}
_loop_end:
```

3.7.3.3.17 f16v4hihoslic

Half-precision floating point vector slim convolution. Input partial-sums are *half-precision*. Results are *half-precision*

Table 3.114: f16v4hihoslic, 2x1x3x4 example sequence

$\$aSrc0 I$	$\$AACC[14]$	$\$AACC[10]$	$\$AACC[6]$	$\$AACC[2]$	$\$AACC[12]$	$\$AACC[8]$	$\$AACC[4]$	$\$AACC[0]$	$\$aDst0$
$x_0 P_0,P_1$	-	$R_1 = x_0 \cdot CW_{5,0} + P_1$	-	-	-	$R_0 = x_0 \cdot CW_{4,0} + P_0$	-	-	-
$x_1 P_2,P_3$	-	$R_3 = x_1 \cdot CW_{5,0} + P_3$	$R_1 + = x_1 \cdot CW_{3,0}$	-	-	$R_2 = x_1 \cdot CW_{4,0} + P_2$	$R_0 + = x_1 \cdot CW_{2,0}$	-	-
$x_2 P_4,P_5$	-	$R_5 = x_2 \cdot CW_{5,0} + P_5$	$R_3 + = x_2 \cdot CW_{3,0}$	$R_1 + = x_2 \cdot CW_{1,0}$	-	$R_4 = x_2 \cdot CW_{4,0} + P_4$	$R_2 + = x_2 \cdot CW_{2,0}$	$R_0 + = x_2 \cdot CW_{0,0}$	-
$x_3 P_6,P_7$	-	$R_7 = x_3 \cdot CW_{5,0} + P_7$	$R_5 + = x_3 \cdot CW_{3,0}$	$R_3 + = x_3 \cdot CW_{1,0}$	-	$R_6 = x_3 \cdot CW_{4,0} + P_6$	$R_4 + = x_3 \cdot CW_{2,0}$	$R_2 + = x_3 \cdot CW_{0,0}$	R_0, R_1
$x_4 P_8,P_9$	-	$R_9 = x_4 \cdot CW_{5,0} + P_9$	$R_7 + = x_4 \cdot CW_{3,0}$	$R_5 + = x_4 \cdot CW_{1,0}$	-	$R_8 = x_4 \cdot CW_{4,0} + P_8$	$R_6 + = x_4 \cdot CW_{2,0}$	$R_4 + = x_4 \cdot CW_{0,0}$	R_2, R_3
$x_5 P_{10},P_{11}$	-	$R_{11} = x_5 \cdot CW_{5,0} + P_{11}$	$R_9 + = x_5 \cdot CW_{3,0}$	$R_7 + = x_5 \cdot CW_{1,0}$	-	$R_{10} = x_5 \cdot CW_{4,0} + P_{10}$	$R_8 + = x_5 \cdot CW_{2,0}$	$R_6 + = x_5 \cdot CW_{0,0}$	R_4, R_5
$x_6 P_{12},P_{13}$	-	$R_{13} = x_6 \cdot CW_{5,0} + P_{13}$	$R_{11} + = x_6 \cdot CW_{3,0}$	$R_9 + = x_6 \cdot CW_{1,0}$	-	$R_{12} = x_6 \cdot CW_{4,0} + P_{12}$	$R_{10} + = x_6 \cdot CW_{2,0}$	$R_8 + = x_6 \cdot CW_{0,0}$	R_6, R_7

Table 3.115: f16v4hihoslic, 1x4 example sequence

$\$aSrc0 I$	$\$AACC[14]$	$\$AACC[10]$	$\$AACC[6]$	$\$AACC[2]$	$\$AACC[12]$	$\$AACC[8]$	$\$AACC[4]$	$\$AACC[0]$	$\$aDst0$
$x_0 P_0,P_1$	$R_1 = x_0 \cdot CW_{7,0} + P_1$	-	-	-	$R_0 = x_0 \cdot CW_{6,0} + P_0$	-	-	-	-
$x_1 P_2,P_3$	$R_3 = x_1 \cdot CW_{7,0} + P_3$	$R_1 + = x_1 \cdot CW_{5,0}$	-	-	$R_2 = x_1 \cdot CW_{6,0} + P_2$	$R_0 + = x_1 \cdot CW_{4,0}$	-	-	-
$x_2 P_4,P_5$	$R_5 = x_2 \cdot CW_{7,0} + P_5$	$R_3 + = x_2 \cdot CW_{5,0}$	$R_1 + = x_2 \cdot CW_{3,0}$	-	$R_4 = x_2 \cdot CW_{6,0} + P_4$	$R_2 + = x_2 \cdot CW_{4,0}$	$R_0 + = x_2 \cdot CW_{2,0}$	-	-
$x_3 P_6,P_7$	$R_7 = x_3 \cdot CW_{7,0} + P_7$	$R_5 + = x_3 \cdot CW_{5,0}$	$R_3 + = x_3 \cdot CW_{3,0}$	$R_1 + = x_3 \cdot CW_{1,0}$	$R_6 = x_3 \cdot CW_{6,0} + P_6$	$R_4 + = x_3 \cdot CW_{4,0}$	$R_2 + = x_3 \cdot CW_{2,0}$	$R_0 + = x_3 \cdot CW_{0,0}$	-
$x_4 P_8,P_9$	$R_9 = x_4 \cdot CW_{7,0} + P_9$	$R_7 + = x_4 \cdot CW_{5,0}$	$R_5 + = x_4 \cdot CW_{3,0}$	$R_3 + = x_4 \cdot CW_{1,0}$	$R_8 = x_4 \cdot CW_{6,0} + P_8$	$R_6 + = x_4 \cdot CW_{4,0}$	$R_4 + = x_4 \cdot CW_{2,0}$	$R_2 + = x_4 \cdot CW_{0,0}$	R_0, R_1
$x_5 P_{10},P_{11}$	$R_{11} = x_5 \cdot CW_{7,0} + P_{11}$	$R_9 + = x_5 \cdot CW_{5,0}$	$R_7 + = x_5 \cdot CW_{3,0}$	$R_5 + = x_5 \cdot CW_{1,0}$	$R_{10} = x_5 \cdot CW_{6,0} + P_{10}$	$R_8 + = x_5 \cdot CW_{4,0}$	$R_6 + = x_5 \cdot CW_{2,0}$	$R_4 + = x_5 \cdot CW_{0,0}$	R_2, R_3
$x_6 P_{12},P_{13}$	$R_{13} = x_6 \cdot CW_{7,0} + P_{13}$	$R_{11} + = x_6 \cdot CW_{5,0}$	$R_9 + = x_6 \cdot CW_{3,0}$	$R_7 + = x_6 \cdot CW_{1,0}$	$R_{12} = x_6 \cdot CW_{6,0} + P_{12}$	$R_{10} + = x_6 \cdot CW_{4,0}$	$R_8 + = x_6 \cdot CW_{2,0}$	$R_6 + = x_6 \cdot CW_{0,0}$	R_4, R_5

P_n is *half-precision* input partial-sum n

x_n is an *f16v4* input vector

$CW_{m,n}$ is the common weight state $\$CWEI_m_n$

R_n is the final *half-precision* result of successive dot-product accumulations that began with P_n

Table 3.116: *f16v4hihoslic* instruction definition

<i>f16v4hihoslic</i>	worker	aux
Syntax	<code>f16v4hihoslic \$aDst0, \$aSrc0:Src0+1, \$aSrc1, enumFlags</code>	
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,2> op2 = { pickHalf(\$aSrc1[0], 0), pickHalf(\$aSrc1[0], 1) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; array<vector<Single>,8> weights; array<Single,8> result; // engine enables uint32_t ee = F16SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain vector<Single> out = { \$AACC[0], \$AACC[2] }; vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; </pre>
Except In		<pre> uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight selection unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3); for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { uint64_t fullCwei = context.getCCCSState((u * 4) + wid); uint32_t *cwei = &fullCwei; weights[u] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[u].data(), weights[u].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
Compute		<pre> for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { Single incomingp; if (engineEnable[(u >> 1) + 1]) { // Engine behind me is enabled // use its current accumulator value incomingp = \$AACC[(u + 2) * 2]; </pre>

Continued on next page

Table 3.116: *f16v4hihoslic* instruction definition (continued)

<i>f16v4hihoslic</i>		worker	aux
Syntax	<code>f16v4hihoslic \$aDst0, \$aSrc0:Src0+1, \$aSrc1, enumFlags</code>		
Semantics	Compute cont'd	<pre> } else { // Engines behind me are disabled (or I am the final engine // in the chain). Use the new partial-sum inputs Half partialIn = op2[u & 1]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = partialIn; } } // 4-element dot-product result[u] = TFPU_F16DotProduct(weights[u], inputs); // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32); } } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCRReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>		
Commit	<pre> for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { \$AACC[u * 2] = result[u]; } } for (i = 0; i < 2; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16) }; </pre>		

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_GenSNanCheck*, *TFPU_F32_QNan*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_ApplyStochasticRound*, *TFPU_AACCRReadFlags*, *TFPU_IsMalign*

f16v4f16slc convolution operation

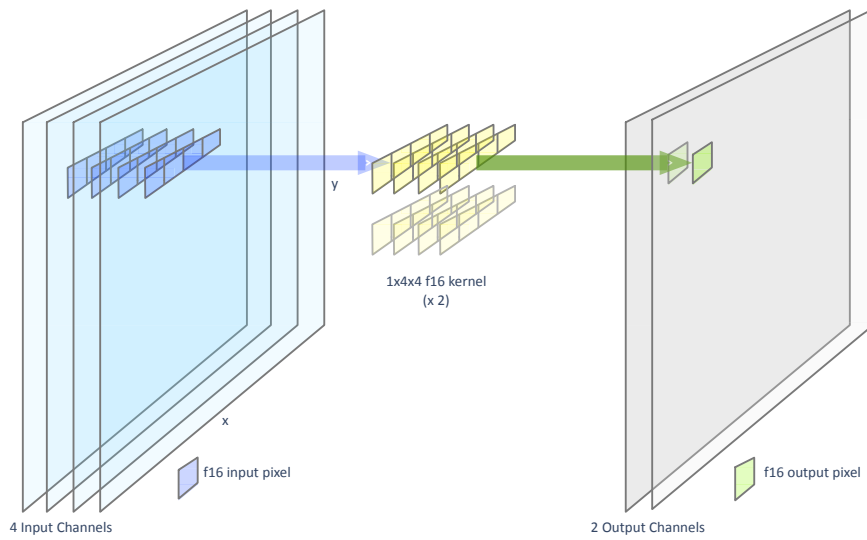


Fig. 3.14: f16v4hihoslic

3.7.3.3.18 f16v4hihov4amp

Half-precision floating point vector accumulating matrix-vector product. Input and result partial-sums are 4-element *half-precision* vectors.

Table 3.117: *f16v4hihov4amp* instruction definition

<i>f16v4hihov4amp</i>	worker	aux
Syntax	f16v4hihov4amp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; vector<Single> out; array<vector<Single>,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weights; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> resultEven; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> resultOdd; // Extract immediate config unsigned phase = F16AMP_ENUMFLAGS__PH__GET(op3); unsigned ampSets = 2; unsigned ampUnits = TFPU_AMP_UNITS_PER_SET * ampSets; // Engine enables uint32_t ee = F16AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain unsigned aaccPerSet = TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; if (phase == 0) { // Output from even accumulators out = { \$AACC[0], \$AACC[2], \$AACC[aaccPerSet + 0], \$AACC[aaccPerSet + 2] }; } else { // Output from odd accumulators out = { \$AACC[1], \$AACC[3], \$AACC[aaccPerSet + 1], \$AACC[aaccPerSet + 3] }; } vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; Except In // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1], op2[2], op2[3] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight operands for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = &fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; } } </pre>	

Continued on next page

Table 3.117: *f16v4hihov4amp* instruction definition (continued)

<i>f16v4hihov4amp</i>		worker	aux
Syntax	f16v4hihov4amp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Except In cont'd	<pre> fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	
Compute		<pre> // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; unsigned set = k / TFPU_AMP_UNITS_PER_SET; unsigned partialIndex = (set * 2) + (k & 1); if (engineEnable[engine]) { // 4-element dot-product resultEven[k] = TFPU_F16DotProduct(weights[k], inputs); Single incomingp; if (phase == 0) { // Even accumulators - // combine incoming partial-sum (currently stored in our // odd-accumulator) with dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2) + 1], resultEven[k], TFPU_FP32); // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * 2]; } } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs Half partialIn = op2[partialIndex]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = (Single)partialIn; } } } } else { // Phase != 0 // Even accumulators - // accumulate dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2)], resultEven[k], TFPU_FP32); // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled // Take the current value from its odd accumulator incomingp = \$AACC[((k + 2) * 2) + 1]; } } </pre>	

Continued on next page

Table 3.117: *f16v4hihov4amp* instruction definition (continued)

<i>f16v4hihov4amp</i>		worker	aux
Syntax	f16v4hihov4amp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Compute cont'd	<pre> } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs Half partialIn = op2[partialIndex]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = (Single)partialIn; } } } resultOdd[k] = incomingp; } } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[k * 2] = resultEven[k]; \$AACC[k * 2 + 1] = resultOdd[k]; } } for (unsigned i = 0; i < 4; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16), Half(out[2]).bitz32(smode) (Half(out[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_GenSNanCheck*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_F32_QNan*, *TFPU_ApplyStochasticRound*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*

3.7.3.3.19 f16v4hihov4slic

Half-precision floating point vector slim convolution. Input and result partial-sums are 4 x *half-precision* values.

Table 3.118: *f16v4hihov4slic* instruction definition

<i>f16v4hihov4slic</i>	worker	aux
Syntax	f16v4hihov4slic \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; array<vector<Single>,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weights; array<Double,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; // Extract immediate config unsigned ampSets = 2; unsigned ampUnits = TFPU_AMP_UNITS_PER_SET * ampSets; unsigned aaccPerSet = TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; // Engine enables uint32_t ee = F16SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain // Output from even accumulators vector<Single> out = { \$AACC[0], \$AACC[2], \$AACC[aaccPerSet + 0], \$AACC[aaccPerSet + 2] }; vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; Except In uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1], op2[2], op2[3] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight set selection unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { uint64_t fullCwei = context.getCCSState((u * 4) + wid); uint32_t *cwei = &fullCwei; weights[u] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[u].data(), weights[u].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	

Continued on next page

Table 3.118: *f16v4hihov4slic* instruction definition (continued)

<i>f16v4hihov4slic</i>		worker	aux
Syntax	f16v4hihov4slic \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Compute	<pre> // Input partial-sum consumption for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { Single incomingp; if (engineEnable[engine + 1]) { // Engine behind me is enabled // use its current accumulator value incomingp = \$AACC[(u + 2) * 2]; } else { // Engines behind me are disabled (or I am the final engine // in the chain). Use the new partial-sum inputs unsigned set = u / TFPU_AMP_UNITS_PER_SET; unsigned partialIndex = (set * 2) + (u & 1); Half partialIn = op2[partialIndex]; if (partialIn.isNaN()) { incomingp = TFPU_F32_QNan(); } else { incomingp = (Single)partialIn; } } } // 4-element dot-product result[u] = TFPU_F16DotProduct(weights[u], inputs); // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32); } } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> // Internal state updates for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[u * 2] = result[u]; } } for (unsigned i = 0; i < 4; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } </pre>	

Continued on next page

Table 3.118: *f16v4hihov4slic* instruction definition (continued)

<i>f16v4hihov4slic</i>		worker	aux
Syntax	f16v4hihov4slic \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Commit cont'd	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16), Half(out[2]).bitz32(smode) (Half(out[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_GenSNanCheck*, *TFPU_F32_QNan*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_ApplyStochasticRound*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*

3.7.3.3.20 f16v4istacc

Sort/shuffle (permute) through accumulators, with new input.

- Present 128-bits of register operand source data to be sorted/shuffled (other otherwise permuted) using the *\$AACC* state. The precise behaviour is dependent on the value of the immediate.
- Perform *\$AACC* state propagation as specified by the immediate.
- The destination register pair is written with 64-bits of result data from a combination of *\$AACC* registers. The precise combination is specified by the immediate.

Table 3.119: *f16v4istacc* instruction definition

<i>f16v4istacc</i>	worker	aux
Syntax	<code>f16v4istacc \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags</code>	
Semantics	<p>Prepare</p> <pre> array<HalfDataWord,4> op1 = { ((\$aSrc0:Src0+1[0] >> 0) & 0xffff), ((\$aSrc0:Src0+1[0] >> 16) & 0xffff), ((\$aSrc0:Src0+1[1] >> 0) & 0xffff), ((\$aSrc0:Src0+1[1] >> 16) & 0xffff) }; array<HalfDataWord,4> op2 = { ((\$aSrc1:Src1+1[0] >> 0) & 0xffff), ((\$aSrc1:Src1+1[0] >> 16) & 0xffff), ((\$aSrc1:Src1+1[1] >> 0) & 0xffff), ((\$aSrc1:Src1+1[1] >> 16) & 0xffff) }; DataWord op3 = enumFlags; Commit int phase = op3 &1; uint32_t result[2] = { \$AACC[1], \$AACC[3] }; // Read and sort input values if (phase == 0) { // Propagate head of current \$AACC state (odd chain) \$AACC[1] = \$AACC[5]; \$AACC[3] = \$AACC[7]; \$AACC[0] = uint32AsFloat((op2[0] << 16) op1[0]); \$AACC[4] = uint32AsFloat((op2[1] << 16) op1[1]); \$AACC[8] = uint32AsFloat((op2[2] << 16) op1[2]); \$AACC[12] = uint32AsFloat((op2[3] << 16) op1[3]); } else { // Phase == 1 \$AACC[2] = uint32AsFloat((op2[0] << 16) op1[0]); \$AACC[6] = uint32AsFloat((op2[1] << 16) op1[1]); \$AACC[10] = uint32AsFloat((op2[2] << 16) op1[2]); \$AACC[14] = uint32AsFloat((op2[3] << 16) op1[3]); } \$aDst0:Dst0+1 = { result[0], result[1] }; </pre>	

Architectural state references: *\$AACC*

f16v4istacc occurs in the following code examples:

- *f16v4stacc* example

3.7.3.3.21 *f16v4max*

Half-precision 4-element vector element-wise max

Table 3.120: *f16v4max* instruction definition

<i>f16v4max</i>	worker	aux
Syntax	f16v4max \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<Half,4> z; Except In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute z[0] = TFPU_Max(op1[0], op2[0]); z[1] = TFPU_Max(op1[1], op2[1]); z[2] = TFPU_Max(op1[2], op2[2]); z[3] = TFPU_Max(op1[3], op2[3]); Commit \$aDst0:Dst0+1 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16), Half(z[2]).bitz32(PROP_INF) (Half(z[3]).bitz32(PROP_INF) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_Max*

3.7.3.3.22 f16v4maxc

Half-precision 4-element vector 2x2 lateral maximum.

Table 3.121: *f16v4maxc* instruction definition

<i>f16v4maxc</i>		worker	aux
Syntax	<code>f16v4maxc \$aDst0, \$aSrc0:Src0+1</code>		
Semantics	<p>Prepare</p> <pre>array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,2> max;</pre> <p>Except In</p> <pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op1[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre> <p>Compute</p> <pre>max[0] = TFPU_Max(op1[0], op1[1]); max[1] = TFPU_Max(op1[2], op1[3]);</pre> <p>Commit</p> <pre>\$aDst0 = { Half(max[0]).bitz32(PROP_INF) (Half(max[1]).bitz32(PROP_INF) << 16) };</pre>		

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_Max*

3.7.3.3.23 *f16v4min*

Half-precision 4-element vector element-wise minimum

Table 3.122: *f16v4min* instruction definition

<i>f16v4min</i>	worker	aux
Syntax	f16v4min \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; array<Half,4> z; Except In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (op1[i].issNaN() op2[i].issNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute z[0] = TFPU_Min(op1[0], op2[0]); z[1] = TFPU_Min(op1[1], op2[1]); z[2] = TFPU_Min(op1[2], op2[2]); z[3] = TFPU_Min(op1[3], op2[3]); Commit \$aDst0:Dst0+1 = { Half(z[0]).bitz32(PROP_INF) (Half(z[1]).bitz32(PROP_INF) << 16), Half(z[2]).bitz32(PROP_INF) (Half(z[3]).bitz32(PROP_INF) << 16) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_IsMalign*, *TFPU_Min*

3.7.3.3.24 f16v4mix

Half-precision 4-element vector $\mathbf{z} = \mathbf{ax} + \mathbf{by}$. The scalar multiplicands a and b are provided by the internal state element *\$TAS*.

Results are stored within the accumulator state. Destination registers are written with the previous accumulator state.

Table 3.123: *f16v4mix* instruction definition

<i>f16v4mix</i>	worker	aux
Syntax	f16v4mix \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1	
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; array<bool,4> specialCase; array<Single,4> result; vector<Single> out = { \$AACC[0], \$AACC[2], \$AACC[4], \$AACC[6] }; Half a = pickHalf(\$TAS, 0); Half b = pickHalf(\$TAS, 1); </pre>
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; // Input exception check for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoAxpbyPreExecute(a, op1[i], b, op2[i], &fpExcpt, &result[i]); } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
	Compute	<pre> if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } // z = ax + by for (i = 0; i < 4; ++i) { if (!specialCase[i]) { vector<Single> xv = { a, b }; vector<Single> yv = { op1[i], op2[i] }; result[i] = TFPU_F16DotProduct(xv, yv); } } </pre>
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>
	Commit	<pre> \$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3]; for (i = 0; i < 4; ++i) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNaN(); } } </pre>

Table 3.123: *f16v4mix* instruction definition (continued)

<i>f16v4mix</i>		worker	aux
Syntax	f16v4mix \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1		
Semantics	Commit cont'd	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16), Half(out[2]).bitz32(smode) (Half(out[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$TAS*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAxpbyPreExecute*, *TFPU_ApplyStochasticRound*, *TFPU_F16DotProduct*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QNan*

3.7.3.3.25 f16v4mul

Half-precision 4-element vector, Hadamard product

Table 3.124: *f16v4mul* instruction definition

<i>f16v4mul</i>	worker	aux
Syntax	<pre>f16v4mul \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4mul \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4mul \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1</pre>	
Semantics	Prepare	<pre>array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; TileRoundMode_t rmode; vector<Single> result; array<bool,4> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; rmode = TFPU_ROUND_EVEN; } else { enableStochasticRounding = false; rmode = \$FP_CTL.RND; } Except In uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(4); for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoMulPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } Compute for (i = 0; i < 4; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) * static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, rmode); } } Except Out if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 4; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } } </pre>

Continued on next page

Table 3.124: *f16v4mul* instruction definition (continued)

<i>f16v4mul</i>		worker	aux
Syntax	f16v4mul \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4mul \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4mul \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1		
Semantics	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoMulPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_ApplyStochasticRound*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*

3.7.3.3.26 f16v4rmask

Half-precision floating-point vector random mask.

The result is a masked version of the input vector, with each element of the input being individually masked with the probability specified by the bottom 17-bits of the 2nd input operand:

- if $\$aSrc1[16] == 1$, no masking is applied (the result is a copy of the input vector)
- else if $\$aSrc1[16:0] == 0$, the result is a zero vector
- otherwise each element is individually unmasked with probability $\frac{\$aSrc1[15:0]}{65536}$

PRNG is used by this instruction to generate 4 x 16-bit random values from the discrete uniform distribution.

Table 3.125: *f16v4rmask* instruction definition

<i>f16v4rmask</i>	worker	aux
Syntax	f16v4rmask \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1	
Semantics	<p>Prepare</p> <pre>array<HalfDataWord,4> op0; array<HalfDataWord,4> op1 = { ((\$aSrc0:Src0+1[0] >> 0) & 0xffff), ((\$aSrc0:Src0+1[0] >> 16) & 0xffff), ((\$aSrc0:Src0+1[1] >> 0) & 0xffff), ((\$aSrc0:Src0+1[1] >> 16) & 0xffff) }; DataWord op2 = \$aSrc1;</pre> <p>Compute</p> <pre>bool always = ((op2 >> 16) & 1) == 1; bool never = (op2 & 0x1ffff) == 0; array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); op0 = { 0, 0, 0, 0 }; if (always) { // No masking required op0 = { op1[0], op1[1], op1[2], op1[3] }; } else if (!never) { uint16_t prob = op2 & 0xffff; // Mask out the 16-bit elements based on the random bit-patterns // and prob if (((randomBits[0] >> 0) & 0xffff) < prob) { op0[0] = op1[0]; } if (((randomBits[0] >> 16) & 0xffff) < prob) { op0[1] = op1[1]; } if (((randomBits[0] >> 32) & 0xffff) < prob) { op0[2] = op1[2]; } if (((randomBits[0] >> 48) & 0xffff) < prob) { op0[3] = op1[3]; } } Commit \$aDst0:Dst0+1 = { ((op0[0] & 0xffff) << 0) ((op0[1] & 0xffff) << 16), ((op0[2] & 0xffff) << 0) ((op0[3] & 0xffff) << 16) };</pre>	

Architectural state references: *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*

3.7.3.3.27 f16v4sihoamp

Half-precision floating point vector accumulating matrix-vector product. Input partial-sums are *single-precision*. Result values are *half-precision*.

Table 3.126: *f16v4sihoamp* instruction definition

<i>f16v4sihoamp</i>	worker	aux
Syntax	f16v4sihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; vector<Single> out; array<vector<Single>,8> weights; array<Single,8> resultEven; array<Single,8> resultOdd; // Phase selection unsigned phase = F16AMP_ENUMFLAGS__PH__GET(op3); // Engine enables uint32_t ee = F16AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain if (0 == phase) { // Output from even accumulators out = { \$AACC[0], \$AACC[2] }; } else { // Output from odd accumulators out = { \$AACC[1], \$AACC[3] }; } vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); for (int k = 0; k < 8; k++) { if (engineEnable[k >> 1]) { // Weight selection - all fp16 uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = &fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
Except In		<pre> // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); for (int k = 0; k < 8; k++) { if (engineEnable[k >> 1]) { // Weight selection - all fp16 uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = &fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>

Continued on next page

Table 3.126: *f16v4sihoamp* instruction definition (continued)

<i>f16v4sihoamp</i>	worker	aux
Syntax	<i>f16v4sihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags</i>	
Semantics	<pre> Compute for (int k = 0; k < 8; k++) { if (engineEnable[k >> 1]) { } } for (int k = 0; k < 8; k++) { unsigned engine = k >> 1; if (engineEnable[engine]) { // 4-element dot-product resultEven[k] = TFPU_F16DotProduct(weights[k], inputs); Single incomingp; if (0 == phase) { // Even accumulators - // combine incoming partial-sum (currently stored in our // odd-accumulator) with dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2) + 1], resultEven[k], TFPU_FP32); // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * 2]; } } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } } else { // phase != 0 // Even accumulators - // accumulate dot-product result resultEven[k] = TFPU_Add(\$AACC[k * 2], resultEven[k], TFPU_FP32); // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled incomingp = \$AACC[((k + 2) * 2) + 1]; } } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } } </pre>	

Continued on next page

Table 3.126: *f16v4sihoamp* instruction definition (continued)

<i>f16v4sihoamp</i>		worker	aux
Syntax	f16v4sihoamp \$aDst0, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Compute cont'd	<pre> resultOdd[k] = incomingp; } } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> for (int k = 0; k < 8; k++) { unsigned engine = k >> 1; if (engineEnable[engine]) { \$AACC[(k * 2)] = resultEven[k]; \$AACC[(k * 2) + 1] = resultOdd[k]; } } for (i = 0; i < 2; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_F32_QNan*, *TFPU_ApplyStochasticRound*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*

3.7.3.3.28 f16v4sihoslic

Half-precision floating point vector slim convolution. Input partial-sums are *single-precision*. Results are *half-precision*

Table 3.127: *f16v4sihoslic* instruction definition

<i>f16v4sihoslic</i>	worker	aux
Syntax	<code>f16v4sihoslic \$aDst0, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags</code>	
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; bool nanoo = \$FP_CTL.NANOO; array<uint64_t,2> randomBits; array<vector<Single>,8> weights; array<Single,8> result; // engine enables uint32_t ee = F16SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain vector<Single> out = { \$AACC[0], \$AACC[2] }; vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; </pre>
Except In		<pre> uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight selection unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3); // Weight operands for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { uint64_t fullCwei = context.getCCCSState((u * 4) + wid); uint32_t *cwei = &fullCwei; weights[u] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[u].data(), weights[u].size()); } } if (\$FP_CTL.ESR) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>
Compute		<pre> for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { Single incomingp; if (engineEnable[(u >> 1) + 1]) { // Engine behind me is enabled // use its current accumulator value incomingp = \$AACC[(u + 2) * 2]; </pre>

Continued on next page

Table 3.127: *f16v4sihoslic* instruction definition (continued)

<i>f16v4sihoslic</i>		worker	aux
Syntax	f16v4sihoslic \$aDst0, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Compute cont'd	<pre> } else { // Engines behind me are disabled (or I am the final engine // in the chain). Use the new partial-sum inputs incomingp = op2[u & 1]; } // 4-element dot-product result[u] = TFPU_F16DotProduct(weights[u], inputs); // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32); } } if (\$FP_CTL.ESR) { TFPU_ApplyStochasticRound(randomBits, out); } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcp = TFPU_AACCRReadFlags(out, TFPU_FP16, nanoo); \$FP_STS = \$FP_STS fpExcp; if (TFPU_IsMalign(fpExcp, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> for (int u = 0; u < 8; u++) { if (engineEnable[u >> 1]) { \$AACC[u * 2] = result[u]; } } for (i = 0; i < 2; i++) { if (isinf(out[i]) isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0 = { Half(out[0]).bitz32(smode) (Half(out[1]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$AACC*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_F16DotProduct*, *TFPU_Add*, *TFPU_ApplyStochasticRound*, *TFPU_AACCRReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QNan*

3.7.3.3.29 f16v4sisoamp

Half-precision floating point vector accumulating matrix-vector product. Input and result partial-sums are 2 x *single-precision* values.

Table 3.128: f16v4sisoamp 8x1x16 example sequence

$\$aSrc0$ I	$\$AACC[14]$	$\$AACC[12]$	$\$AACC[10]$	$\$AACC[8]$	$\$AACC[6]$	$\$AACC[4]$	$\$AACC[2]$	$\$AACC[0]$	$\$aDst0$
0^3 P_0, P_1	-	-	-	-	-	-	-	-	-
0 P_2, P_3	-	-	-	[WARM-UP	PERIOD]	-	-	-	-
0 P_4, P_5	-	-	-	-	-	-	-	-	-
0 P_6, P_7	-	-	-	-	-	-	-	-	-
x_0 P_8, P_9	$R_7 = x_0 \cdot CW_{7,0} + P_7$	$R_6 = x_0 \cdot CW_{6,0} + P_6$	$R_5 = x_0 \cdot CW_{5,0} + P_5$	$R_4 = x_0 \cdot CW_{4,0} + P_4$	$R_3 = x_0 \cdot CW_{3,0} + P_3$	$R_2 = x_0 \cdot CW_{2,0} + P_2$	$R_1 = x_0 \cdot CW_{1,0} + P_1$	$R_0 = x_0 \cdot CW_{0,0} + P_0$	-
x_1 P_{10}, P_{11}	$R_7 + = x_1 \cdot CW_{7,1}$	$R_6 + = x_1 \cdot CW_{6,1}$	$R_5 + = x_1 \cdot CW_{5,1}$	$R_4 + = x_1 \cdot CW_{4,1}$	$R_3 + = x_1 \cdot CW_{3,1}$	$R_2 + = x_1 \cdot CW_{2,1}$	$R_1 + = x_1 \cdot CW_{1,1}$	$R_0 + = x_1 \cdot CW_{0,1}$	-
x_2 P_{12}, P_{13}	$R_7 + = x_2 \cdot CW_{7,2}$	$R_6 + = x_2 \cdot CW_{6,2}$	$R_5 + = x_2 \cdot CW_{5,2}$	$R_4 + = x_2 \cdot CW_{4,2}$	$R_3 + = x_2 \cdot CW_{3,2}$	$R_2 + = x_2 \cdot CW_{2,2}$	$R_1 + = x_2 \cdot CW_{1,2}$	$R_0 + = x_2 \cdot CW_{0,2}$	-
x_3 P_{14}, P_{15}	$R_7 + = x_3 \cdot CW_{7,3}$	$R_6 + = x_3 \cdot CW_{6,3}$	$R_5 + = x_3 \cdot CW_{5,3}$	$R_4 + = x_3 \cdot CW_{4,3}$	$R_3 + = x_3 \cdot CW_{3,3}$	$R_2 + = x_3 \cdot CW_{2,3}$	$R_1 + = x_3 \cdot CW_{1,3}$	$R_0 + = x_3 \cdot CW_{0,3}$	-
x_4 P_{16}, P_{17}	$R_{15} = x_4 \cdot CW_{7,0} + P_{15}$	$R_{14} = x_4 \cdot CW_{6,0} + P_{14}$	$R_{13} = x_4 \cdot CW_{5,0} + P_{13}$	$R_{12} = x_4 \cdot CW_{4,0} + P_{12}$	$R_{11} = x_4 \cdot CW_{3,0} + P_{11}$	$R_{10} = x_4 \cdot CW_{2,0} + P_{10}$	$R_9 = x_4 \cdot CW_{1,0} + P_9$	$R_8 = x_4 \cdot CW_{0,0} + P_8$	R_0, R_1
x_5 P_{18}, P_{19}	$R_{15} + = x_5 \cdot CW_{7,1}$	$R_{14} + = x_5 \cdot CW_{6,1}$	$R_{13} + = x_5 \cdot CW_{5,1}$	$R_{12} + = x_5 \cdot CW_{4,1}$	$R_{11} + = x_5 \cdot CW_{3,1}$	$R_{10} + = x_5 \cdot CW_{2,1}$	$R_9 + = x_5 \cdot CW_{1,1}$	$R_8 + = x_5 \cdot CW_{0,1}$	R_2, R_3
x_6 P_{20}, P_{21}	$R_{15} + = x_6 \cdot CW_{7,2}$	$R_{14} + = x_6 \cdot CW_{6,2}$	$R_{13} + = x_6 \cdot CW_{5,2}$	$R_{12} + = x_6 \cdot CW_{4,2}$	$R_{11} + = x_6 \cdot CW_{3,2}$	$R_{10} + = x_6 \cdot CW_{2,2}$	$R_9 + = x_6 \cdot CW_{1,2}$	$R_8 + = x_6 \cdot CW_{0,2}$	R_4, R_5
x_7 P_{22}, P_{23}	$R_{15} + = x_7 \cdot CW_{7,3}$	$R_{14} + = x_7 \cdot CW_{6,3}$	$R_{13} + = x_7 \cdot CW_{5,3}$	$R_{12} + = x_7 \cdot CW_{4,3}$	$R_{11} + = x_7 \cdot CW_{3,3}$	$R_{10} + = x_7 \cdot CW_{2,3}$	$R_9 + = x_7 \cdot CW_{1,3}$	$R_8 + = x_7 \cdot CW_{0,3}$	R_6, R_7
x_8 P_{24}, P_{25}	$R_{23} = x_8 \cdot CW_{7,0} + P_{23}$	$R_{22} = x_8 \cdot CW_{6,0} + P_{22}$	$R_{21} = x_8 \cdot CW_{5,0} + P_{21}$	$R_{20} = x_8 \cdot CW_{4,0} + P_{20}$	$R_{19} = x_8 \cdot CW_{3,0} + P_{19}$	$R_{18} = x_8 \cdot CW_{2,0} + P_{18}$	$R_{17} = x_8 \cdot CW_{1,0} + P_{17}$	$R_{16} = x_8 \cdot CW_{0,0} + P_{16}$	R_8, R_9
x_9 P_{26}, P_{27}	$R_{23} + = x_9 \cdot CW_{7,1}$	$R_{22} + = x_9 \cdot CW_{6,1}$	$R_{21} + = x_9 \cdot CW_{5,1}$	$R_{20} + = x_9 \cdot CW_{4,1}$	$R_{19} + = x_9 \cdot CW_{3,1}$	$R_{18} + = x_9 \cdot CW_{2,1}$	$R_{17} + = x_9 \cdot CW_{1,1}$	$R_{16} + = x_9 \cdot CW_{0,1}$	R_{10}, R_{11}
x_{10} P_{28}, P_{29}	$R_{23} + = x_{10} \cdot CW_{7,2}$	$R_{22} + = x_{10} \cdot CW_{6,2}$	$R_{21} + = x_{10} \cdot CW_{5,2}$	$R_{20} + = x_{10} \cdot CW_{4,2}$	$R_{19} + = x_{10} \cdot CW_{3,2}$	$R_{18} + = x_{10} \cdot CW_{2,2}$	$R_{17} + = x_{10} \cdot CW_{1,2}$	$R_{16} + = x_{10} \cdot CW_{0,2}$	R_{12}, R_{13}
x_{11} P_{30}, P_{31}	$R_{23} + = x_{11} \cdot CW_{7,3}$	$R_{22} + = x_{11} \cdot CW_{6,3}$	$R_{21} + = x_{11} \cdot CW_{5,3}$	$R_{20} + = x_{11} \cdot CW_{4,3}$	$R_{19} + = x_{11} \cdot CW_{3,3}$	$R_{18} + = x_{11} \cdot CW_{2,3}$	$R_{17} + = x_{11} \cdot CW_{1,3}$	$R_{16} + = x_{11} \cdot CW_{0,3}$	R_{14}, R_{15}
x_{12} P_{32}, P_{33}	$R_{31} = x_{12} \cdot CW_{7,0} + P_{31}$	$R_{30} = x_{12} \cdot CW_{6,0} + P_{30}$	$R_{29} = x_{12} \cdot CW_{5,0} + P_{29}$	$R_{28} = x_{12} \cdot CW_{4,0} + P_{28}$	$R_{27} = x_{12} \cdot CW_{3,0} + P_{27}$	$R_{26} = x_{12} \cdot CW_{2,0} + P_{26}$	$R_{25} = x_{12} \cdot CW_{1,0} + P_{25}$	$R_{24} = x_{12} \cdot CW_{0,0} + P_{24}$	R_{16}, R_{17}

P_n is *single-precision* input partial-sum n

x_n is an *f16v4* input vector

$CW_{m,n}$ is the common weight state $\$CWEI$ m n

R_n is the final *single-precision* result of successive dot-product accumulations that began with P_n

³ 0 input used to fill AMP pipeline during warm-up period

enumFlags format:

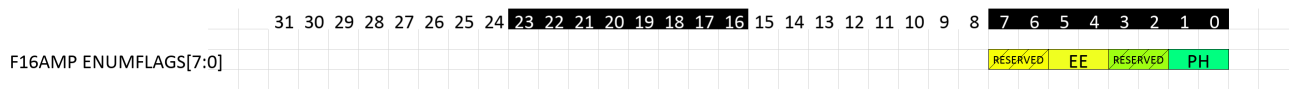


Fig. 3.15: f16v4sisoamp immediate format

8 output channels are processed/produced.

Table 3.129: *f16v4sisoamp* instruction definition

<i>f16v4sisoamp</i>	worker	aux
Syntax	f16v4sisoamp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; array<uint64_t,2> randomBits; vector<Single> out; array<vector<Single>,TFPU_AMP_UNITS_PER_SET> weights; array<Single,TFPU_AMP_UNITS_PER_SET> resultEven; array<Single,TFPU_AMP_UNITS_PER_SET> resultOdd; // Extract immediate config unsigned phase = F16AMP_ENUMFLAGS__PH__GET(op3); unsigned ampUnits = TFPU_AMP_UNITS_PER_SET; // Engine enables uint32_t ee = F16AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain if (phase == 0) { // Output from even accumulators out = { \$AACC[0], \$AACC[2] }; } else { // Output from odd accumulators out = { \$AACC[1], \$AACC[3] }; } vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; Except In // sNaN/INF input check uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight operands for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { uint64_t fullCwei = context.getCCCSState((k * 4) + phase); uint32_t *cwei = & fullCwei; weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[k].data(), weights[k].size()); } } </pre>	

Continued on next page

Table 3.129: *f16v4sisoamp* instruction definition (continued)

<i>f16v4sisoamp</i>	worker	aux
Syntax	f16v4sisoamp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	<pre> Compute // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { // 4-element dot-product resultEven[k] = TFPU_F16DotProduct(weights[k], inputs); Single incomingp; if (phase == 0) { // Even accumulators - // combine incoming partial-sum (currently stored in our // odd-accumulator) with dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2) + 1], resultEven[k], TFPU_FP32); // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * 2]; } } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs incomingp = op2[k & 1]; if (isnan(incomingp)) { incomingp = TFPU_F32_QNan(); } } } else { // Phase != 0 // Even accumulators - // accumulate dot-product result resultEven[k] = TFPU_Add(\$AACC[(k * 2)], resultEven[k], TFPU_FP32); // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled // Take the current value from its odd accumulator incomingp = \$AACC[((k + 2) * 2) + 1]; } } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs incomingp = op2[k & 1]; if (isnan(incomingp)) { incomingp = TFPU_F32_QNan(); } } } resultOdd[k] = incomingp; } </pre>	

Continued on next page

Table 3.129: *f16v4sisoamp* instruction definition (continued)

<i>f16v4sisoamp</i>		worker	aux
Syntax	f16v4sisoamp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCRdFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>		
Except Out			
Commit	<pre> // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[(k * 2)] = resultEven[k]; \$AACC[(k * 2) + 1] = resultOdd[k]; } } for (unsigned i = 0; i < 2; i++) { if (isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } \$aDst0:Dst0+1 = { TFPU_BitsFromF32(out[0]), TFPU_BitsFromF32(out[1]) }; </pre>		

Architectural state references: *\$AACC* , *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_GenSNanCheck* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCRdFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

f16v4f16/f32amp calculation (4 phases)

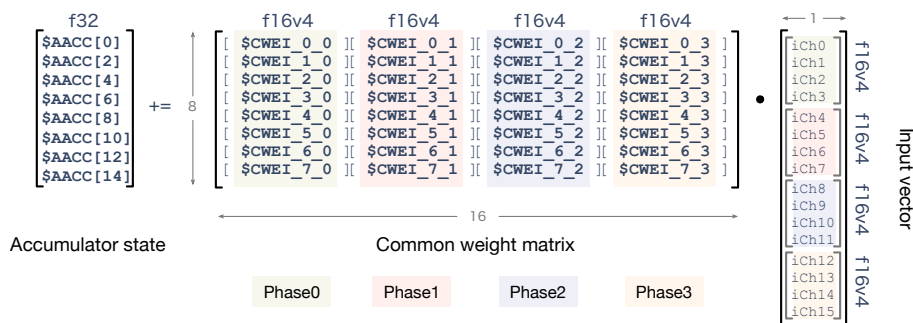


Fig. 3.16: f16v4sisoamp

f16v4f32amp convolution operation

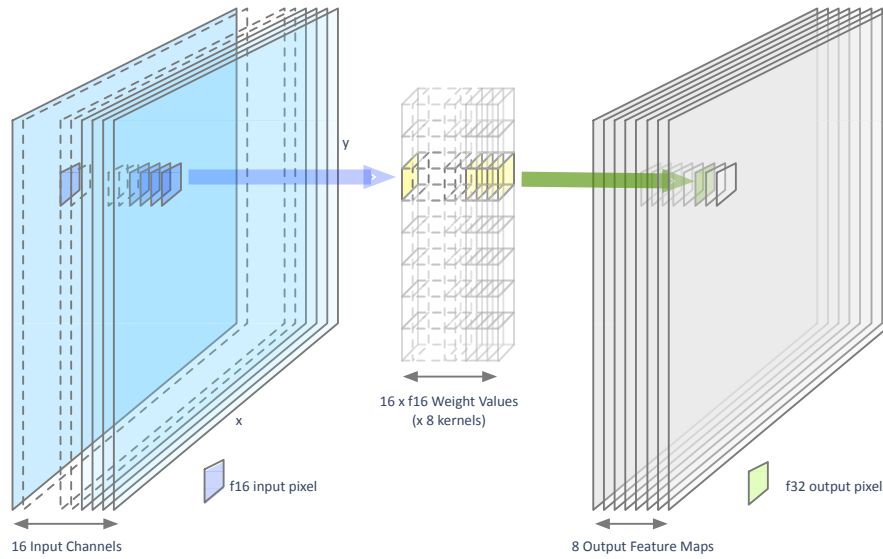


Fig. 3.17: f16v4sisoamp

Listing 3.6: f16v4sisoamp example

```

.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
_loop_start:
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+=$mzero, 0
  f16v4sisov2amp $outPartials, $inData, $inPartials, TAMP_F16V4_E4_P0
}
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+=$mzero, 0
  f16v4sisov2amp $outPartials, $inData, $inPartials, TAMP_F16V4_E4_P1
}
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+=$mzero, 0
  f16v4sisov2amp $outPartials, $inData, $inPartials, TAMP_F16V4_E4_P2
}
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+=$mzero, 0
  f16v4sisov2amp $outPartials, $inData, $inPartials, TAMP_F16V4_E4_P3
}
_loop_end:

```

3.7.3.3.30 f16v4sisolic

Half-precision floating point vector slim convolution. Input and result partial-sums are 2 x *single-precision* values.

Table 3.130: f16v4sisoslic, 2x1x3x4 example sequence

$\$aSrc0 I$	$\$AACC[14]$	$\$AACC[10]$	$\$AACC[6]$	$\$AACC[2]$	$\$AACC[12]$	$\$AACC[8]$	$\$AACC[4]$	$\$AACC[0]$	$\$aDst0$
$x_0 P_0,P_1$	-	$R_1 = x_0 \cdot CW_{5,0} + P_1$	-	-	-	$R_0 = x_0 \cdot CW_{4,0} + P_0$	-	-	-
$x_1 P_2,P_3$	-	$R_3 = x_1 \cdot CW_{5,0} + P_3$	$R_1 + = x_1 \cdot CW_{3,0}$	-	-	$R_2 = x_1 \cdot CW_{4,0} + P_2$	$R_0 + = x_1 \cdot CW_{2,0}$	-	-
$x_2 P_4,P_5$	-	$R_5 = x_2 \cdot CW_{5,0} + P_5$	$R_3 + = x_2 \cdot CW_{3,0}$	$R_1 + = x_2 \cdot CW_{1,0}$	-	$R_4 = x_2 \cdot CW_{4,0} + P_4$	$R_2 + = x_2 \cdot CW_{2,0}$	$R_0 + = x_2 \cdot CW_{0,0}$	-
$x_3 P_6,P_7$	-	$R_7 = x_3 \cdot CW_{5,0} + P_7$	$R_5 + = x_3 \cdot CW_{3,0}$	$R_3 + = x_3 \cdot CW_{1,0}$	-	$R_6 = x_3 \cdot CW_{4,0} + P_6$	$R_4 + = x_3 \cdot CW_{2,0}$	$R_2 + = x_3 \cdot CW_{0,0}$	R_0, R_1
$x_4 P_8,P_9$	-	$R_9 = x_4 \cdot CW_{5,0} + P_9$	$R_7 + = x_4 \cdot CW_{3,0}$	$R_5 + = x_4 \cdot CW_{1,0}$	-	$R_8 = x_4 \cdot CW_{4,0} + P_8$	$R_6 + = x_4 \cdot CW_{2,0}$	$R_4 + = x_4 \cdot CW_{0,0}$	R_2, R_3
$x_5 P_{10},P_{11}$	-	$R_{11} = x_5 \cdot CW_{5,0} + P_{11}$	$R_9 + = x_5 \cdot CW_{3,0}$	$R_7 + = x_5 \cdot CW_{1,0}$	-	$R_{10} = x_5 \cdot CW_{4,0} + P_{10}$	$R_8 + = x_5 \cdot CW_{2,0}$	$R_6 + = x_5 \cdot CW_{0,0}$	R_4, R_5
$x_6 P_{12},P_{13}$	-	$R_{13} = x_6 \cdot CW_{5,0} + P_{13}$	$R_{11} + = x_6 \cdot CW_{3,0}$	$R_9 + = x_6 \cdot CW_{1,0}$	-	$R_{12} = x_6 \cdot CW_{4,0} + P_{12}$	$R_{10} + = x_6 \cdot CW_{2,0}$	$R_8 + = x_6 \cdot CW_{0,0}$	R_6, R_7

Table 3.131: f16v4sisoslic, 2x1x4x4 example sequence

$\$aSrc0 I$	$\$AACC[14]$	$\$AACC[10]$	$\$AACC[6]$	$\$AACC[2]$	$\$AACC[12]$	$\$AACC[8]$	$\$AACC[4]$	$\$AACC[0]$	$\$aDst0$
$x_0 P_0,P_1$	$R_1 = x_0 \cdot CW_{7,0} + P_1$	-	-	-	$R_0 = x_0 \cdot CW_{6,0} + P_0$	-	-	-	-
$x_1 P_2,P_3$	$R_3 = x_1 \cdot CW_{7,0} + P_3$	$R_1 + = x_1 \cdot CW_{5,0}$	-	-	$R_2 = x_1 \cdot CW_{6,0} + P_2$	$R_0 + = x_1 \cdot CW_{4,0}$	-	-	-
$x_2 P_4,P_5$	$R_5 = x_2 \cdot CW_{7,0} + P_5$	$R_3 + = x_2 \cdot CW_{5,0}$	$R_1 + = x_2 \cdot CW_{3,0}$	-	$R_4 = x_2 \cdot CW_{6,0} + P_4$	$R_2 + = x_2 \cdot CW_{4,0}$	$R_0 + = x_2 \cdot CW_{2,0}$	-	-
$x_3 P_6,P_7$	$R_7 = x_3 \cdot CW_{7,0} + P_7$	$R_5 + = x_3 \cdot CW_{5,0}$	$R_3 + = x_3 \cdot CW_{3,0}$	$R_1 + = x_3 \cdot CW_{1,0}$	$R_6 = x_3 \cdot CW_{6,0} + P_6$	$R_4 + = x_3 \cdot CW_{4,0}$	$R_2 + = x_3 \cdot CW_{2,0}$	$R_0 + = x_3 \cdot CW_{0,0}$	-
$x_4 P_8,P_9$	$R_9 = x_4 \cdot CW_{7,0} + P_9$	$R_7 + = x_4 \cdot CW_{5,0}$	$R_5 + = x_4 \cdot CW_{3,0}$	$R_3 + = x_4 \cdot CW_{1,0}$	$R_8 = x_4 \cdot CW_{6,0} + P_8$	$R_6 + = x_4 \cdot CW_{4,0}$	$R_4 + = x_4 \cdot CW_{2,0}$	$R_2 + = x_4 \cdot CW_{0,0}$	R_0, R_1
$x_5 P_{10},P_{11}$	$R_{11} = x_5 \cdot CW_{7,0} + P_{11}$	$R_9 + = x_5 \cdot CW_{5,0}$	$R_7 + = x_5 \cdot CW_{3,0}$	$R_5 + = x_5 \cdot CW_{1,0}$	$R_{10} = x_5 \cdot CW_{6,0} + P_{10}$	$R_8 + = x_5 \cdot CW_{4,0}$	$R_6 + = x_5 \cdot CW_{2,0}$	$R_4 + = x_5 \cdot CW_{0,0}$	R_2, R_3
$x_6 P_{12},P_{13}$	$R_{13} = x_6 \cdot CW_{7,0} + P_{13}$	$R_{11} + = x_6 \cdot CW_{5,0}$	$R_9 + = x_6 \cdot CW_{3,0}$	$R_7 + = x_6 \cdot CW_{1,0}$	$R_{12} = x_6 \cdot CW_{6,0} + P_{12}$	$R_{10} + = x_6 \cdot CW_{4,0}$	$R_8 + = x_6 \cdot CW_{2,0}$	$R_6 + = x_6 \cdot CW_{0,0}$	R_4, R_5

P_n is *single-precision* input partial-sum n

x_n is an *f16v4* input vector

$CW_{m,n}$ is the common weight state $\$CWEI_m_n$

R_n is the final *single-precision* result of successive dot-product accumulations that began with P_n

enumFlags format:

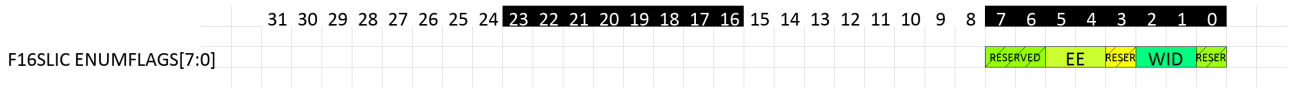


Fig. 3.18: f16v4sisoslic immediate format

When *enumFlags*.SE[0] is 0b0, 8 output channels are processed/produced. When *enumFlags*.SE[0] is 0b1, and the platform supports it, 16 output channels are processed/produced.

Table 3.132: *f16v4sisoslic* instruction definition

<i>f16v4sisoslic</i>	worker	aux
Syntax	f16v4sisoslic \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags	
Semantics	<pre> Prepare array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; array<uint64_t,2> randomBits; array<vector<Single>,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weights; array<Double,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; unsigned ampUnits = TFPU_AMP_UNITS_PER_SET; // Engine enables uint32_t ee = F16SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain // Output from even accumulators vector<Single> out = { \$AACC[0], \$AACC[2] }; vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3] }; Except In uint32_t fpExcpt = TFPU_GenSNanCheck(inputs.data(), inputs.size()); vector<Single> ops = { op2[0], op2[1] }; fpExcpt = TFPU_GenSNanCheck(ops.data(), ops.size()); // Weight set selection unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { uint64_t fullCwei = context.getCCCSState((u * 4) + wid); uint32_t *cwei = & fullCwei; weights[u] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1), pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) }; fpExcpt = TFPU_GenSNanCheck(weights[u].data(), weights[u].size()); } } Compute // Input partial-sum consumption for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { Single incomingp; if (engineEnable[engine + 1]) { // Engine behind me is enabled // use its current accumulator value incomingp = \$AACC[(u + 2) * 2]; } } } </pre>	

Continued on next page

Table 3.132: *f16v4sisoslic* instruction definition (continued)

<i>f16v4sisoslic</i>		worker	aux
Syntax	f16v4sisoslic \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1, enumFlags		
Semantics	Compute cont'd	<pre> } else { // Engines behind me are disabled (or I am the final engine // in the chain). Use the new partial-sum inputs incomingp = op2[u & 1]; if (isnan(incomingp)) { incomingp = TFPU_F32_QNan(); } } // 4-element dot-product result[u] = TFPU_F16DotProduct(weights[u], inputs); // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32); } } </pre>	
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> // Internal state updates for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[u * 2] = result[u]; } } for (unsigned i = 0; i < 2; i++) { if (isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } \$aDst0:Dst0+1 = { TFPU_BitsFromF32(out[0]), TFPU_BitsFromF32(out[1]) }; </pre>	

Architectural state references: *\$AACC* , *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_GenSNanCheck* , *TFPU_F32_QNan* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

f16v4f32slic convolution operation

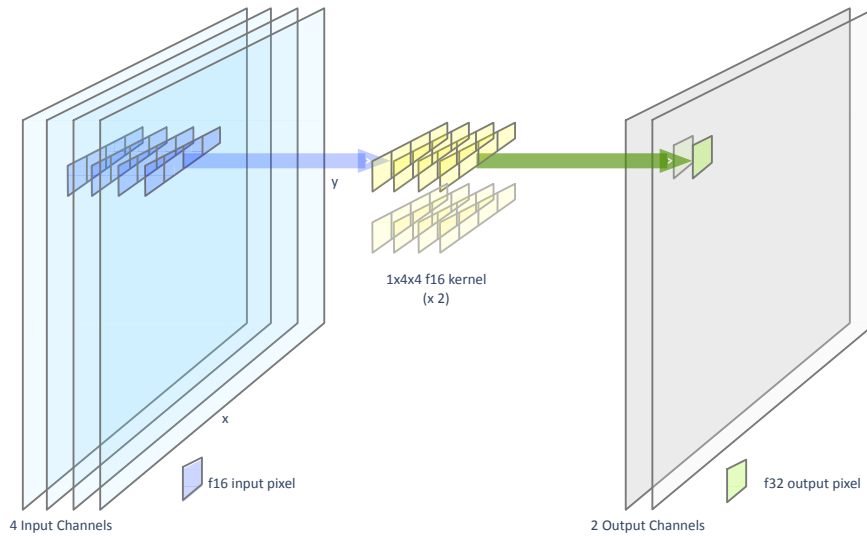


Fig. 3.19: f16v4sisoslic

f16v4sisoslic occurs in the following code examples:

- *f16v4sisoslic example part 1*
- *f16v4sisoslic example part 2*

Listing 3.7: f16v4sisoslic example part 1

```
// Create tri-packed address from input pointer, partial-sum input pointer
// plus partial-sum output pointer (partial sum output ptr typically lags behind
// partial-sum input ptr)
tapack      $triPtr, $inDataPtr, $inPartialsPtr, $outPartialsPtr

ld2x64pace  $inData, $inPartials, $triPtr+8, $mzero, 0
{
  ld2x64pace  $inData, $inPartials, $triPtr+8, $mzero, 0
  f16v4sisoslic $outPartials, $inData, $inPartials, TSLIC_F16V4_1x3_W0
}

.align 8
{
  rpt        $numRepeats, ((_loop_end - _loop_start) / 8) - 1;
  fnop
}
_loop_start:
{
  ld2xst64pace $inDataInPartials, $outPartials, $triPtr+8, $mzero, 0
  f16v4sisoslic $outPartials, $inData, $inPartials, TSLIC_F16V4_1x3_W0
}
_loop_end:

// Store final outputs
st64pace    $outPartials, $triPtr+8, $mzero, 0
```

Listing 3.8: f16v4sisoslic example part 2

```
// Create tri-packed address from input pointer, partial-sum input pointer
// plus partial-sum output pointer (partial sum output ptr typically lags behind
// partial-sum input ptr)
tapack      $triPtr, $inDataPtr, $inPartialsPtr, $outPartialsPtr

// Note that in this scenario, the latency of f16v4sisoslic is such that
// results must be held in the ARF for a tick in order to guarantee the
// avoidance of memory bank clash - hence the use of outPartialsA and
// outPartialsB
ld2x64pace  $inData, $inPartials, $triPtr+8, $mzero, 0
```

```

{
  ld2x64pace  $inData, $inPartials, $striPtr+=", $mzero, 0
  f16v4sisoslic $outPartialsA, $inData, $inPartials, TSLIC_F16V4_1x4_W0
}
{
  ld2x64pace  $inData, $inPartials, $striPtr+=", $mzero, 0
  f16v4sisoslic $outPartialsB, $inData, $inPartials, TSLIC_F16V4_1x4_W0
}

.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1;
  fnop
}
_loop_start:
{
  ld2xst64pace $inDataInPartials, $outPartialsA, $striPtr+=", $mzero, 0
  f16v4sisoslic $outPartialsA, $inData, $inPartials, TSLIC_F16V4_1x4_W0
}
{
  ld2xst64pace $inDataInPartials, $outPartialsB, $striPtr+=", $mzero, 0
  f16v4sisoslic $outPartialsB, $inData, $inPartials, TSLIC_F16V4_1x4_W0
}
_loop_end:

// Store final outputs
st64pace      $outPartialsA, $striPtr+=", $mzero, 0

```

3.7.3.3.31 f16v4stacc

Sort/shuffle (permute) through accumulators.

- Perform *\$AACC* state propagation as specified by the immediate.
- The destination register pair is written with 64-bits of result data from a combination of *\$AACC* registers. The precise combination is specified by the immediate.

Table 3.133: *f16v4stacc* instruction definition

<i>f16v4stacc</i>	worker	aux
Syntax	f16v4stacc \$aDst0:Dst0+1, enumFlags	
Semantics	<p>Prepare array<DataWord,2> op0; DataWord op1 = enumFlags;</p> <p>Commit int phase = op1 & 1;</p> <pre> if (phase == 0) { // Result op0[0] = \$AACC[0]; op0[1] = \$AACC[2]; // Propagate current \$AACC state (even to odd) \$AACC[1] = \$AACC[4]; \$AACC[5] = \$AACC[8]; \$AACC[9] = \$AACC[12]; \$AACC[3] = \$AACC[6]; \$AACC[7] = \$AACC[10]; \$AACC[11] = \$AACC[14]; } else { // Phase 1 // Result op0[0] = \$AACC[1]; op0[1] = \$AACC[3]; // Propagate current \$AACC state (odd chain) \$AACC[1] = \$AACC[5]; \$AACC[5] = \$AACC[9]; \$AACC[3] = \$AACC[7]; \$AACC[7] = \$AACC[11]; } \$aDst0:Dst0+1 = { op0[0], op0[1] }; </pre>	

Architectural state references: *\$AACC*

f16v4stacc occurs in the following code examples:

- *f16v4stacc* example

Listing 3.9: *f16v4stacc* example

```

// Setup input and output pointers
setzi    $inputPtr, inputData
setzi    $outputPtr, result

// Setup the stride values
ldconst $strides, STRIDE(-59) << 20 | STRIDE(-11) << 10 | STRIDE(4)

// Need to go round the entire outer loop 4 times
ldconst $counter, (4 - 1)

// Warm-up
ld64step $a0:1, $mzero, $inputPtr+=, 4
ld64step $a2:3, $mzero, $inputPtr+=, 4
ld64step $a4:5, $mzero, $inputPtr+=, 4

{
  ld64step $a6:7, $mzero, $inputPtr+=, -11
  f16v4istacc $zeros, $a0:1, $a2:3, TISTACC_P0
}
{
  ld64step $a0:1, $mzero, $inputPtr+=, 4
  f16v4istacc $zeros, $a4:5, $a6:7, TISTACC_P1
}
{
  ld64step $a2:3, $mzero, $inputPtr+=, 4
  f16v4stacc $a6:7, TSTACC_P0
}

```

```

}

tapack $striPtr, $inputPtr, $mzero, $outputPtr

_outer_loop_start:

.align 8
{
  rpt          3, ((_inner_loop_end - _inner_loop_start) / 8) - 1
  fnop
}
_inner_loop_start:
{
  ldst64pace   $a4:5, $a6:7, $striPtr+=", $strides, INC_ST4_LD4
  f16v4stacc   $a6:7, TSTACC_P1
}
{
  ldst64pace   $a6:7, $a6:7, $striPtr+=", $strides, INC_ST4_LDM11
  f16v4istacc  $a2:3, $a0:1, $a2:3, TISTACC_P0
}
{
  ldst64pace   $a0:1, $a2:3, $striPtr+=", $strides, INC_ST4_LD4
  f16v4istacc  $a2:3, $a4:5, $a6:7, TISTACC_P1
}
{
  ldst64pace   $a2:3, $a2:3, $striPtr+=", $strides, INC_ST4_LD4
  f16v4stacc   $a6:7, TSTACC_P0
}
_inner_loop_end:

{
  ldst64pace   $a4:5, $a6:7, $striPtr+=", $strides, INC_ST4_LD4
  f16v4stacc   $a6:7, TSTACC_P1
}
{
  ldst64pace   $a6:7, $a6:7, $striPtr+=", $strides, INC_ST4_LD1
  f16v4istacc  $a2:3, $a0:1, $a2:3, TISTACC_P0
}
{
  ldst64pace   $a0:1, $a2:3, $striPtr+=", $strides, INC_ST4_LD4
  f16v4istacc  $a2:3, $a4:5, $a6:7, TISTACC_P1
}
{
  ldst64pace   $a2:3, $a2:3, $striPtr+=", $strides, INC_STM59_LD4
  f16v4stacc   $a6:7, TSTACC_P0
}

brnzdec      $counter, _outer_loop_start

_outer_loop_end:

```

3.7.3.3.32 f16v4sub

Half-precision floating-point 4-element vector subtraction

Table 3.134: *f16v4sub* instruction definition

<i>f16v4sub</i>		worker	aux
Syntax	f16v4sub \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4sub \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4sub \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1		
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[0], 1), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 0), pickHalf(<(\$aSrc0:Src0+1, \$aSrc0:BL, \$aSrc0:BU)>[1], 1) }; array<Half,4> op2 = { pickHalf(\$aSrc1:Src1+1[0], 0), pickHalf(\$aSrc1:Src1+1[0], 1), pickHalf(\$aSrc1:Src1+1[1], 0), pickHalf(\$aSrc1:Src1+1[1], 1) }; bool nanoo = \$FP_CTL.NANOO; bool enableStochasticRounding; vector<Single> result; array<bool,4> specialCase; array<uint64_t,2> randomBits; if (\$FP_CTL.ESR) { enableStochasticRounding = true; } else { enableStochasticRounding = false; } </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(4); for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); } </pre>	
	Compute	<pre> for (i = 0; i < 4; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) - static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, TFPU_ROUND_EVEN); } } </pre>	
	Except Out	<pre> if (enableStochasticRounding) { TFPU_ApplyStochasticRound(randomBits, result); } for (i = 0; i < 4; ++i) { fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo); } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	

Continued on next page

Table 3.134: *f16v4sub* instruction definition (continued)

<i>f16v4sub</i>		worker	aux
Syntax	f16v4sub \$aDst0:Dst0+1, \$aSrc0:BL, \$aSrc1:Src1+1 f16v4sub \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f16v4sub \$aDst0:Dst0+1, \$aSrc0:BU, \$aSrc1:Src1+1		
Semantics	Commit	<pre> HalfSaturationMode smode = getNanooMode(nanoo); \$aDst0:Dst0+1 = { Half(result[0]).bitz32(smode) (Half(result[1]).bitz32(smode) << 16), Half(result[2]).bitz32(smode) (Half(result[3]).bitz32(smode) << 16) }; </pre>	

Architectural state references: *\$FP_CTL*, *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*, *\$FP_STS*

Function references: *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_ApplyStochasticRound*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*

3.7.3.3.33 f16v4sum

Half-precision 4-element vector 2x2 lateral summation to 2-element *single-precision* vector.

Table 3.135: *f16v4sum* instruction definition

<i>f16v4sum</i>		worker	aux
Syntax	f16v4sum \$aDst0:Dst0+1, \$aSrc0:Src0+1		
Semantics	Prepare	<pre> array<Half,4> op1 = { pickHalf(\$aSrc0:Src0+1[0], 0), pickHalf(\$aSrc0:Src0+1[0], 1), pickHalf(\$aSrc0:Src0+1[1], 0), pickHalf(\$aSrc0:Src0+1[1], 1) }; array<bool,2> specialCase; array<Single,2> result; </pre>	
	Except	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; specialCase[0] = TFPU_DoAddPreExecute(op1[0], op1[1], &fpExcpt, &result[0]); specialCase[1] = TFPU_DoAddPreExecute(op1[2], op1[3], &fpExcpt, &result[1]); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	In		
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { Double res = op1[2*i] + op1[(2*i)+1]; result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); } } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_DoAddPreExecute*, *TFPU_IsMalign*, *TFPU_RoundFP64ToFmt*, *TFPU_BitsFromF32*

3.7.3.4 f16 8-element vector

3.7.3.4.1 f16v8absacc

Half-precision 8-element vector accumulation of absolute values to *single-precision*.

Table 3.136: *f16v8absacc* instruction definition

<i>f16v8absacc</i>	worker	aux
Syntax	f16v8absacc \$aSrc0:Src0+3	
Semantics	Prepare	<pre>array<Half,8> op0 = { pickHalf(\$aSrc0:Src0+3[0], 0), pickHalf(\$aSrc0:Src0+3[0], 1), pickHalf(\$aSrc0:Src0+3[1], 0), pickHalf(\$aSrc0:Src0+3[1], 1), pickHalf(\$aSrc0:Src0+3[2], 0), pickHalf(\$aSrc0:Src0+3[2], 1), pickHalf(\$aSrc0:Src0+3[3], 0), pickHalf(\$aSrc0:Src0+3[3], 1) }; array<Single,8> result;</pre>
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 8; ++i) { if (op0[i].isNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>
	Compute	<pre>for (i = 0; i < 8; ++i) { result[i] = TFPU_Add(op0[i], \$AACC[i * 2], TFPU_FP32, TFPU_ABS); }</pre>
	Commit	<pre>for (i = 0; i < 8; ++i) { \$AACC[i * 2] = result[i]; }</pre>

Architectural state references: *\$FP_STS*, *\$FP_CTL*, *\$AACC*

Function references: *TFPU_IsMalign*, *TFPU_Add*

3.7.3.4.2 f16v8acc

Half-precision 8-element vector accumulation to *single-precision*.

Table 3.137: *f16v8acc* instruction definition

<i>f16v8acc</i>		worker	aux
Syntax	f16v8acc \$aSrc0:Src0+3		
Semantics	Prepare	<pre> array<Half,8> op0 = { pickHalf(\$aSrc0:Src0+3[0], 0), pickHalf(\$aSrc0:Src0+3[0], 1), pickHalf(\$aSrc0:Src0+3[1], 0), pickHalf(\$aSrc0:Src0+3[1], 1), pickHalf(\$aSrc0:Src0+3[2], 0), pickHalf(\$aSrc0:Src0+3[2], 1), pickHalf(\$aSrc0:Src0+3[3], 0), pickHalf(\$aSrc0:Src0+3[3], 1) }; array<Single,8> result; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 8; ++i) { if (op0[i].isNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> for (i = 0; i < 8; ++i) { result[i] = TFPU_Add(op0[i], \$AACC[i * 2], TFPU_FP32); } </pre>	
	Commit	<pre> for (i = 0; i < 8; ++i) { \$AACC[i * 2] = result[i]; } </pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_IsMalign* , *TFPU_Add*

3.7.3.4.3 f16v8sqacc

Half-precision 8-element vector accumulation of squares to *single-precision*.

Table 3.138: *f16v8sqacc* instruction definition

<i>f16v8sqacc</i>		worker	aux
Syntax	f16v8sqacc \$aSrc0:Src0+3		
Semantics	Prepare	<pre> array<Half,8> op0 = { pickHalf(\$aSrc0:Src0+3[0], 0), pickHalf(\$aSrc0:Src0+3[0], 1), pickHalf(\$aSrc0:Src0+3[1], 0), pickHalf(\$aSrc0:Src0+3[1], 1), pickHalf(\$aSrc0:Src0+3[2], 0), pickHalf(\$aSrc0:Src0+3[2], 1), pickHalf(\$aSrc0:Src0+3[3], 0), pickHalf(\$aSrc0:Src0+3[3], 1) }; array<Single,8> result; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 8; ++i) { if (op0[i].isNaN()) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 8; ++i) { result[i] = TFPU_Mac(op0[i], op0[i], \$AACC[i * 2], TFPU_FP32, rmode); } </pre>	
	Commit	<pre> for (i = 0; i < 8; ++i) { \$AACC[i * 2] = result[i]; } </pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_IsMalign* , *TFPU_Mac*

3.7.3.5 f32 2-element vector

3.7.3.5.1 f32v2absadd

Single-precision 2-element vector element-wise addition of absolute values.

Table 3.139: *f32v2absadd* instruction definition

<i>f32v2absadd</i>		worker	aux
Syntax	<code>f32v2absadd \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; array<bool,2> specialCase; </pre>	
	Except In	<pre> for (i = 0; i < 2; ++i) { op1[i] = fabs(op1[i]); op2[i] = fabs(op2[i]); } uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NAN00); } } </pre>	
	Except Out	<pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_DoAddPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.5.2 *f32v2absmax*

Single-precision 2-element vector element-wise max of absolute values.

Table 3.140: *f32v2absmax* instruction definition

<i>f32v2absmax</i>		worker	aux
Syntax	<i>f32v2absmax</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc1:Src1+1</i> , <i>\$aSrc0:Src0+1</i>		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsSNan(op2[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { result[i] = TFPU_Max(fabs(op1[i]), fabs(op2[i])); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_IsMalign*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.5.3 *f32v2add*

Single-precision floating-point vector add

Table 3.141: *f32v2add* instruction definition

<i>f32v2add</i>		worker	aux
Syntax	f32v2add \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2add \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; array<bool,2> specialCase; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NAN00); } } </pre>	
	Except Out	<pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.5.4 f32v2aop

Single-precision 2-element vector outer product with accumulation to *single-precision*.

Table 3.142: *f32v2aop* instruction definition

<i>f32v2aop</i>	worker	aux
Syntax	<i>f32v2aop</i> <i>\$aSrc0:Src0+1</i> , <i>\$aSrc1:Src1+1</i> , <i>enumFlags</i>	
Semantics	<pre> Prepare array<Single,2> op0 = { TFPU_F32FromBits(<i>\$aSrc0:Src0+1[0]</i>), TFPU_F32FromBits(<i>\$aSrc0:Src0+1[1]</i>) }; array<Single,2> op1 = { TFPU_F32FromBits(<i>\$aSrc1:Src1+1[0]</i>), TFPU_F32FromBits(<i>\$aSrc1:Src1+1[1]</i>) }; DataWord op2 = enumFlags; array<bool,4> specialCase; array<Single,4> result; Except In uint32_t fpExcpt = TFPEXCPT_NONE; specialCase[0] = TFPU_DoMacPreExecute(op0[0], op1[0], <i>\$AACC[0]</i>, &fpExcpt, &result[0]); specialCase[1] = TFPU_DoMacPreExecute(op0[1], op1[0], <i>\$AACC[2]</i>, &fpExcpt, &result[1]); specialCase[2] = TFPU_DoMacPreExecute(op0[0], op1[1], <i>\$AACC[4]</i>, &fpExcpt, &result[2]); specialCase[3] = TFPU_DoMacPreExecute(op0[1], op1[1], <i>\$AACC[6]</i>, &fpExcpt, &result[3]); <i>\$FP_STS</i> = <i>\$FP_STS</i> fpExcpt; if (TFPU_IsMalign(fpExcpt, <i>\$FP_CTL</i>)) { EXCEPT(TEXCPT_FP); } Compute TileRoundMode_t rmode = <i>\$FP_CTL</i>.RND; // MAC isn't "fused" in the sense that Tile rounds to the accumulator // format prior to the addition. if (!specialCase[0]) { Single mRes = TFPU_Mul(op0[0], op1[0], TFPU_FP32, rmode); result[0] = TFPU_Add(mRes, <i>\$AACC[0]</i>, TFPU_FP32); } if (!specialCase[1]) { Single mRes = TFPU_Mul(op0[1], op1[0], TFPU_FP32, rmode); result[1] = TFPU_Add(mRes, <i>\$AACC[2]</i>, TFPU_FP32); } if (!specialCase[2]) { Single mRes = TFPU_Mul(op0[0], op1[1], TFPU_FP32, rmode); result[2] = TFPU_Add(mRes, <i>\$AACC[4]</i>, TFPU_FP32); } if (!specialCase[3]) { Single mRes = TFPU_Mul(op0[1], op1[1], TFPU_FP32, rmode); result[3] = TFPU_Add(mRes, <i>\$AACC[6]</i>, TFPU_FP32); } Commit <i>\$AACC[0]</i> = result[0]; <i>\$AACC[2]</i> = result[1]; <i>\$AACC[4]</i> = result[2]; <i>\$AACC[6]</i> = result[3]; </pre>	

Architectural state references: *\$AACC* , *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_DoMacPreExecute* , *TFPU_IsMalign* , *TFPU_Mul* , *TFPU_Add*

$$\begin{array}{c}
 \text{f32} \quad \text{f32} \quad \text{f32} \quad \text{f32} \\
 \begin{bmatrix} \$\text{AACC}[0] & \$\text{AACC}[4] \\ \$\text{AACC}[2] & \$\text{AACC}[6] \end{bmatrix} += \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \begin{bmatrix} y_0 & y_1 \end{bmatrix} \\
 \text{\$aSrc0} \quad \text{\$aSrc1}
 \end{array}$$

Fig. 3.20: f32v2aop

3.7.3.5.5 f32v2axpy

Single-precision 2-element vector $\mathbf{z} = a\mathbf{x} + \mathbf{y}$ The scalar multiplicand a is provided by the internal state element $\$TAS$.

Results are stored within the accumulator state. Destination registers are written with the previous accumulator state.

Table 3.143: *f32v2axpy* instruction definition

<i>f32v2axpy</i>	worker	aux
Syntax	<code>f32v2axpy \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>	
Semantics	<pre> Prepare array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> result; array<bool,2> specialCase; vector<Single> out = { \$AACC[0], \$AACC[2] }; Single a = TFPU_F32FromBits(\$TAS); Except In uint32_t fpExcpt = TFPEXCPT_NONE; // Input exception check for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAxpbyPreExecute(a, op2[i], 1.0, op1[i], &fpExcpt, &result[i]); } Compute TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { Double res = a * op2[i]; Single mRes = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode); result[i] = TFPU_Add(mRes, op1[i], TFPU_FP32); } // Output processing if (isnan(out[i])) { out[i] = TFPU_F32_QNan(); } } Except Out // Output exception check fpExcpt = TFPU_AACCRdFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit \$AACC[0] = result[0]; \$AACC[2] = result[1]; \$aDst0:Dst0+1 = { TFPU_BitsFromF32(out[0]), TFPU_BitsFromF32(out[1]) }; </pre>	

Architectural state references: *\$AACC* , *\$TAS* , *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoAxpbyPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCRdFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.5.6 *f32v2clamp*

Single-precision floating-point vector min-of-maximum

Table 3.144: *f32v2clamp* instruction definition

<i>f32v2clamp</i>	worker	aux
Syntax	<code>f32v2clamp \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1</code>	
Semantics	<pre> Prepare array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; Except In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsSNan(op2[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute Single lower = op2[0]; Single upper = op2[1]; // Unlike min/max, clamp explicitly propagates (regenerates) // NaN inputs if (isnan(lower) isnan(upper)) { for (i = 0; i < 2; i++) { result[i] = TFPU_F32_QNan(); } } else { for (i = 0; i < 2; i++) { if (isnan(op1[i])) { result[i] = TFPU_F32_QNan(); } else if (op1[i] > upper) { result[i] = upper; } else { result[i] = TFPU_Max(op1[i], lower); } } } Commit \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_IsMalign*, *TFPU_F32_QNan*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.5.7 f32v2class

Single-precision floating-point vector classifier. IEEE 754-2008: 5.7.2

Table 3.145: *f32v2class* instruction definition

<i>f32v2class</i>		worker	aux
Syntax	<i>f32v2class</i> <i>\$aDst0</i> , <i>\$aSrc0:Src0+1</i>		
Semantics	Prepare	<pre>array<QuarterDataWord,4> op0; array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) };</pre>	
	Compute	<pre>op0 = { 0, 0, 0, 0 }; // Denorms will have been flushed to zero for (i = 0; i < 2; i++) { DataWord c1ss; Single s = op1[i]; bool sign = signbit(s); if (TFPU_F32_IsSNan(s)) { c1ss = TFPU_CLASS_SNaN; } else if (TFPU_F32_IsQNaN(s)) { c1ss = TFPU_CLASS_QNaN; } else if (isinf(s)) { c1ss = (sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF); } else if (fabs(s) == 0.0) { c1ss = (sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO); } else { c1ss = (sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM); } op0[i] = c1ss; }</pre>	
	Commit	<pre>\$aDst0 = { ((op0[0] & 0xff) << 0) ((op0[1] & 0xff) << 8) ((op0[2] & 0xff) << 16) ((op0[3] & 0xff) << 24) };</pre>	

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_F32_IsQNaN*

3.7.3.5.8 *f32v2cmpeq*

Single-precision floating-point vector equality test

Table 3.146: *f32v2cmpeq* instruction definition

<i>f32v2cmpeq</i>	worker	aux
Syntax	<pre>f32v2cmpeq \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmpeq \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1</pre>	
Semantics	<pre>Prepare array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute for (i = 0; i < 2; i++) { result[i] = (r1[i] == TFPU_RELATION_EQ); } Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.5.9 *f32v2cmpge*

Single-precision floating-point vector greater-than or equal-to test

Table 3.147: *f32v2cmpge* instruction definition

<i>f32v2cmpge</i>		worker	aux
Syntax	f32v2cmpge \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmpge \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); } </pre>	
	Except	<pre> // IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> for (i = 0; i < 2; i++) { result[i] = ((r1[i] == TFPU_RELATION_EQ) (r1[i] == TFPU_RELATION_GT)); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.5.10 *f32v2cmpgt*

Single-precision floating-point vector greater-than test

Table 3.148: *f32v2cmpgt* instruction definition

<i>f32v2cmpgt</i>	worker	aux
Syntax	<pre>f32v2cmpgt \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmpgt \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1</pre>	
Semantics	<pre>Prepare array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); } Except // IEEE 754-2008: 5.11, 7.2 In uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Compute for (i = 0; i < 2; i++) { result[i] = (r1[i] == TFPU_RELATION_GT); } Commit \$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.5.11 *f32v2cmple*

Single-precision floating-point vector less-than or equal-to test

Table 3.149: *f32v2cmple* instruction definition

<i>f32v2cmple</i>		worker	aux
Syntax	<pre>f32v2cmple \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmple \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1</pre>		
Semantics	Prepare	<pre>array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); }</pre>	
	Except	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	In		
	Compute	<pre>for (i = 0; i < 2; i++) { result[i] = ((r1[i] == TFPU_RELATION_EQ) (r1[i] == TFPU_RELATION_LT)); }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.5.12 *f32v2cmplt*

Single-precision floating-point vector less-than test

Table 3.150: *f32v2cmplt* instruction definition

<i>f32v2cmplt</i>		worker	aux
Syntax	<pre>f32v2cmplt \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmplt \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1</pre>		
Semantics	Prepare	<pre>array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); }</pre>	
	Except	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>for (i = 0; i < 2; i++) { result[i] = (r1[i] == TFPU_RELATION_LT); }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) };</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.5.13 *f32v2cmpne*

Single-precision floating-point vector inequality test

Table 3.151: *f32v2cmpne* instruction definition

<i>f32v2cmpne</i>		worker	aux
Syntax	<pre>f32v2cmpne \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2cmpne \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1</pre>		
Semantics	Prepare	<pre>array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<TileFPRelation_t,2> r1; array<bool,2> result; for (i = 0; i < 2; i++) { r1[i] = TFPU_Relation(op1[i], op2[i]); }</pre>	
	Except In	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { if (r1[i] == TFPU_RELATION_UN) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>for (i = 0; i < 2; i++) { result[i] = (r1[i] != TFPU_RELATION_EQ); }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { (result[0] ? 0xffffffff : 0x00000000), (result[1] ? 0xffffffff : 0x00000000) };</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.5.14 *f32v2gina*

Get and initialise accumulators.

- Read a pair of internal accumulator values as *single-precision* values.
- Write 2-element vector of *single-precision* input values to internal accumulator state.
- The instruction immediate specifies which pair of accumulator registers are to be read and written:
 1. Read *\$AACC[0]* and *\$AACC[2]*, write *\$AACC[12]* and *\$AACC[14]*
 2. Read *\$AACC[1]* and *\$AACC[3]*, write *\$AACC[13]* and *\$AACC[15]*
and if and only if the platform supports 2 AMP sets:
 3. Read *\$AACC[16]* and *\$AACC[18]*, write *\$AACC[28]* and *\$AACC[30]*
 4. Read *\$AACC[17]* and *\$AACC[19]*, write *\$AACC[29]* and *\$AACC[31]*
- Propagate internal accumulator state such that all accumulator registers may be read and written via a sequence of this instruction.

zimm12 immediate format:

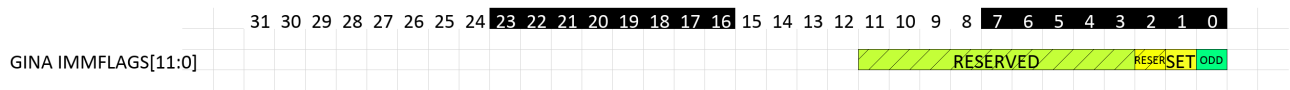


Fig. 3.21: f32v2gina immediate format

Table 3.152: *f32v2gina* instruction definition

<i>f32v2gina</i>	worker	aux
Syntax	f32v2gina \$aDst0:Dst0+1, \$aSrc0:Src0+1, zimm12	
Semantics	<pre> Prepare array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; DataWord op2 = zimm12; vector<Single> result; unsigned set = GINA_IMMFLAGS__SET__GET(op2); // base accumulator Id for set unsigned b = set * TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; uint32_t fpExcpt = TFPEXCPT_NONE; // Input exception check if (TFPU_F32_IsSNan(op1[0]) TFPU_F32_IsSNan(op1[1])) { fpExcpt = TFPEXCPT_INV; } Compute if (GINA_IMMFLAGS__ODD__GET(op2) == 0) { result = { \$AACC[b+0], \$AACC[b+2] }; } else { result = { \$AACC[b+1], \$AACC[b+3] }; } Except Out // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(result, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit if (GINA_IMMFLAGS__ODD__GET(op2) == 0) { // Propagate internal accumulator state \$AACC[b+0] = \$AACC[b+4]; \$AACC[b+4] = \$AACC[b+8]; \$AACC[b+8] = \$AACC[b+12]; \$AACC[b+2] = \$AACC[b+6]; \$AACC[b+6] = \$AACC[b+10]; \$AACC[b+10] = \$AACC[b+14]; // Commit 2 input values \$AACC[b+12] = isnan(op1[0]) ? TFPU_F32_QuietenNan(op1[0]) : op1[0]; \$AACC[b+14] = isnan(op1[1]) ? TFPU_F32_QuietenNan(op1[1]) : op1[1]; } else { // Propagate internal accumulator state \$AACC[b+1] = \$AACC[b+5]; \$AACC[b+5] = \$AACC[b+9]; \$AACC[b+9] = \$AACC[b+13]; \$AACC[b+3] = \$AACC[b+7]; \$AACC[b+7] = \$AACC[b+11]; \$AACC[b+11] = \$AACC[b+15]; // Commit 2 input values \$AACC[b+13] = isnan(op1[0]) ? TFPU_F32_QuietenNan(op1[0]) : op1[0]; \$AACC[b+15] = isnan(op1[1]) ? TFPU_F32_QuietenNan(op1[1]) : op1[1]; } </pre>	

Continued on next page

Table 3.152: *f32v2gina* instruction definition (continued)

<i>f32v2gina</i>		worker	aux
Syntax	f32v2gina \$aDst0:Dst0+1, \$aSrc0:Src0+1, zimm12		
Semantics	Commit cont'd	<pre> if (isnan(result[0])) { result[0] = TFPU_F32_QNan(); } if (isnan(result[1])) { result[1] = TFPU_F32_QNan(); } \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$AACC*, *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QuietenNan*, *TFPU_F32_QNan*, *TFPU_BitsFromF32*

f32v2gina occurs in the following code examples:

- *f16v4cmac example*
- *ldb16b16 example*
- *f32mac example*

3.7.3.5.15 *f32v2grand*

Gaussian distribution, 2-element *single-precision* random vector

Table 3.153: *f32v2grand* instruction definition

<i>f32v2grand</i>		worker	aux
Syntax	f32v2grand \$aDst0:Dst0+1		
Semantics	Prepare	array<Single,2> result;	
	Compute	<pre> array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); // Calculate overall summation of outputs from successive // applications of xoroshiro128aox int16_t rsum[2] = { 0, 0 }; for (i = 0; i < 12; i++) { rsum[0] += ((randomBits[0] >> (i * 5)) & 0x1f); rsum[1] += ((randomBits[1] >> (i * 5)) & 0x1f); } result[0] = rsum[0] - 186 / 32.0; result[1] = rsum[1] - 186 / 32.0; </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*

Function references: *TFPU_BitsFromF32*

3.7.3.5.16 f32v2mac

Single-precision floating point vector multiply and 32-bit accumulate

Table 3.154: *f32v2mac* instruction definition

<i>f32v2mac</i>	worker	aux
Syntax	f32v2mac \$aSrc0:Src0+1, \$aSrc1:Src1+1	
Semantics	<pre> array<Single,2> op0 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<bool,2> specialCase; array<Single,2> result; </pre>	
Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; specialCase[0] = TFPU_DoMacPreExecute(op0[0], op1[0], \$AACC[0], &fpExcpt, &result[0]); specialCase[1] = TFPU_DoMacPreExecute(op0[1], op1[1], \$AACC[2], &fpExcpt, &result[1]); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
Compute	<pre> TileRoundMode_t rmode = \$FP_CTL.RND; // MAC isn't "fused" in the sense that Tile rounds to the accumulator // format prior to the addition. if (!specialCase[0]) { Single mRes = TFPU_Mul(op0[0], op1[0], TFPU_FP32, rmode); result[0] = TFPU_Add(mRes, \$AACC[0], TFPU_FP32); } if (!specialCase[1]) { Single mRes = TFPU_Mul(op0[1], op1[1], TFPU_FP32, rmode); result[1] = TFPU_Add(mRes, \$AACC[2], TFPU_FP32); } </pre>	
Commit	<pre> \$AACC[0] = result[0]; \$AACC[2] = result[1]; </pre>	

Architectural state references: *\$AACC* , *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_DoMacPreExecute* , *TFPU_IsMalign* , *TFPU_Mul* , *TFPU_Add*

3.7.3.5.17 f32v2max

Single-precision floating-point vector element-wise max

Table 3.155: *f32v2max* instruction definition

<i>f32v2max</i>		worker	aux
Syntax	f32v2max \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; </pre>	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsSNan(op2[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { result[i] = TFPU_Max(op1[i], op2[i]); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_IsMalign*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.5.18 f32v2min

Single-precision 2-element vector element-wise minimum

Table 3.156: *f32v2min* instruction definition

<i>f32v2min</i>		worker	aux
Syntax	f32v2min \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1		
Semantics	Prepare	<pre>array<Single,2> op1 = { TFPU_F32FromBits(\$aSrc0:Src0+1[0]), TFPU_F32FromBits(\$aSrc0:Src0+1[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { if (TFPU_F32_IsSNan(op1[i]) TFPU_F32_IsSNan(op2[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>for (i = 0; i < 2; ++i) { result[i] = TFPU_Min(op1[i], op2[i]); }</pre>	
	Commit	<pre>\$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) };</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_IsMalign*, *TFPU_Min*, *TFPU_BitsFromF32*

3.7.3.5.19 f32v2mul

Single-precision floating-point 2 element vector, Hadamard product

Table 3.157: *f32v2mul* instruction definition

<i>f32v2mul</i>	worker	aux
Syntax	f32v2mul \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2mul \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1	
Semantics	Prepare <pre> array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; array<bool,2> specialCase; </pre> Except In <pre> uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; i++) { specialCase[i] = TFPU_DoMulPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } </pre> Compute <pre> TileRoundMode_t rmode = \$FP_CTL.RND; for (i = 0; i < 2; ++i) { if (!specialCase[i]) { Double res = op1[i] * op2[i]; fpExcpt = TFPU_GenOFLOCheck(res, TFPU_FP32, TFPU_NO_NAN00); result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode); } } </pre> Except Out <pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre> Commit <pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoMulPreExecute* , *TFPU_GenOFLOCheck* , *TFPU_RoundFP64ToFmt* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.5.20 *f32v2rmask*

Single-precision floating-point vector random mask.

The result is a masked version of the input vector, with each element of the input being individually masked with the probability specified by the bottom 17-bits of the 2nd input operand:

- if $\$aSrc1[16] == 1$, no masking is applied (the result is a copy of the input vector)
- else if $\$aSrc1[16:0] == 0$, the result is a zero vector
- otherwise each element is individually unmasked with probability $\frac{\$aSrc1[15:0]}{65536}$

PRNG is used by this instruction to generate 2 x 16-bit random values from the discrete uniform distribution.

Table 3.158: *f32v2rmask* instruction definition

<i>f32v2rmask</i>		worker	aux
Syntax	<i>f32v2rmask</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0:Src0+1</i> , <i>\$aSrc1</i>		
Semantics	Prepare	<pre> array<DataWord,2> op0; array<DataWord,2> op1 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; DataWord op2 = \$aSrc1; </pre>	
	Compute	<pre> bool always = ((op2 >> 16) & 1) == 1; bool never = (op2 & 0x1ffff) == 0; array<uint64_t,2> randomBits; TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits); op0 = { 0, 0 }; if (always) { // No masking required op0[0] = op1[0]; op0[1] = op1[1]; } else if (!never) { uint16_t probab = op2 & 0xffff; // Mask out the 32-bit elements based on the random bit-patterns and probab if (((randomBits[0] >> 0) & 0xffff) < probab) { op0[0] = op1[0]; } if (((randomBits[0] >> 32) & 0xffff) < probab) { op0[1] = op1[1]; } } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { op0[0], op0[1] }; </pre>	

Architectural state references: *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*

3.7.3.5.21 *f32v2sub*

Single-precision floating-point vector subtraction

Table 3.159: *f32v2sub* instruction definition

<i>f32v2sub</i>		worker	aux
Syntax	f32v2sub \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$aSrc1:Src1+1 f32v2sub \$aDst0:Dst0+1, \$aSrc0:B, \$aSrc1:Src1+1		
Semantics	Prepare	<pre> array<Single,2> op1 = { TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[0]), TFPU_F32FromBits(<(\$aSrc0:Src0+1, \$aSrc0:B)>[1]) }; array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; array<Single,2> result; array<bool,2> specialCase; </pre>	
	Except In	<pre> for (i = 0; i < 2; ++i) { op2[i] = -op2[i]; } uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 2; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } </pre>	
	Compute	<pre> for (i = 0; i < 2; ++i) { if (!specialCase[i]) { double res = static_cast<double>(op1[i]) + static_cast<double>(op2[i]); result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NAN00); } } </pre>	
	Except Out	<pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> \$aDst0:Dst0+1 = { TFPU_BitsFromF32(result[0]), TFPU_BitsFromF32(result[1]) }; </pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.6 f32 4-element vector

3.7.3.6.1 f32v4absacc

Single-precision 4-element vector accumulation of absolute values to *single-precision*.

Table 3.160: *f32v4absacc* instruction definition

<i>f32v4absacc</i>		worker	aux
Syntax	<i>f32v4absacc</i> <i>\$aSrc0:Src0+3</i>		
Semantics	Prepare	<pre>array<Single,4> op0 = { TFPU_F32FromBits(\$aSrc0:Src0+3[0]), TFPU_F32FromBits(\$aSrc0:Src0+3[1]), TFPU_F32FromBits(\$aSrc0:Src0+3[2]), TFPU_F32FromBits(\$aSrc0:Src0+3[3]) }; array<Single,4> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (TFPU_F32_IsSNan(op0[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>result[0] = TFPU_Add(op0[0], \$AACC[0], TFPU_FP32, TFPU_ABS); result[1] = TFPU_Add(op0[1], \$AACC[2], TFPU_FP32, TFPU_ABS); result[2] = TFPU_Add(op0[2], \$AACC[4], TFPU_FP32, TFPU_ABS); result[3] = TFPU_Add(op0[3], \$AACC[6], TFPU_FP32, TFPU_ABS);</pre>	
	Commit	<pre>\$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3];</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_F32FromBits* , *TFPU_F32_IsSNan* , *TFPU_IsMalign* , *TFPU_Add*

3.7.3.6.2 *f32v4acc*

Single-precision 4-element vector accumulation to *Single-precision*.

Table 3.161: *f32v4acc* instruction definition

<i>f32v4acc</i>		worker	aux
Syntax	f32v4acc \$aSrc0:Src0+3		
Semantics	Prepare	<pre>array<Single,4> op0 = { TFPU_F32FromBits(\$aSrc0:Src0+3[0]), TFPU_F32FromBits(\$aSrc0:Src0+3[1]), TFPU_F32FromBits(\$aSrc0:Src0+3[2]), TFPU_F32FromBits(\$aSrc0:Src0+3[3]) }; array<Single,4> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (TFPU_F32_IsSNan(op0[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>result[0] = TFPU_Add(op0[0], \$AACC[0], TFPU_FP32); result[1] = TFPU_Add(op0[1], \$AACC[2], TFPU_FP32); result[2] = TFPU_Add(op0[2], \$AACC[4], TFPU_FP32); result[3] = TFPU_Add(op0[3], \$AACC[6], TFPU_FP32);</pre>	
	Commit	<pre>\$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3];</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_F32FromBits* , *TFPU_F32_IsSNan* , *TFPU_IsMalign* , *TFPU_Add*

3.7.3.6.3 f32v4sqacc

Single-precision 4-element vector accumulation of squares to *single-precision*.

Table 3.162: *f32v4sqacc* instruction definition

<i>f32v4sqacc</i>		worker	aux
Syntax	f32v4sqacc \$aSrc0:Src0+3		
Semantics	Prepare	<pre>array<Single,4> op0 = { TFPU_F32FromBits(\$aSrc0:Src0+3[0]), TFPU_F32FromBits(\$aSrc0:Src0+3[1]), TFPU_F32FromBits(\$aSrc0:Src0+3[2]), TFPU_F32FromBits(\$aSrc0:Src0+3[3]) }; array<Single,4> result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; ++i) { if (TFPU_F32_IsSNan(op0[i])) { fpExcpt = TFPEXCPT_INV; } } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; result[0] = TFPU_Mac(op0[0], op0[0], \$AACC[0], TFPU_FP32, rmode); result[1] = TFPU_Mac(op0[1], op0[1], \$AACC[2], TFPU_FP32, rmode); result[2] = TFPU_Mac(op0[2], op0[2], \$AACC[4], TFPU_FP32, rmode); result[3] = TFPU_Mac(op0[3], op0[3], \$AACC[6], TFPU_FP32, rmode);</pre>	
	Commit	<pre>\$AACC[0] = result[0]; \$AACC[2] = result[1]; \$AACC[4] = result[2]; \$AACC[6] = result[3];</pre>	

Architectural state references: *\$FP_STS* , *\$FP_CTL* , *\$AACC*

Function references: *TFPU_F32FromBits* , *TFPU_F32_IsSNan* , *TFPU_IsMalign* , *TFPU_Mac*

3.7.3.7 f32 scalar

3.7.3.7.1 f32absadd

Scalar *single-precision* addition of two absolute register source values.

Table 3.163: *f32absadd* instruction definition

<i>f32absadd</i>		worker	aux
Syntax	f32absadd \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); Single result;	
	Except In	op1 = fabs(op1); op2 = fabs(op2); uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);	
	Compute	if (!specialCase) { double res = static_cast<double>(op1) + static_cast<double>(op2); result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NAN00); }	
	Except Out	\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.7.2 f32absmax

Determine the maximum floating point value from two absolute register source values.

Table 3.164: *f32absmax* instruction definition

<i>f32absmax</i>		worker	aux
Syntax	f32absmax \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1);	
	Except In	Single ops[2] = { op1, op2 }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 2); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Compute	Single result = TFPU_Max(fabs(op1), fabs(op2));	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.7.3 f32add

Single-precision addition of two register source values.

Table 3.165: *f32add* instruction definition

<i>f32add</i>		worker	aux
Syntax	<code>f32add \$aDst0, \$aSrc0, \$aSrc1</code>		
Semantics	Prepare	<pre>Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); Single result;</pre>	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);</pre>	
	Compute	<pre>if (!specialCase) { double res = static_cast<double>(op1) + static_cast<double>(op2); result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NAN00); }</pre>	
	Except Out	<pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	<pre>\$aDst0 = TFPU_BitsFromF32(result);</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

f32add occurs in the following code examples:

- *f16v4cmac* example

3.7.3.7.4 f32clamp

Single-precision floating-point vector min-of-maximum

Table 3.166: *f32clamp* instruction definition

<i>f32clamp</i>		worker	aux
Syntax	f32clamp \$aDst0, \$aSrc0, \$aSrc1:Src1+1		
Semantics	Prepare	<pre>Single op1 = TFPU_F32FromBits(\$aSrc0); array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; Single result;</pre>	
	Except In	<pre>Single ops[3] = { op1, op2[0], op2[1] }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 3); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>Single lower = op2[0]; Single upper = op2[1]; // Unlike min/max, clamp explicitly propagates (regenerates) // NaN inputs if (isnan(lower) isnan(upper)) { result = TFPU_F32_QNan(); } else { if (isnan(op1)) { result = TFPU_F32_QNan(); } else if (op1 > upper) { result = upper; } else { result = TFPU_Max(op1, lower); } }</pre>	
	Commit	<pre>\$aDst0 = TFPU_BitsFromF32(result);</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_F32_QNan*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.7.5 f32class

Single-precision floating-point number classifier. IEEE 754-2008: 5.7.2

Table 3.167: *f32class* instruction definition

<i>f32class</i>	worker	aux
Syntax	f32class \$aDst0, \$aSrc0	
Semantics	<p>Prepare DataWord op0; Single op1 = TFPU_F32FromBits(\$aSrc0);</p> <p>Compute bool sign = signbit(op1);</p> <pre>// Denorms will have been flushed to zero if (TFPU_F32_IsSNan(op1)) { op0 = TFPU_CLASS_SNaN; } else if (TFPU_F32_IsQNaN(op1)) { op0 = TFPU_CLASS_QNaN; } else if (isinf(op1)) { op0 = (sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF); } else if (fabs(op1) == 0.0) { op0 = (sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO); } else { op0 = (sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM); }</pre> <p>Commit \$aDst0 = op0;</p>	

Function references: *TFPU_F32FromBits*, *TFPU_F32_IsSNan*, *TFPU_F32_IsQNaN*

3.7.3.7.6 f32cmpeq

Test if two floating point numbers are equal. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.168: *f32cmpeq* instruction definition

<i>f32cmpeq</i>	worker	aux
Syntax	f32cmpeq \$aDst0, \$aSrc0, \$aSrc1	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1);</p> <p>TileFPRelation_t relation = TFPU_Relation(op1, op2);</p> <p>Except In // IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE;</p> <pre>if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre> <p>Compute bool result = (relation == TFPU_RELATION_EQ);</p> <p>Commit \$aDst0 = (result ? 0xffffffff : 0x00000000);</p>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.7 f32cmpge

Test if a floating point number is greater than or equal to a second floating point number. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.169: *f32cmpge* instruction definition

<i>f32cmpge</i>		worker	aux
Syntax	f32cmpge \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	<pre>Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); TileFPRelation_t relation = TFPU_Relation(op1, op2);</pre>	
	Except In	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>bool result = ((relation == TFPU_RELATION_GT) (relation == TFPU_RELATION_EQ));</pre>	
	Commit	<pre>\$aDst0 = (result ? 0xffffffff : 0x00000000);</pre>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.8 f32cmpgt

Test if a floating point number is greater than a second floating point number. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.170: *f32cmpgt* instruction definition

<i>f32cmpgt</i>		worker	aux
Syntax	f32cmpgt \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); TileFPRelation_t relation = TFPU_Relation(op1, op2);	
	Except In	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	bool result = (relation == TFPU_RELATION_GT);	
	Commit	\$aDst0 = (result ? 0xffffffff : 0x00000000);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.9 *f32cmple*

Test if a floating point number is less than or equal to a second floating point number. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.171: *f32cmple* instruction definition

<i>f32cmple</i>		worker	aux
Syntax	f32cmple \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); TileFPRelation_t relation = TFPU_Relation(op1, op2);	
	Except In	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	bool result = ((relation == TFPU_RELATION_LT) (relation == TFPU_RELATION_EQ));	
	Commit	\$aDst0 = (result ? 0xffffffff : 0x00000000);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.10 f32cmplt

Test if a floating point number is less than a second floating point number. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.172: *f32cmplt* instruction definition

<i>f32cmplt</i>		worker	aux
Syntax	f32cmplt \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); TileFPRelation_t relation = TFPU_Relation(op1, op2);	
	Except In	// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Compute	bool result = (relation == TFPU_RELATION_LT);	
	Commit	\$aDst0 = (result ? 0xffffffff : 0x00000000);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.11 f32cmpne

Test if a floating point number is not equal to second floating point number. If so, the destination register is set to *TFPU_FP32_TRUE*, otherwise it is set to *TFPU_FP32_FALSE*.

Table 3.173: *f32cmpne* instruction definition

<i>f32cmpne</i>		worker	aux
Syntax	f32cmpne \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); TileFPRelation_t relation = TFPU_Relation(op1, op2);	
	Except In	<pre>// IEEE 754-2008: 5.11, 7.2 uint32_t fpExcpt = TFPEXCPT_NONE; if (TFPU_RELATION_UN == relation) { fpExcpt = TFPEXCPT_INV; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	bool result = (relation != TFPU_RELATION_EQ);	
	Commit	\$aDst0 = (result ? 0xffffffff : 0x00000000);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_Relation*, *TFPU_IsMalign*

3.7.3.7.12 *f32div*

Floating point division of two register source values.

Table 3.174: *f32div* instruction definition

<i>f32div</i>		worker	aux
Syntax	f32div \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); Single result;	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoDivPreExecute(op1, op2, &fpExcpt, &result);</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { Double res = op1 / op2; result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode); fpExcpt = TFPU_GenOFL0Check(result, TFPU_FP32, TFPU_NO_NAN00); }</pre>	
	Except Out	<pre>if (TFPU_F32DivExceptIsImprecise(fpExcpt, \$FP_CTL)) { state.exceptionImprecise = TEXCPT_FP; } \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoDivPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_GenOFLOCheck* , *TFPU_F32DivExceptIsImprecise* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

f32div occurs in the following code examples:

- *ldb16b16* example

3.7.3.7.13 *f32exp*

Table 3.175: *f32exp* instruction definition

<i>f32exp</i>		worker	aux
Syntax	<i>f32exp</i> <i>\$aDst0</i> , <i>\$aSrc0</i>		
Semantics	Prepare	Single op1 = <i>TFPU_F32FromBits</i> (\$aSrc0); Single result;	
	Except In	uint32_t fpExcpt = <i>TFPEXCPT_NONE</i> ; bool specialCase = <i>TFPU_DoExpPreExecute</i> (op1, &fpExcpt, &result);	
	Compute	TileRoundMode_t rmode = <i>\$FP_CTL.RND</i> ; if (!specialCase) { result = <i>TFPU_F32Exp</i> (op1, <i>TFPU_BASE_E</i> , rmode); fpExcpt = <i>TFPU_GenOFLOCheck</i> (result, <i>TFPU_FP32</i> , <i>TFPU_NO_NAN00</i>); }	
	Except Out	<i>\$FP_STS</i> = <i>\$FP_STS</i> fpExcpt; if (<i>TFPU_IsMalign</i> (fpExcpt, <i>\$FP_CTL</i>)) { <i>EXCEPT</i> (<i>TEXCPT_FP</i>); }	
	Commit	<i>\$aDst0</i> = <i>TFPU_BitsFromF32</i> (result);	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoExpPreExecute* , *TFPU_F32Exp* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.14 f32exp2

Table 3.176: *f32exp2* instruction definition

<i>f32exp2</i>		worker	aux
Syntax	f32exp2 \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;	
	Except In	<pre>uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoExpPreExecute(op1, &fpExcpt, &result);</pre>	
	Compute	<pre>TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Exp(op1, TFPU_BASE_2, rmode); fpExcpt = TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NAN00); }</pre>	
	Except Out	<pre>\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_DoExpPreExecute*, *TFPU_F32Exp*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.7.15 f32int

Round a *single-precision* floating-point value to an integral, rounding as specified by the instruction immediate.

Table 3.177: *f32int* instruction definition

<i>f32int</i>		worker	aux
Syntax	f32int \$aDst0, \$aSrc1, enumRnd		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc1); DataWord op2 = enumRnd;	
	Except In	<pre>// IEEE 754-2008: 5.9 Single ops[] = { op1 }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 1); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</pre>	
	Compute	<pre>TileRoundMode_t mode = op2 & 0x7; Single result = TFPU_RoundFP32ToIntegral(op1, mode);</pre>	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_RoundFP32ToIntegral*, *TFPU_BitsFromF32*

3.7.3.7.16 f32ln

Table 3.178: *f32ln* instruction definition

<i>f32ln</i>		worker	aux
Syntax	f32ln \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;	
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoLogPreExecute(op1, &fpExcpt, &result);	
	Compute	TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Log(op1, TFPU_BASE_E, rmode); }	
	Except Out	\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoLogPreExecute* , *TFPU_F32Log* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.17 f32log2

Table 3.179: *f32log2* instruction definition

<i>f32log2</i>		worker	aux
Syntax	f32log2 \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;	
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoLogPreExecute(op1, &fpExcpt, &result);	
	Compute	TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Log(op1, TFPU_BASE_2, rmode); }	
	Except Out	\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoLogPreExecute* , *TFPU_F32Log* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.18 f32mac

Single-precision floating-point multiplication of two source registers with *single-precision* accumulate.

Table 3.180: *f32mac* instruction definition

<i>f32mac</i>	worker	aux
Syntax	f32mac \$aSrc0, \$aSrc1	
Semantics	Prepare	Single op0 = TFPU_F32FromBits(\$aSrc0); Single op1 = TFPU_F32FromBits(\$aSrc1); Single result;
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoMacPreExecute(op0, op1, \$AACC[0], &fpExcpt, &result); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }
	Compute	TileRoundMode_t rmode = \$FP_CTL.RND; // MAC isn't "fused" in the sense that Tile rounds to the accumulator // format prior to the addition. if (!specialCase) { float mRes = TFPU_Mul(op0, op1, TFPU_FP32, rmode); result = TFPU_Add(mRes, \$AACC[0], TFPU_FP32); }
	Commit	\$AACC[0] = result;

Architectural state references: *\$AACC* , *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_DoMacPreExecute* , *TFPU_IsMalign* , *TFPU_Mul* , *TFPU_Add*

f32mac occurs in the following code examples:

- *f32mac* example

Listing 3.10: *f32mac* example

```

// Load first pair of weights plus 1 sparse data item
ld64a32    $input1AndWeight, $weightPtr++, $mvertex_base, $deltas

.align 8
{
  rpt      $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
_loop_start:
// Performance is 1 single-precision fmac per tick
{
  ldd16v2a32 $input0, $deltaPtr++, $mvertex_base, $deltas@
  f32mac    $input1, $weight1
}
{
  ld64a32    $input1AndWeight, $weightPtr++, $mvertex_base, $deltas
  f32mac    $input0, $weight0
}
_loop_end:

// Final fmacs
{
  ldd16v2a32 $input0, $deltaPtr++, $mvertex_base, $deltas@
  f32mac    $input1, $weight1
}
f32mac    $input0, $weight0

```

```
// Read out the accumulator result
f32v2gina $a0:1, $zeros, 0
```

3.7.3.7.19 f32max

Determine the maximum floating point value from two register source values.

Table 3.181: *f32max* instruction definition

<i>f32max</i>	worker	aux
Syntax	f32max \$aDst0, \$aSrc0, \$aSrc1	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1);</p> <p>Except In Single ops[2] = { op1, op2 }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 2); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Compute Single result = TFPU_Max(op1, op2);</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_Max*, *TFPU_BitsFromF32*

3.7.3.7.20 f32min

Determine the minimum floating point value from two register source values.

Table 3.182: *f32min* instruction definition

<i>f32min</i>	worker	aux
Syntax	f32min \$aDst0, \$aSrc0, \$aSrc1	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1);</p> <p>Except In Single ops[2] = { op1, op2 }; uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 2); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Compute Single result = TFPU_Min(op1, op2);</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_GenSNanCheck*, *TFPU_IsMalign*, *TFPU_Min*, *TFPU_BitsFromF32*

3.7.3.7.21 f32mul

Single precision floating point multiplication on 2 source register values.

Table 3.183: *f32mul* instruction definition

<i>f32mul</i>		worker	aux
Syntax	f32mul \$aDst0, \$aSrc0, \$aSrc1		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); Single result;	
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoMulPreExecute(op1, op2, &fpExcpt, &result);	
	Compute	TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { Double res = op1 * op2; fpExcpt = TFPU_GenOFLOCheck(res, TFPU_FP32, TFPU_NO_NAN00); result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode); }	
	Except Out	\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_DoMulPreExecute*, *TFPU_GenOFLOCheck*, *TFPU_RoundFP64ToFmt*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.7.22 f32oorx

Single-precision reciprocal of square-root.

Table 3.184: *f32oorx* instruction definition

<i>f32oorx</i>		worker	aux
Syntax	f32oorx \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;	
	Except In	uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoRSqrtPreExecute(op1, &fpExcpt, &result);	
	Compute	TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32RSqrt(op1, rmode); }	
	Except Out	\$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }	
	Commit	\$aDst0 = TFPU_BitsFromF32(result);	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoRSqrtPreExecute* , *TFPU_F32RSqrt* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.23 f32oox

Single-precision reciprocal.

Table 3.185: *f32oox* instruction definition

<i>f32oox</i>	worker	aux
Syntax	f32oox \$aDst0, \$aSrc0	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;</p> <p>Except In uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoRecipPreExecute(op1, &fpExcpt, &result);</p> <p>Compute TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Recip(op1, rmode); }</p> <p>Except Out \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_CTL* , *\$FP_STS*

Function references: *TFPU_F32FromBits* , *TFPU_DoRecipPreExecute* , *TFPU_F32Recip* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.24 f32sigm

Table 3.186: *f32sigm* instruction definition

<i>f32sigm</i>	worker	aux
Syntax	f32sigm \$aDst0, \$aSrc0	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0);</p> <p>Except In uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoSigmoidPreExecute(op1, &fpExcpt, &result); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Compute TileRoundMode_t rmode = \$FP_CTL.RND; Single result = specialCase ? result : TFPU_F32Sigmoid(op1, rmode);</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_STS* , *\$FP_CTL*

Function references: *TFPU_F32FromBits* , *TFPU_DoSigmoidPreExecute* , *TFPU_IsMalign* , *TFPU_F32Sigmoid* , *TFPU_BitsFromF32*

3.7.3.7.25 f32sisoamp

Single-precision floating point **accumulating matrix-vector product**. Input partial-sums and result values are *single-precision*.

enumFlags format:

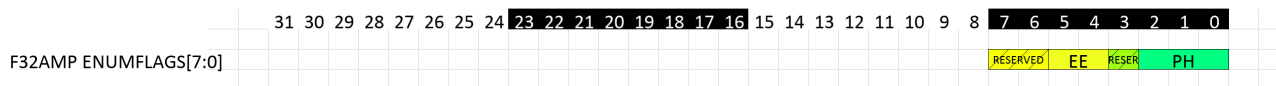


Fig. 3.22: f32sisoamp immediate format

8 output channels are processed/produced.

Table 3.187: *f32sisoamp* instruction definition

<i>f32sisoamp</i>	worker	aux
Syntax	f32sisoamp \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags	
Semantics	Prepare	<pre> Single op1 = TFPU_F32FromBits(\$aSrc0); array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; vector<Single> out; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weight; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> addend; array<bool,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> specialCase; // Engine enables uint32_t ee = F32AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain // Extract immediate config unsigned phase = F32AMP_ENUMFLAGS__PH__GET(op3); unsigned ampUnits = TFPU_AMP_UNITS_PER_SET; if (phase == 0) { out = { \$AACC[0], \$AACC[2] }; } else { out = { \$AACC[1], \$AACC[3] }; } for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { // Weight selection uint64_t cwei = context.getCCCSState((k * 4) + (phase >> 1)); weight[k] = TFPU_F32FromBits(cwei >> (32 * (phase & 1))); if (phase == 0) { addend[k] = \$AACC[(k * TFPU_AACC_PER_AMP_UNIT) + 1]; } else { addend[k] = \$AACC[(k * TFPU_AACC_PER_AMP_UNIT)]; } } } Except In uint32_t fpExcpt = TFPEXCPT_NONE; fpExcpt = TFPU_SNaNCheck(op2[0]); fpExcpt = TFPU_SNaNCheck(op2[1]); fpExcpt = TFPU_SNaNCheck(op1); for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { fpExcpt = TFPU_SNaNCheck(weight[k]); specialCase[k] = TFPU_DoMulPreExecute(op1, weight[k], &fpExcpt, &result[k]); } } </pre>

Continued on next page

Table 3.187: *f32sisoamp* instruction definition (continued)

<i>f32sisoamp</i>	worker	aux
Syntax	f32sisoamp \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags	
Semantics	Compute	<pre> for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { if (!specialCase[k]) { Double res = op1 * weight[k]; result[k] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); } result[k] = TFPU_Add(addend[k], result[k], TFPU_FP32); } } </pre>
	Except Out	<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>
	Commit	<pre> // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[(k * 2)] = result[k]; Single incomingp; if (phase == 0) { // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * TFPU_AACC_PER_AMP_UNIT]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Consume new partial-sum inputs // into our odd accumulator if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } } \$AACC[(k * TFPU_AACC_PER_AMP_UNIT) + 1] = incomingp; } } </pre>

Continued on next page

Table 3.187: *f32sisoamp* instruction definition (continued)

<i>f32sisoamp</i>		worker	aux
Syntax	<i>f32sisoamp</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0</i> , <i>\$aSrc1:Src1+1</i> , <i>enumFlags</i>		
Semantics	Commit cont'd	<pre> } else { // phase != 0 if ((phase & 1) == 0) { // Even phases // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled incomingp = \$AACC[((k + 2) * TFPU_AACC_PER_AMP_UNIT) + 1]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } } \$AACC[(k * TFPU_AACC_PER_AMP_UNIT) + 1] = incomingp; } } } } // Final output \$aDst0:Dst0+1 = { TFPU_BitsFromF32(isnan(out[0]) ? TFPU_F32_QNan() : out[0]), TFPU_BitsFromF32(isnan(out[1]) ? TFPU_F32_QNan() : out[1]) }; </pre>	

Architectural state references: *\$AACC*, *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_SNaNCheck*, *TFPU_DoMulPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_Add*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QNan*, *TFPU_BitsFromF32*

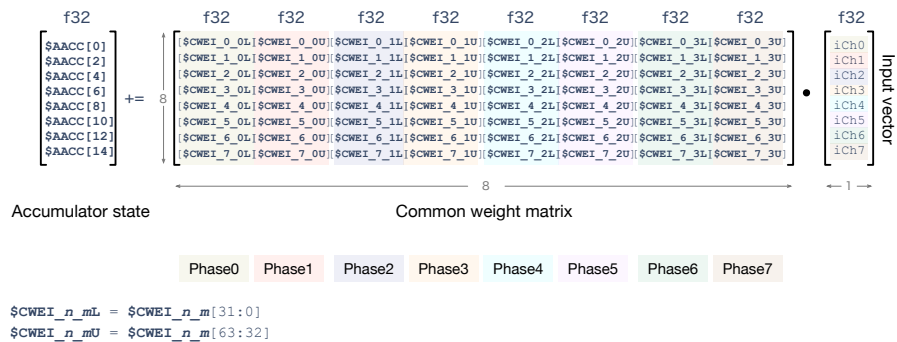


Fig. 3.23: *f32sisoamp*

f32amp convolution operation

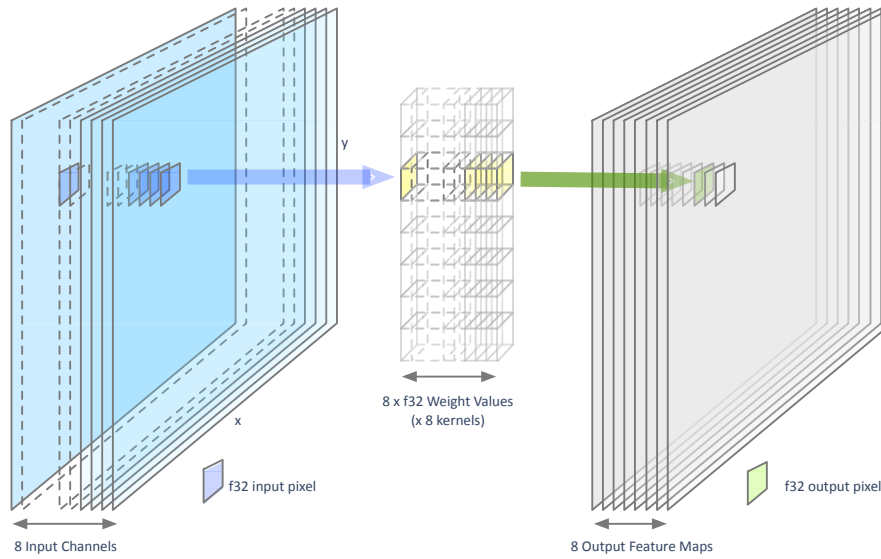


Fig. 3.24: f32sisoamp

f32sisoamp occurs in the following code examples:

- *f32sisoamp* example

Listing 3.11: f32sisoamp example

```
.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
_loop_start:
{
  nop
  // Supply single f32 input and 2 x partial sum inputs.
  // Obtain 2 x f32 partial sum results
  f32sisoamp   $outPartials, $inData0, $inPartials, TAMP_F32_E4_P0
}
{
  // Load 2 x f32 inputs along with 2 x f32 partial sums, store 2 x f32 partial sums
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+%, $mzero, 0
  // Supply single f32 input. No results provided and no partials used on odd phases
  f32sisoamp   $azeros, $inData1, $azeros, TAMP_F32_E4_P1
}
{
  nop
  f32sisoamp   $outPartials, $inData0, $inPartials, TAMP_F32_E4_P2
}
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+%, $mzero, 0
  f32sisoamp   $azeros, $inData1, $azeros, TAMP_F32_E4_P3
}
{
  nop
  f32sisoamp   $outPartials, $inData0, $inPartials, TAMP_F32_E4_P4
}
{
  ld2xst64pace $inDataAndPartials, $outPartials, $triPtr+%, $mzero, 0
  f32sisoamp   $azeros, $inData1, $azeros, TAMP_F32_E4_P5
}
{
  nop
  f32sisoamp   $outPartials, $inData0, $inPartials, TAMP_F32_E4_P6
}
{
```

```
    ld2xst64pace $inDataAndPartials, $outPartials, $striPtr+%, $mzero, 0
    f32sisoamp   $azeros, $inData1, $azeros, TAMP_F32_E4_P7
}
_loop_end:
```

3.7.3.7.26 f32sisoslic

Single-precision floating point **slim** convolution. Input partial-sums are *single-precision*. Results are *single-precision*.

Table 3.188: f32sisolic, 2x1x3x2 example sequence

\$aSrc0 I	\$AACC[14]	\$AACC[10]	\$AACC[6]	\$AACC[2]	\$AACC[12]	\$AACC[8]	\$AACC[4]	\$AACC[0]	\$aDst0
$x_0^L P_0,P_1$	-	$R_1 = x_0^L \times CW_{5,0}^L + P_1$	-	-	-	$R_0 = x_0^L \times CW_{4,0}^L + P_0$	-	-	-
$x_0^U D_0,D_1$	-	$R_1 += x_0^U \times CW_{5,0}^U$	-	-	-	$R_0 += x_0^U \times CW_{4,0}^U$	-	-	-
$x_1^L P_2,P_3$	-	$R_3 = x_1^L \times CW_{5,0}^L + P_3$	$R_1 += x_1^L \times CW_{3,0}^L$	-	-	$R_2 = x_1^L \times CW_{4,0}^L + P_2$	$R_0 += x_1^L \times CW_{2,0}^L$	-	-
$x_1^U D_2,D_3$	-	$R_3 += x_1^U \times CW_{5,0}^U$	$R_1 += x_1^U \times CW_{3,0}^U$	-	-	$R_2 += x_1^U \times CW_{4,0}^U$	$R_0 += x_1^U \times CW_{2,0}^U$	-	-
$x_2^L P_4,P_5$	-	$R_5 = x_2^L \times CW_{5,0}^L + P_5$	$R_3 += x_2^L \times CW_{3,0}^L$	$R_1 += x_2^L \times CW_{1,0}^L$	-	$R_4 = x_2^L \times CW_{4,0}^L + P_4$	$R_2 += x_2^L \times CW_{2,0}^L$	$R_0 += x_2^L \times CW_{0,0}^L$	-
$x_2^U D_4,D_5$	-	$R_5 += x_2^U \times CW_{5,0}^U$	$R_3 += x_2^U \times CW_{3,0}^U$	$R_1 += x_2^U \times CW_{1,0}^U$	-	$R_4 += x_2^U \times CW_{4,0}^U$	$R_2 += x_2^U \times CW_{2,0}^U$	$R_0 += x_2^U \times CW_{0,0}^U$	-
$x_3^L P_6,P_7$	-	$R_7 = x_3^L \times CW_{5,0}^L + P_7$	$R_5 += x_3^L \times CW_{3,0}^L$	$R_3 += x_3^L \times CW_{1,0}^L$	-	$R_6 = x_3^L \times CW_{4,0}^L + P_6$	$R_4 += x_3^L \times CW_{2,0}^L$	$R_2 += x_3^L \times CW_{0,0}^L$	R_0, R_1
$x_3^U D_6,D_7$	-	$R_7 += x_3^U \times CW_{5,0}^U$	$R_5 += x_3^U \times CW_{3,0}^U$	$R_3 += x_3^U \times CW_{1,0}^U$	-	$R_6 += x_3^U \times CW_{4,0}^U$	$R_4 += x_3^U \times CW_{2,0}^U$	$R_2 += x_3^U \times CW_{0,0}^U$	-
$x_4^L P_8,P_9$	-	$R_9 = x_4^L \times CW_{5,0}^L + P_9$	$R_7 += x_4^L \times CW_{3,0}^L$	$R_5 += x_4^L \times CW_{1,0}^L$	-	$R_8 = x_4^L \times CW_{4,0}^L + P_8$	$R_6 += x_4^L \times CW_{2,0}^L$	$R_4 += x_4^L \times CW_{0,0}^L$	R_2, R_3
$x_4^U D_8,D_9$	-	$R_9 += x_4^U \times CW_{5,0}^U$	$R_7 += x_4^U \times CW_{3,0}^U$	$R_5 += x_4^U \times CW_{1,0}^U$	-	$R_8 += x_4^U \times CW_{4,0}^U$	$R_6 += x_4^U \times CW_{2,0}^U$	$R_4 += x_4^U \times CW_{0,0}^U$	-
$x_5^L P_{10},P_{11}$	-	$R_{11} = x_5^L \times CW_{5,0}^L + P_{11}$	$R_9 += x_5^L \times CW_{3,0}^L$	$R_7 += x_5^L \times CW_{1,0}^L$	-	$R_{10} = x_5^L \times CW_{4,0}^L + P_{10}$	$R_8 += x_5^L \times CW_{2,0}^L$	$R_6 += x_5^L \times CW_{0,0}^L$	R_4, R_5
$x_5^U D_{10},D_{11}$	-	$R_{11} += x_5^U \times CW_{5,0}^U$	$R_9 += x_5^U \times CW_{3,0}^U$	$R_7 += x_5^U \times CW_{1,0}^U$	-	$R_{10} += x_5^U \times CW_{4,0}^U$	$R_8 += x_5^U \times CW_{2,0}^U$	$R_6 += x_5^U \times CW_{0,0}^U$	-

P_n is *single-precision* input partial-sum n

D_n is 0 under normal circumstances (\$a14:15)

x_n^L is the 1st element (element 0) of a *f32v2* input vector

x_n^U is the 2nd element (element 1) of a *f32v2* input vector

$CW_{m,n}^L$ is the least significant 32-bits of common weight state \$CWEI_m_n

$CW_{m,n}^U$ is the most significant 32-bits of common weight state \$CWEI_m_n

R_n is the final *single-precision* result of successive multiply-accumulations that began with P_n

enumFlags format:

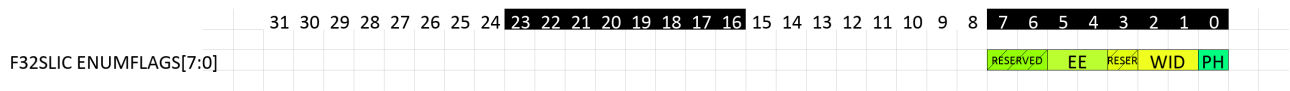


Fig. 3.25: f32sisoslic immediate format

2 output channels are processed/produced.

Table 3.189: *f32sisoslic* instruction definition

<i>f32sisoslic</i>	worker	aux
Syntax	<code>f32sisoslic \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags</code>	
Semantics	<p>Prepare</p> <pre> Single op1 = TFPU_F32FromBits(\$aSrc0); array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> incomingp; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weight; array<bool,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> specialCase; // Extract immediate config unsigned phase = F32SLIC_ENUMFLAGS__PH__GET(op3); unsigned ampUnits = TFPU_AMP_UNITS_PER_SET; // Engine enables uint32_t ee = F32SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain unsigned wid = F32SLIC_ENUMFLAGS__WID__GET(op3); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { // Weight selection uint64_t cwei = context.getCCCSState((u * 4) + wid); weight[u] = TFPU_F32FromBits(cwei >> (32 * phase)); if (phase == 0) { if (engineEnable[engine + 1]) { // There is an engine behind me and it // is enabled. Use its current accumulator value incomingp[u] = \$AACC[(u + 2) * TFPU_AACC_PER_AMP_UNIT]; } else { // Engines behind me are disabled // (or I am the final engine in the chain). // Use the new partial-sum inputs incomingp[u] = op2[u & 1]; } } else { // Accumulate into the current value incomingp[u] = \$AACC[u * TFPU_AACC_PER_AMP_UNIT]; } } } // Output exception check vector<Single> out = { \$AACC[0], \$AACC[2] }; </pre>	

Continued on next page

Table 3.189: *f32sisoslic* instruction definition (continued)

<i>f32sisoslic</i>	worker	aux
Syntax	<code>f32sisoslic \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags</code>	
Semantics	<pre> Except In uint32_t fpExcpt = TFPEXCPT_NONE; fpExcpt = TFPU_SNaNCheck(op2[0]); fpExcpt = TFPU_SNaNCheck(op2[1]); fpExcpt = TFPU_SNaNCheck(op1); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { fpExcpt = TFPU_SNaNCheck(weight[u]); specialCase[u] = TFPU_DoMulPreExecute(op1, weight[u], &fpExcpt, &result[u]); } } Compute for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { if (!specialCase[u]) { Double res = op1 * weight[u]; result[u] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); } // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp[u], result[u], TFPU_FP32); } } Except Out // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[u * 2] = result[u]; } } // Final output processing \$aDst0:Dst0+1 = { TFPU_BitsFromF32(isnan(out[0]) ? TFPU_F32_QNaN() : out[0]), TFPU_BitsFromF32(isnan(out[1]) ? TFPU_F32_QNaN() : out[1]) }; </pre>	

Architectural state references: `$AACC`, `$FP_STS`, `$FP_CTL`

Function references: `TFPU_F32FromBits`, `TFPU_SNaNCheck`, `TFPU_DoMulPreExecute`, `TFPU_RoundFP64ToFmt`, `TFPU_Add`, `TFPU_AACCReadFlags`, `TFPU_IsMalign`, `TFPU_BitsFromF32`, `TFPU_F32_QNaN`

f32slic convolution operation

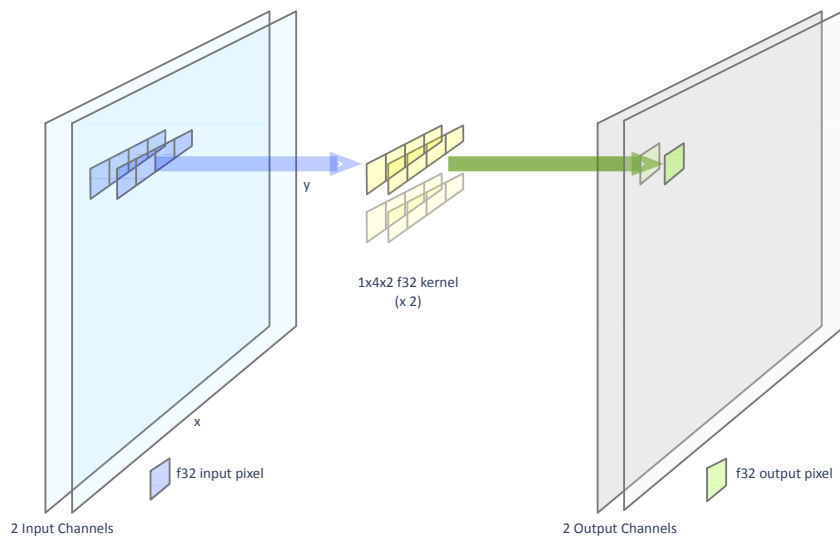


Fig. 3.26: f32sisoslic

f32sisoslic occurs in the following code examples:

- *f32sisoslic* example

Listing 3.12: f32sisoslic example

```
.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
_loop_start:
{
  // Store out f32v2 result (2 output channels)
  st64pace    $outPartials, $striPtr+=, $mzero, 0
  f32sisoslic $outPartials, $inData0, $inPartials, TSLIC_F32_1x3_W0_P0
}
{
  // Load f32v2 input (2 channels) plus 2 partial-sums (2 channels)
  ld2x64pace  $inData, $inPartials, $striPtr+=, $mzero, 0
  f32sisoslic $azeros, $inData1, $azeros, TSLIC_F32_1x3_W0_P1
}
_loop_end:

// Store out final f32v2 result (2 output channels)
st64pace    $outPartials, $striPtr+=, $mzero, 0
```

3.7.3.7.27 f32sisov2amp

Single-precision floating point **accumulating matrix-vector product**. Input partial-sums and result values are *single-precision*.

enumFlags format:

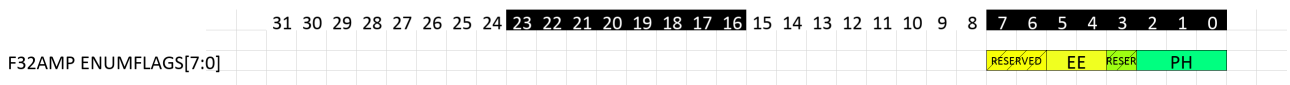


Fig. 3.27: f32sisoamp immediate format

16 output channels are processed/produced.

Table 3.190: *f32sisov2amp* instruction definition

<i>f32sisov2amp</i>	worker	aux
Syntax	<i>f32sisov2amp</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0</i> , <i>\$aSrc1:Src1+1</i> , <i>enumFlags</i>	
Semantics	<p>Prepare</p> <pre> Single op1 = TFPU_F32FromBits(\$aSrc0); array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; vector<Single> out; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> input; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weight; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> addend; array<bool,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> specialCase; // Engine enables uint32_t ee = F32AMP_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain // Extract immediate config unsigned phase = F32AMP_ENUMFLAGS__PH__GET(op3); unsigned ampSets = 2; unsigned ampUnits = TFPU_AMP_UNITS_PER_SET * ampSets; unsigned oSet = phase & 1; // AMP set arrangement array<unsigned,2> specialPhase = {0, 1}; unsigned setPhaseLag[2] = {0, 1}; unsigned baseReg = oSet * TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; if (phase == specialPhase[oSet]) { out = { \$AACC[baseReg], \$AACC[baseReg+2] }; } else { out = { \$AACC[baseReg+1], \$AACC[baseReg+3] }; } for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { // Weight selection unsigned set = k / TFPU_AMP_UNITS_PER_SET; unsigned lPhase = (phase - setPhaseLag[set]) & 0x7; uint64_t cwei = context.getCCCSState((k * 4) + (lPhase >> 1)); weight[k] = TFPU_F32FromBits(cwei >> (32 * (lPhase & 1))); // Input selection input[k] = (set == 0) ? op1 : TFPU_F32FromBits(\$TAS); if (phase == specialPhase[set]) { addend[k] = \$AACC[(k * TFPU_AACC_PER_AMP_UNIT) + 1]; } else { addend[k] = \$AACC[(k * TFPU_AACC_PER_AMP_UNIT)]; } } } </pre>	

Continued on next page

Table 3.190: *f32sisov2amp* instruction definition (continued)

<i>f32sisov2amp</i>		worker	aux
Syntax	<i>f32sisov2amp</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0</i> , <i>\$aSrc1:Src1+1</i> , <i>enumFlags</i>		
Semantics	<pre> Except In uint32_t fpExcpt = TFPEXCPT_NONE; fpExcpt = TFPU_SNaNCheck(op2[0]); fpExcpt = TFPU_SNaNCheck(op2[1]); fpExcpt = TFPU_SNaNCheck(op1); fpExcpt = TFPU_SNaNCheck(TFPU_F32FromBits(\$TAS)); for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { fpExcpt = TFPU_SNaNCheck(weight[k]); specialCase[k] = TFPU_DoMulPreExecute(input[k], weight[k], &fpExcpt, &result[k]); } } Compute for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { if (!specialCase[k]) { Double res = input[k] * weight[k]; result[k] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); } result[k] = TFPU_Add(addend[k], result[k], TFPU_FP32); } } Except Out // Floating-point result exception check fpExcpt = TFPU_AACCRdFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } Commit // AMP set arrangement unsigned propagateOddAccs[2] = {0, 1}; // Input partial sum consumption and internal state updates for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; unsigned set = k / TFPU_AMP_UNITS_PER_SET; if (engineEnable[engine]) { \$AACC[(k * 2)] = result[k]; } } </pre>		

Continued on next page

Table 3.190: *f32sisov2amp* instruction definition (continued)

<i>f32sisov2amp</i>		worker	aux
Syntax	<i>f32sisov2amp</i> <i>\$aDst0:Dst0+1</i> , <i>\$aSrc0</i> , <i>\$aSrc1:Src1+1</i> , <i>enumFlags</i>		
Semantics	Commit cont'd	<pre> Single incomingp; if (phase == specialPhase[set]) { // Odd accumulators - // result propagation if (engineEnable[engine + 1]) { // Engine behind me is enabled - propagate // partial sum from its even accumulator // into our odd accumulator incomingp = \$AACC[(k + 2) * TFPU_AACC_PER_AMP_UNIT]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Consume new partial-sum inputs // into our odd accumulator if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } \$AACC[(k * 2) + 1] = incomingp; } else { // phase != specialPhase if ((phase & 1) == propagateOddAccts[set]) { // Even phases (non special phase) // Odd accumulators - // propagate partial-sum inputs if (engineEnable[engine + 1]) { // Engine behind me is enabled incomingp = \$AACC[((k + 2) * TFPU_AACC_PER_AMP_UNIT) + 1]; } else { // Engines behind me are disabled (or I am the final // engine in the chain). Use new partial-sum inputs if (isnan(op2[k & 1])) { incomingp = TFPU_F32_QNan(); } else { incomingp = op2[k & 1]; } } } \$AACC[(k * TFPU_AACC_PER_AMP_UNIT) + 1] = incomingp; } } } // Use \$TAS as a single tick delay for the input to AMP set 1 \$TAS = TFPU_BitsFromF32(op1, false); // Final output \$aDst0:Dst0+1 = { TFPU_BitsFromF32(isnan(out[0]) ? TFPU_F32_QNan() : out[0]), TFPU_BitsFromF32(isnan(out[1]) ? TFPU_F32_QNan() : out[1]) }; </pre>	

Architectural state references: *\$AACC*, *\$TAS*, *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_SNaNCheck*, *TFPU_DoMulPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_Add*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_F32_QNan*, *TFPU_BitsFromF32*

3.7.3.7.28 *f32sisov2slic*

Single-precision floating point **slim** convolution. Input partial-sums are *single-precision*. Results are *single-precision*.

4 output channels are processed/produced

Table 3.191: *f32sisov2slic* instruction definition

<i>f32sisov2slic</i>	worker	aux
Syntax	<code>f32sisov2slic \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags</code>	
Semantics Prepare	<pre> Single op1 = TFPU_F32FromBits(\$aSrc0); array<Single,2> op2 = { TFPU_F32FromBits(\$aSrc1:Src1+1[0]), TFPU_F32FromBits(\$aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> result; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> incomingp; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> input; array<Single,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> weight; array<bool,TFPU_AMP_UNITS_PER_SET * TFPU_NUM_AMP_SETS> specialCase; // AMP set arrangement unsigned ioPhase[2] = {0, 1}; unsigned setPhaseLag[2] = {0, 1}; // Extract immediate config unsigned phase = F32SLIC_ENUMFLAGS__PH__GET(op3); unsigned ampSets = 2; unsigned ampUnits = TFPU_AMP_UNITS_PER_SET * ampSets; // Engine enables uint32_t ee = F32SLIC_ENUMFLAGS__EE__GET(op3); vector<bool> engineEnable = { true, // Engine 0 always enabled ee > 0, ee > 1, ee > 2, false }; // End of the chain unsigned wid = F32SLIC_ENUMFLAGS__WID__GET(op3); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { unsigned set = u / TFPU_AMP_UNITS_PER_SET; unsigned lPhase = (phase - setPhaseLag[set]) & 0x1; // Weight selection uint64_t cwei = context.getCCCSState((u * 4) + wid); weight[u] = TFPU_F32FromBits(cwei >> (32 * lPhase)); // Input selection input[u] = (set == 0) ? op1 : TFPU_F32FromBits(\$TAS); </pre>	

Continued on next page

Table 3.191: *f32sisov2slic* instruction definition (continued)

<i>f32sisov2slic</i>		worker	aux
Syntax	<code>f32sisov2slic \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags</code>		
Semantics	Prepare cont'd	<pre> if (phase == ioPhase[set]) { if (engineEnable[engine + 1]) { // There is an engine behind me and it // is enabled. Use its current accumulator // value incomingp[u] = \$AACC[(u + 2) * TFPU_AACC_PER_AMP_UNIT]; } else { // Engines behind me are disabled // (or I am the final engine in the chain). // Use the new partial-sum inputs incomingp[u] = op2[u & 1]; } } else { // Accumulate into the current value incomingp[u] = \$AACC[u * TFPU_AACC_PER_AMP_UNIT]; } } } // Output exception check unsigned oSet = (phase == ioPhase[0]) ? 0 : 1; unsigned aaccIndex = oSet * TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT; vector<Single> out = { \$AACC[aaccIndex], \$AACC[aaccIndex + 2] }; </pre>	
Except In		<pre> uint32_t fpExcpt = TFPEXCPT_NONE; fpExcpt = TFPU_SNaNCheck(op2[0]); fpExcpt = TFPU_SNaNCheck(op2[1]); fpExcpt = TFPU_SNaNCheck(op1); fpExcpt = TFPU_SNaNCheck(TFPU_F32FromBits(\$TAS)); for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { fpExcpt = TFPU_SNaNCheck(weight[u]); specialCase[u] = TFPU_DoMulPreExecute(input[u], weight[u], &fpExcpt, &result[u]); } } </pre>	
Compute		<pre> for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { if (!specialCase[u]) { Double res = input[u] * weight[u]; result[u] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); } // Combine internal, incoming partial-result // with result of local dot-product result[u] = TFPU_Add(incomingp[u], result[u], TFPU_FP32); } } </pre>	
Except Out		<pre> // Floating-point result exception check fpExcpt = TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NAN00); \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	

Continued on next page

Table 3.191: *f32sisov2slic* instruction definition (continued)

<i>f32sisov2slic</i>		worker	aux
Syntax	f32sisov2slic \$aDst0:Dst0+1, \$aSrc0, \$aSrc1:Src1+1, enumFlags		
Semantics	Commit	<pre> for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { \$AACC[u * 2] = result[u]; } } // Use \$TAS as a single tick delay for the input to AMP set 1 \$TAS = TFPU_BitsFromF32(op1, false); // Final output processing \$aDst0:Dst0+1 = { TFPU_BitsFromF32(isnan(out[0]) ? TFPU_F32_QNan() : out[0]), TFPU_BitsFromF32(isnan(out[1]) ? TFPU_F32_QNan() : out[1]) }; </pre>	

Architectural state references: *\$TAS*, *\$AACC*, *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_SNaNCheck*, *TFPU_DoMulPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_Add*, *TFPU_AACCReadFlags*, *TFPU_IsMalign*, *TFPU_BitsFromF32*, *TFPU_F32_QNan*

3.7.3.7.29 f32sqrt

Computes the square root of a single precision floating point register source.

Table 3.192: *f32sqrt* instruction definition

<i>f32sqrt</i>		worker	aux
Syntax	f32sqrt \$aDst0, \$aSrc0		
Semantics	Prepare	Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;	
	Except In	<pre> uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoSqrtPreExecute(op1, &fpExcpt, &result); </pre>	
	Compute	<pre> TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Sqrt(op1, rmode); } </pre>	
	Except Out	<pre> \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); } </pre>	
	Commit	<pre> \$aDst0 = TFPU_BitsFromF32(result); </pre>	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_DoSqrtPreExecute*, *TFPU_F32Sqrt*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.7.30 f32sub

Subtracts two floating point values.

Table 3.193: *f32sub* instruction definition

<i>f32sub</i>	worker	aux
Syntax	f32sub \$aDst0, \$aSrc0, \$aSrc1	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single op2 = TFPU_F32FromBits(\$aSrc1); Single result;</p> <p>Except In op2 = -op2; uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);</p> <p>Compute if (!specialCase) { double res = static_cast<double>(op1) + static_cast<double>(op2); result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN); fpExcpt = TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NAN00); }</p> <p>Except Out \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_STS*, *\$FP_CTL*

Function references: *TFPU_F32FromBits*, *TFPU_DoAddPreExecute*, *TFPU_RoundFP64ToFmt*, *TFPU_GenOFLOCheck*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.3.7.31 f32tanh

Single-precision floating-point hyperbolic tangent.

Table 3.194: *f32tanh* instruction definition

<i>f32tanh</i>	worker	aux
Syntax	f32tanh \$aDst0, \$aSrc0	
Semantics	<p>Prepare Single op1 = TFPU_F32FromBits(\$aSrc0); Single result;</p> <p>Except In uint32_t fpExcpt = TFPEXCPT_NONE; bool specialCase = TFPU_DoTanhPreExecute(op1, &fpExcpt, &result);</p> <p>Compute TileRoundMode_t rmode = \$FP_CTL.RND; if (!specialCase) { result = TFPU_F32Tanh(op1, rmode); }</p> <p>Except Out \$FP_STS = \$FP_STS fpExcpt; if (TFPU_IsMalign(fpExcpt, \$FP_CTL)) { EXCEPT(TEXCPT_FP); }</p> <p>Commit \$aDst0 = TFPU_BitsFromF32(result);</p>	

Architectural state references: *\$FP_CTL*, *\$FP_STS*

Function references: *TFPU_F32FromBits*, *TFPU_DoTanhPreExecute*, *TFPU_F32Tanh*, *TFPU_IsMalign*, *TFPU_BitsFromF32*

3.7.4 Integer

Table 3.195: integer instructions summary

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>abs</i>	✓	✓	✓	✓	✗	Absolute value of signed 32-bit integer
<i>add</i>	✓	✓	✓	✓	✗	Integer addition
<i>cmpeq</i>	✓	✓	✓	✓	✗	Equality test
<i>cmpne</i>	✓	✓	✓	✓	✗	Inequality test
<i>cmpslt</i>	✓	✓	✓	✓	✗	Signed less-than test
<i>cmpult</i>	✓	✓	✓	✓	✗	Unsigned less-than test
<i>max</i>	✓	✓	✓	✓	✗	Maximum
<i>min</i>	✓	✓	✓	✓	✗	Minimum
<i>movz</i>	✓	✓	✓	✓	✗	Conditional move
<i>mul</i>	✓	✓	✓	✓	✗	Signed multiplication
<i>shl</i>	✓	✓	✓	✓	✗	Logical shift left
<i>shr</i>	✓	✓	✓	✓	✗	Logical shift right
<i>shrs</i>	✓	✓	✓	✓	✗	Signed (arithmetic) shift right
<i>sub</i>	✓	✓	✓	✓	✗	Subtraction
<i>tapack</i>	✓	✗	✓	✓	✗	Triple address pack
<i>urand32</i>	✓	✗	✓	✗	✓	Uniform distribution, 32-bit random integer
<i>urand64</i>	✓	✗	✓	✗	✓	Uniform distribution, 64-bit random integer

3.7.4.1 *abs*

Absolute value of signed 32-bit integer.

Table 3.196: *abs* instruction definition

<i>abs</i>	both	main
Syntax	<code>abs \$mDst0, \$mSrc0</code>	
Semantics	<p>Prepare SignedDataWord op1 = \$mSrc0; SignedDataWord result;</p> <p>Compute <pre>if (op1 == INT32_MIN) { // Result cannot be represented as a signed int result = 0; } else if (op1 < 0) { // op1 is negative result = -op1; } else { // op1 is positive result = op1; }</pre></p> <p>Commit \$mDst0 = result;</p>	

3.7.4.2 *add*

Signed integer addition of 2 source register values, or 1 source register and 1 immediate. immediates may be *sign extended* or *zero extended* to *word* width. No scaling of the source operands (register or immediate) is performed.

Table 3.197: *add* instruction definition

<i>add</i>		both	main
Syntax	<pre>add \$mDst0, \$mSrc0, \$mSrc1 add \$mDst0, \$mSrc0, <i>simm16</i> add \$mDst0, \$mSrc0, <i>zimm16</i></pre>		
Semantics	Prepare	<pre>SignedDataWord op1 = \$mSrc0; SignedDataWord op2 = <(\$mSrc1, <i>zimm16</i>, SignExtend32(<i>simm16</i>, 16))>;</pre>	
	Compute	<pre>SignedDataWord result = op1 + op2;</pre>	
	Commit	<pre>\$mDst0 = result;</pre>	

3.7.4.3 *cmpeq*

Equality comparison of two source values. The destination is set to 1 if the two source operands are equal. Otherwise the destination register is set to 0.

Table 3.198: *cmpeq* instruction definition

<i>cmpeq</i>		both	main
Syntax	<pre>cmpeq \$mDst0, \$mSrc0, \$mSrc1 cmpeq \$mDst0, \$mSrc0, <i>simm16</i> cmpeq \$mDst0, \$mSrc0, <i>zimm16</i></pre>		
Semantics	Prepare	<pre>DataWord op1 = \$mSrc0; DataWord op2 = <(\$mSrc1, <i>zimm16</i>, SignExtend32(<i>simm16</i>, 16))>; DataWord result;</pre>	
	Compute	<pre>if (op1 == op2) { result = 0x1; } else { result = 0x0; }</pre>	
	Commit	<pre>\$mDst0 = result;</pre>	

3.7.4.4 *cmpne*

Inequality comparison of two source values. The destination is set to 0 if the two source operands are equal. Otherwise the destination register is set to 1.

Table 3.199: *cmpne* instruction definition

<i>cmpne</i>		both	main
Syntax	<pre>cmpne \$mDst0, \$mSrc0, \$mSrc1</pre>		
Semantics	Prepare	<pre>DataWord op1 = \$mSrc0; DataWord op2 = \$mSrc1; DataWord result;</pre>	
	Compute	<pre>if (op1 != op2) { result = 0x1; } else { result = 0x0; }</pre>	
	Commit	<pre>\$mDst0 = result;</pre>	

3.7.4.5 `cmpslt`

Less than comparison of two **signed** source values. Destination register is set to 1 if the first source operand is less than the second. Otherwise the destination register is set to 0. The comparison operation is **signed**.

Table 3.200: `cmpslt` instruction definition

<code>cmpslt</code>		both	main
Syntax	<code>cmpslt \$mDst0, \$mSrc0, \$mSrc1</code> <code>cmpslt \$mDst0, \$mSrc0, <i>simm16</i></code>		
Semantics	Prepare	<code>SignedDataWord op1 = \$mSrc0;</code> <code>SignedDataWord op2 = <(\$mSrc1, SignExtend32(<i>simm16</i>, 16))>;</code> <code>DataWord result;</code>	
	Compute	<code>if (op1 < op2) {</code> <code> result = 1;</code> <code>} else {</code> <code> result = 0;</code> <code>}</code>	
	Commit	<code>\$mDst0 = result;</code>	

3.7.4.6 `cmpult`

Less than comparison of two **unsigned** source values. Destination register is set to 1 if the first source operand value is less than the second. Otherwise the destination register is set to 0. The comparison operation is **unsigned**.

Table 3.201: `cmpult` instruction definition

<code>cmpult</code>		both	main
Syntax	<code>cmpult \$mDst0, \$mSrc0, \$mSrc1</code> <code>cmpult \$mDst0, \$mSrc0, <i>zimm16</i></code>		
Semantics	Prepare	<code>DataWord op1 = \$mSrc0;</code> <code>DataWord op2 = <(\$mSrc1, <i>zimm16</i>)>;</code> <code>DataWord result;</code>	
	Compute	<code>if (op1 < op2) {</code> <code> result = 1;</code> <code>} else {</code> <code> result = 0;</code> <code>}</code>	
	Commit	<code>\$mDst0 = result;</code>	

3.7.4.7 `max`

Select the maximum of 1 signed register source value and 1 signed register or *sign extended/zero extended/zero tailed* immediate value.

Table 3.202: *max* instruction definition

<i>max</i>		both	main
Syntax	max <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> max <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>simm16</i> max <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>zimm16</i>		
Semantics	Prepare	SignedDataWord op1 = <i>\$mSrc0</i> ; SignedDataWord op2 = <(<i>\$mSrc1</i> , <i>zimm16</i> , SignExtend32(<i>simm16</i> , 16))>; SignedDataWord result;	
	Compute	<pre>if (op1 > op2) { result = op1; } else { result = op2; }</pre>	
	Commit	<i>\$mDst0</i> = result;	

3.7.4.8 min

Select the minimum of 2 signed integer values. immediates may be *sign extended*, *zero extended* or *zero tailed*.

Table 3.203: *min* instruction definition

<i>min</i>		both	main
Syntax	min <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> min <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>simm16</i> min <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>zimm16</i>		
Semantics	Prepare	SignedDataWord op1 = <i>\$mSrc0</i> ; SignedDataWord op2 = <(<i>\$mSrc1</i> , <i>zimm16</i> , SignExtend32(<i>simm16</i> , 16))>; SignedDataWord result;	
	Compute	<pre>if (op1 < op2) { result = op1; } else { result = op2; }</pre>	
	Commit	<i>\$mDst0</i> = result;	

3.7.4.9 movz

Conditional copy of one register into another, gated on the value of a third.

Table 3.204: *movz* instruction definition

<i>movz</i>		both	main
Syntax	movz <i>\$mSrcDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i>		
Semantics	Prepare	DataWord op2 = <i>\$mSrc1</i> ; DataWord op1 = <i>\$mSrcDst0</i> ; DataWord op0 = <i>\$mSrc0</i> ;	
	Compute	DataWord result = ((op0 != 0) ? op2 : op1);	
	Commit	<i>\$mSrcDst0</i> = result;	

3.7.4.10 mul

Multiply a signed 32-bit register source value with a signed 32-bit register or sign extended 16-bit immediate.

Table 3.205: *mul* instruction definition

<i>mul</i>		both	main
Syntax	mul <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> mul <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>simm16</i>		
Semantics	Prepare	SignedDataWord op1 = <i>\$mSrc0</i> ; SignedDataWord op2 = <(<i>\$mSrc1</i> , SignExtend32(<i>simm16</i> , 16))>;	
	Compute	// 32 x 32 -> 32 DataWord result = op1 * op2;	
	Commit	<i>\$mDst0</i> = result;	

3.7.4.11 shl

Perform a logical left shift, of up-to 31-bits, on a register value.

Table 3.206: *shl* instruction definition

<i>shl</i>		both	main
Syntax	shl <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> shl <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>zimm12</i>		
Semantics	Prepare	DataWord op1 = <i>\$mSrc0</i> ; DataWord op2 = <(<i>\$mSrc1</i> , <i>zimm12</i>)>;	
	Compute	DataWord result = op1 << (op2 % 32);	
	Commit	<i>\$mDst0</i> = result;	

shl occurs in the following code examples:

- *ldb16b16 example*

3.7.4.12 shr

Perform a logical right shift, of up-to 31-bits, on a register value.

Table 3.207: *shr* instruction definition

<i>shr</i>		both	main
Syntax	shr <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>\$mSrc1</i> shr <i>\$mDst0</i> , <i>\$mSrc0</i> , <i>zimm12</i>		
Semantics	Prepare	DataWord op1 = <i>\$mSrc0</i> ; DataWord op2 = <(<i>\$mSrc1</i> , <i>zimm12</i>)>;	
	Compute	DataWord result = op1 >> (op2 % 32);	
	Commit	<i>\$mDst0</i> = result;	

3.7.4.13 shrs

Perform an arithmetic right shift (the sign-bit is shifted in), of up-to 31-bits, on a register value.

Table 3.208: *shrs* instruction definition

<i>shrs</i>		both	main
Syntax	<pre>shrs \$mDst0, \$mSrc0, \$mSrc1 shrs \$mDst0, \$mSrc0, zimm12</pre>		
Semantics	Prepare	<pre>SignedDataWord op1 = \$mSrc0; DataWord op2 = <(\$mSrc1, zimm12)>;</pre>	
	Compute	<pre>SignedDataWord result = op1 >> (op2 % 32);</pre>	
	Commit	<pre>\$mDst0 = result;</pre>	

3.7.4.14 sub

Integer subtraction of 1 register value from another or from immediate. Immediates may be *sign extended*, *zero extended* or *zero tailed* to word width.

Table 3.209: *sub* instruction definition

<i>sub</i>		both	main
Syntax	<pre>sub \$mDst0, \$mSrc1, \$mSrc0 sub \$mDst0, simm16, \$mSrc0 sub \$mDst0, zimm16, \$mSrc0</pre>		
Semantics	Prepare	<pre>DataWord op1 = \$mSrc0; DataWord op2 = <(\$mSrc1, zimm16, SignExtend32(simm16, 16))>;</pre>	
	Compute	<pre>DataWord result = op2 - op1;</pre>	
	Commit	<pre>\$mDst0 = result;</pre>	

3.7.4.15 tapack

Convert 3 absolute addresses to the triple-packed address format.

Table 3.210: *tapack* instruction definition

<i>tapack</i>		worker	main
Syntax	<pre>tapack \$mDst0:Dst0+1, \$mAddr0, \$mAddr1, \$mAddr2</pre>		
Semantics	Prepare	<pre>DataWord op0 = \$mAddr0; DataWord op1 = \$mAddr1; DataWord op2 = \$mAddr2; array<DataWord,3> addr;</pre>	
	Compute	<pre>addr[0] = op0 & TMEM_FULL_ADDRESS_MASK; addr[1] = op1 & TMEM_FULL_ADDRESS_MASK; addr[2] = op2 & TMEM_FULL_ADDRESS_MASK;</pre>	
	Commit	<pre>// MRF result values \$mDst0:Dst0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) };</pre>	

Function references: [Tile_TripleAddressPack_Lower](#), [Tile_TripleAddressPack_Upper](#)

tapack occurs in the following code examples:

- [f16v4cmac example](#)
- [f16v4sisoslic example part 1](#)
- [f16v4sisoslic example part 2](#)
- [f16v4stacc example](#)

3.7.4.16 urand32

Uniform distribution, 32-bit random integer.

Table 3.211: *urand32* instruction definition

<i>urand32</i>		worker	aux
Syntax	urand32 \$aDst0		
Semantics	Prepare	array<uint64_t,2> randomBits;	
	Compute	TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits);	
	Commit	// Return the ls 32-bits of the result to the ARF. \$aDst0 = randomBits[0];	

Architectural state references: *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*

3.7.4.17 urand64

Uniform distribution, 64-bit random integer.

Table 3.212: *urand64* instruction definition

<i>urand64</i>		worker	aux
Syntax	urand64 \$aDst0:1		
Semantics	Prepare	array<uint64_t,2> randomBits;	
	Compute	TPRNG_Advance(\$PRNG_0_0, \$PRNG_0_1, \$PRNG_1_0, \$PRNG_1_1, randomBits);	
	Commit	\$aDst0:1 = { randomBits[0], (randomBits[0] >> 32) };	

Architectural state references: *\$PRNG_0_0*, *\$PRNG_0_1*, *\$PRNG_1_0*, *\$PRNG_1_1*

3.7.5 Memory

Table 3.213: memory instructions summary

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>atom</i>	✓	✗	✓	✓	✗	Copy an <i>arf</i> register value to <i>mrf</i>
<i>ld128</i>	✓	✗	✓	✓	✗	Single 128-bit load from interleaved memory region
<i>ld128putcs</i>	✓	✓	✗	✓	✗	128-bit load and put to common configuration space
<i>ld128step</i>	✓	✗	✓	✓	✗	Post-incrementing 128-bit load from interleaved memory region
<i>ld2x64pace</i>	✓	✗	✓	✓	✗	Post-incrementing, dual 64-bit load
<i>ld2xst64pace</i>	✓	✗	✓	✓	✗	Post-incrementing dual 64-bit load with simultaneous 64-bit store
<i>ld32</i>	✓	✓	✓	✓	✗	Single 32-bit load
<i>ld32step</i>	✓	✓	✓	✓	✗	Post-incrementing 32-bit load
<i>ld64</i>	✓	✗	✓	✓	✗	Single 64-bit load
<i>ld64a32</i>	✓	✗	✓	✓	✗	Post-incrementing dense 64-bit plus sparse 32-bit load
<i>ld64a32pace</i>	✓	✗	✓	✓	✗	Post-incrementing dual 64/32-bit load
<i>ld64b16pace</i>	✓	✗	✓	✓	✗	Post-incrementing, dual 64/16-bit load
<i>ld64putcs</i>	✓	✓	✗	✓	✗	64-bit load and put to common configuration space
<i>ld64step</i>	✓	✗	✓	✓	✗	Post-incrementing 64-bit load
<i>ldb16</i>	✓	✗	✓	✓	✗	16-bit load and broadcast
<i>ldb16b16</i>	✓	✗	✓	✓	✗	Post incrementing, lightly-sparse 16-bit with dense 16-bit load
<i>ldb16step</i>	✓	✗	✓	✓	✗	Post-incrementing 16-bit load and broadcast
<i>ldd16a32</i>	✓	✗	✓	✓	✗	Post-incrementing 16-bit delta and 32-bit data load
<i>ldd16a64</i>	✓	✗	✓	✓	✗	Post-incrementing 16-bit delta and 64-bit data load
<i>ldd16b16</i>	✓	✗	✓	✓	✗	Post-incrementing 16-bit delta and broadcast 16-bit data load
<i>ldd16v2a32</i>	✓	✗	✓	✓	✗	Post-incrementing delta-pair plus 32-bit load
<i>lds16</i>	✓	✓	✓	✓	✗	Sign extending 16-bit load
<i>lds16step</i>	✓	✓	✓	✓	✗	Post-incrementing sign-extending 16-bit load
<i>lds8</i>	✓	✓	✓	✓	✗	Sign extending 8-bit load
<i>lds8step</i>	✓	✓	✓	✓	✗	Post-incrementing sign-extending 8-bit load
<i>ldst64pace</i>	✓	✗	✓	✓	✗	Post-incrementing 64-bit load with simultaneous 64-bit store
<i>ldz16</i>	✓	✓	✓	✓	✗	Zero-extending 16-bit load
<i>ldz16step</i>	✓	✓	✓	✓	✗	Post-incrementing zero-extending 16-bit load
<i>ldz8</i>	✓	✓	✓	✓	✗	Zero-extending 8-bit load

Continued on next page

Table 3.213 – continued from previous page

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>ldz8step</i>	✓	✓	✓	✓	✗	Post-incrementing <i>zero extended</i> 8-bit load
<i>st32</i>	✓	✗	✓	✓	✗	32-bit store
<i>st32step</i>	✓	✓	✓	✓	✗	Post-incrementing 32-bit store
<i>st64</i>	✓	✗	✓	✓	✗	64-bit store
<i>st64pace</i>	✓	✗	✓	✓	✗	Post-incrementing 64-bit store, using packed addresses and offsets
<i>st64step</i>	✓	✗	✓	✓	✗	Post-incrementing 64-bit store
<i>stm32</i>	✓	✓	✓	✓	✗	32-bit store from MRF
<i>stm32step</i>	✓	✓	✓	✓	✗	Post-incrementing 32-bit store from MRF

3.7.5.1 Load-store

3.7.5.1.1 *ldst64pace*

Naturally aligned 64-bit load and simultaneous 64-bit store, with dual independent post-incrementing addresses.

Destination register-file: *ARF* only

Source register-file: *ARF* only

Effective addresses:

- independent load and store addresses
- provided directly from *MRF* as a register pair
- lower register provides load address
- store address is split across the upper-bits of both registers (see *tapack*)

Data format:

- Load result is an unmodified 64-bit value stored in a naturally aligned register pair.

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Table 3.214: *ldst64pace* instruction definition

<i>ldst64pace</i>		worker	main
Syntax	<code>ldst64pace \$aDst0:Dst0+1, \$aSrc0:Src0+1, \$mAddr0:Addr0+1+=\$mStride0, Strimm2x2</code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; array<DataWord,2> op3 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; DataWord op4 = Strimm2x2; array<DataWord,3> addr;</pre> <pre>EA[0] = Tile_ExtractPackedAddress(op1, 0); EA[2] = Tile_ExtractPackedAddress(op1, 2);</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[2])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[2] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[2])) { EXCEPT(TEXCPT_CONFLICT); }</pre> <p>Compute</p> <pre>// Post-increment both addresses as specified by the immediate int32_t stride0 = Tile_ExtractPackedStride(op0, op4); int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2)); addr[0] = (EA[0] + (stride0 * 8)) & TMem_FULL_ADDRESS_MASK; addr[1] = Tile_ExtractPackedAddress(op1, 1); // Unchanged addr[2] = (EA[2] + (stride1 * 8)) & TMem_FULL_ADDRESS_MASK;</pre> <p>Memory</p> <pre>uint64_t storeData = (op3[1] << 32ULL) op3[0]; uint64_t data = loadDoubleWord(EA[0]); storeDoubleWord(EA[2], storeData);</pre> <p>Commit</p> <pre>\$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) }; \$aDst0:Dst0+1 = { data & 0xffffffffFULL, (data >> 32ULL) & 0xffffffffFULL };</pre>		

Function references: [Tile_ExtractPackedAddress](#) , [Tile_ExtractPackedStride](#) , [Tile_TripleAddressPack_Lower](#) , [Tile_TripleAddressPack_Upper](#) , [TMem_IsValidAddress](#)

ldst64pace occurs in the following code examples:

-
- *f16v4hihoamp example*

- *f16v4stacc example*

3.7.5.1.2 ld2xst64pace

Naturally aligned dual 64-bit load and simultaneous 64-bit store, with 3 independent post-incrementing addresses.

Destination register-file: *ARF* only

Source register-file: *ARF* only

Effective addresses:

- 3 independent addresses provided directly from *MRF*, packed into a register pair

Data format:

- Load results are 2 unmodified 64-bit values stored in a naturally aligned register quad.

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Table 3.215: *ld2xst64pace* instruction definition

<i>ld2xst64pace</i>		worker	main
Syntax	ld2xst64pace \$aDst0:Dst0+3, \$aSrc0:Src0+1, \$mAddr0:Addr0+1+=\$, \$mStride0, Strimm3x2		
Semantics	Prepare	<pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; array<DataWord,2> op3 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; DataWord op4 = Strimm3x2; array<DataWord,3> addr;</pre> <pre>EA[0] = Tile_ExtractPackedAddress(op1, 0); EA[1] = Tile_ExtractPackedAddress(op1, 1); EA[2] = Tile_ExtractPackedAddress(op1, 2);</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[2])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (TMem_AddressIsExecutable(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[2] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[2])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Post-increment all 3 addresses as specified by the immediate int32_t stride0 = Tile_ExtractPackedStride(op0, op4); int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2)); int32_t stride2 = Tile_ExtractPackedStride(op0, (op4 >> 4)); addr[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[1] = (EA[1] + (stride1 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[2] = (EA[2] + (stride2 * 8)) & TMEM_FULL_ADDRESS_MASK;</pre>	
	Memory	<pre>uint64_t storeData = (op3[1] << 32ULL) op3[0]; uint64_t data0 = loadDoubleWord(EA[0]); uint64_t data1 = loadDoubleWord(EA[1]); storeDoubleWord(EA[2], storeData);</pre>	

Continued on next page

Table 3.215: *ld2xst64pace* instruction definition (continued)

<i>ld2xst64pace</i>		worker	main
Syntax	<i>ld2xst64pace</i> \$aDst0:Dst0+3, \$aSrc0:Src0+1, \$mAddr0:Addr0+1+=\$mStride0, Strimm3x2		
Semantics	Commit	<pre> \$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) }; \$aDst0:Dst0+3 = { data0 & 0xffffffffFULL, (data0 >> 32ULL) & 0xffffffffFULL, data1 & 0xffffffffFULL, (data1 >> 32ULL) & 0xffffffffFULL }; </pre>	

Function references: *Tile_ExtractPackedAddress* , *Tile_ExtractPackedStride* , *Tile_TripleAddressPack_Lower* , *Tile_TripleAddressPack_Upper* , *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

ld2xst64pace occurs in the following code examples:

- *f16v4sisoamp example*
- *f16v4sisoslic example part 1*
- *f16v4sisoslic example part 2*
- *f32sisoamp example*

3.7.5.2 Multi-load

3.7.5.2.1 Idd16b16

Post-incrementing 16-bit delta load with simultaneous broadcast 16-bit data load.

Destination register-file: Combination of *MRF* and *ARF*

Effective addresses:

1. A full-pointer value (\$m register)
2. Base address (\$m register) plus 16-bit, unsigned address delta (\$m register)

Data format:

1. A 16-bit value (new delta-offset) written to the MRF delta register.
2. A 32-bit value formed via a broadcast operation on the 16-bit loaded data value written to the ARF destination register.

Address auto-increment:

1. The full-pointer source register value is incremented by 2 (bytes).

Table 3.216: *ldd16b16* instruction definition

<i>ldd16b16</i>		worker	main
Syntax	<code>ldd16b16 \$aDst0, \$mAddr0++, \$mBase0, \$mDelta0@</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mAddr0; array<HalfDataWord,2> op2 = { ((\$mDelta0[0] >> 0) & 0xffff), ((\$mDelta0[0] >> 16) & 0xffff) }; uint16_t delta = op2[0]; EA[0] = op0 + delta; EA[1] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<code>DataWord nextDeltaAddress = op1 + 2;</code>	
	Memory	<pre>uint16_t data = loadHalf(EA[0]); uint16_t newDelta = loadHalf(EA[1]);</pre>	
	Commit	<pre>\$mAddr0 = nextDeltaAddress; \$mDelta0 = { ((newDelta & 0xffff) << 0) 0 }; \$aDst0 = { ((data & 0xffff) << 0) ((data & 0xffff) << 16) };</pre>	

Function references: [*TMem_IsValidAddress*](#)

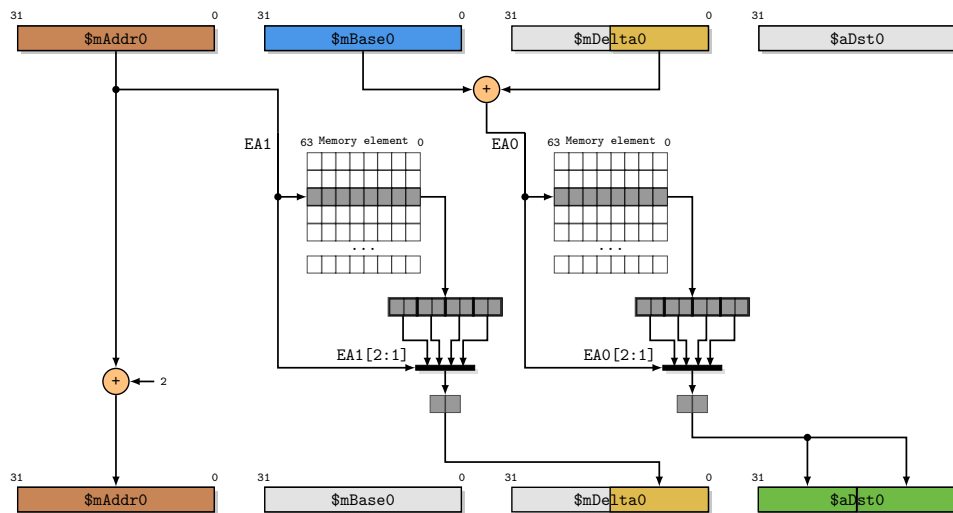


Fig. 3.28: ldd16b16

3.7.5.2.2 ldd16a32

Post-incrementing 16-bit delta load with simultaneous 32-bit data load.

Destination register-file: Combination of *MRF* and *ARF*

Effective addresses:

1. A full-pointer value (\$m register)
2. Base address (\$m register) plus 16-bit, unsigned address delta (\$m register)

Data format:

1. A 16-bit value (new delta-offset) written to the MRF delta register.
2. A 32-bit value written to the ARF destination register.

Address auto-increment:

1. The full-pointer source register value is incremented by 2 (bytes).

Table 3.217: *ldd16a32* instruction definition

<i>ldd16a32</i>		worker	main
Syntax	<code>ldd16a32 \$aDst0, \$mAddr0++, \$mBase0, \$mDelta0@</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mAddr0; array<HalfDataWord,2> op2 = { ((\$mDelta0[0] >> 0) & 0xffff), ((\$mDelta0[0] >> 16) & 0xffff) }; uint16_t delta = op2[0]; EA[0] = op0 + delta; EA[1] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<code>DataWord nextDeltaAddress = op1 + 2;</code>	
	Memory	<pre>uint32_t data = loadWord(EA[0]); uint16_t newDelta = loadHalf(EA[1]);</pre>	
	Commit	<pre>\$mAddr0 = nextDeltaAddress; \$mDelta0 = { ((newDelta & 0xffff) << 0) 0 }; \$aDst0 = data;</pre>	

Function references: [*TMem_IsValidAddress*](#)

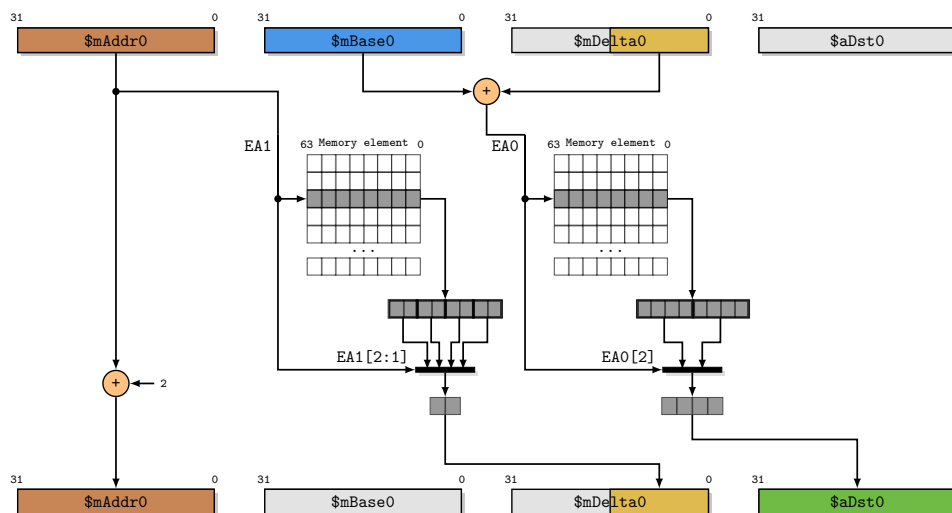


Fig. 3.29: ldd16a32

3.7.5.2.3 ldd16v2a32

Post incrementing load of dense 16-bit delta-pair plus a sparse 32-bit data value.

Destination register-file: *ARF* only

Effective addresses:

1. A full-pointer register value (notionally a pointer into an array of 16-bit deltas)
2. A base address register value added to a 16-bit delta-offset located in the msbs of a third source register value.

Data format:

- Results are:
 - A new pair of 16-bit deltas
 - A naturally aligned 32-bit data value, written to the destination register

Address auto-increment:

- The full pointer register is post-incremented by 4 (bytes)

Table 3.218: *ldd16v2a32* instruction definition

<i>ldd16v2a32</i>		worker	main
Syntax	<code>ldd16v2a32 \$aDst0, \$mAddr0++, \$mBase0, \$mDelta0@</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mAddr0; DataWord op2 = \$mDelta0; uint16_t delta = (op2 >> 16) & 0xffff; EA[0] = op0 + delta; EA[1] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<code>DataWord nextAddr = op1 + 4; // Increment ptr address</code>	
	Memory	<pre>uint32_t data = loadWord(EA[0]); uint32_t newDelta = loadWord(EA[1]);</pre>	
	Commit	<pre>\$mAddr0 = nextAddr; \$mDelta0 = newDelta; \$aDst0 = data;</pre>	

Function references: *TMem_IsValidAddress*

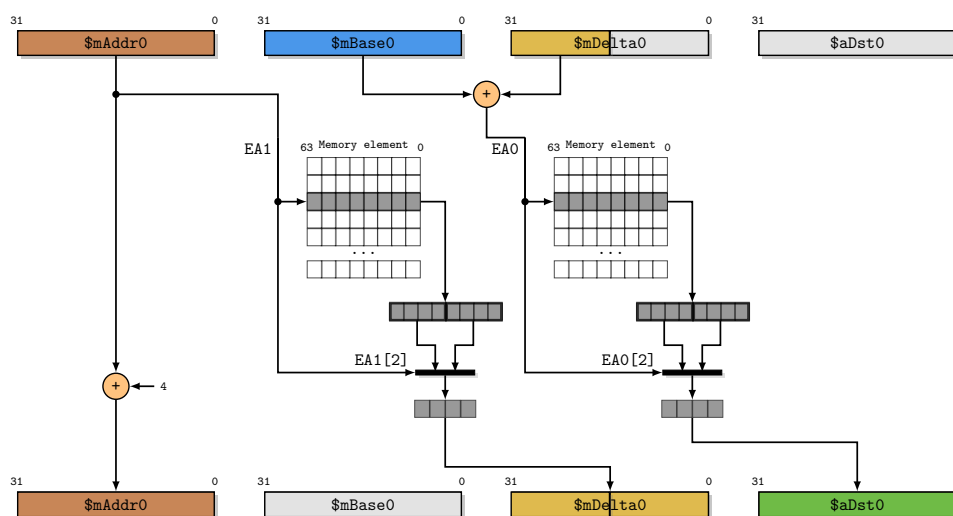


Fig. 3.30: *ldd16v2a32*

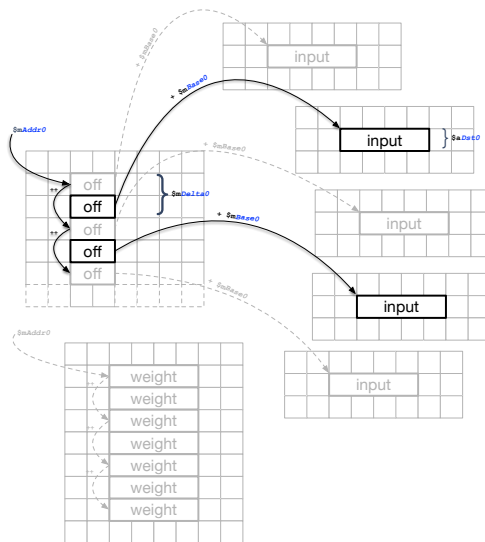


Fig. 3.31: ldd16v2a32

ldd16v2a32 occurs in the following code examples:

- *f32mac example*

3.7.5.2.4 ldd16a64

Post-incrementing 16-bit delta load with simultaneous 64-bit data load.

Destination register-file: Combination of *MRF* and *ARF*

Effective addresses:

1. A full-pointer value ($\$m$ register)
2. Base address ($\$m$ register) plus 16-bit, unsigned address delta ($\$m$ register)

Data format:

1. A 16-bit value (new delta-offset) written to the MRF delta register.
2. A 64-bit value written to the ARF destination register pair.

Address auto-increment:

1. The full-pointer source register value is incremented by 2 (bytes).

Table 3.219: *ldd16a64* instruction definition

<i>ldd16a64</i>		worker	main
Syntax	<code>ldd16a64 \$aDst0:Dst0+1, \$mAddr0++, \$mBase0, \$mDelta0@</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mAddr0; array<HalfDataWord,2> op2 = { ((\$mDelta0[0] >> 0) & 0xffff), ((\$mDelta0[0] >> 16) & 0xffff) }; uint16_t delta = op2[0]; EA[0] = op0 + delta; EA[1] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<code>DataWord nextDeltaAddress = op1 + 2;</code>	
	Memory	<pre>uint64_t data = loadDoubleWord(EA[0]); uint16_t newDelta = loadHalf(EA[1]);</pre>	
	Commit	<pre>\$mAddr0 = nextDeltaAddress; \$mDelta0 = { ((newDelta & 0xffff) << 0) 0 }; \$aDst0:Dst0+1 = { data & 0xffffffffFULL, (data >> 32ULL) & 0xffffffffFULL };</pre>	

Function references: [*TMem_IsValidAddress*](#)

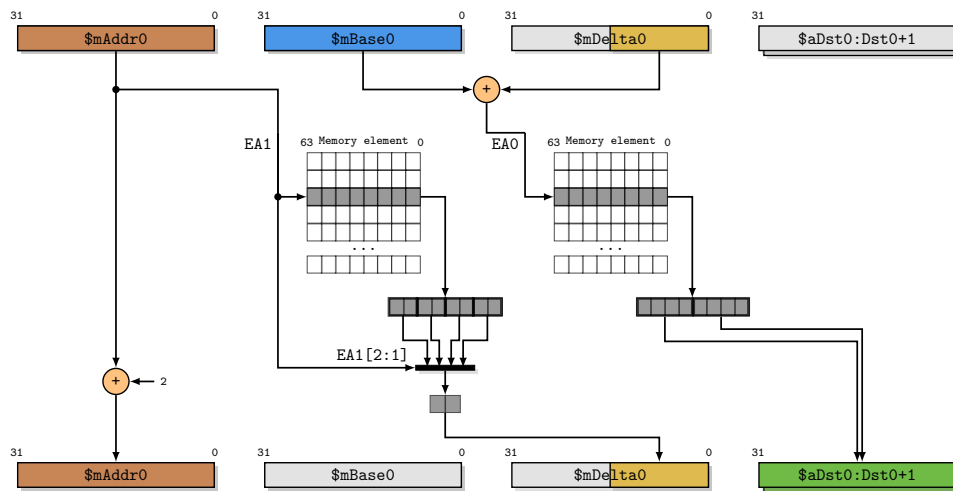


Fig. 3.32: ldd16a64

3.7.5.2.5 ldb16b16

Broadcast 16-bit load from base + 16-bit delta-offset with 2nd broadcast 16-bit load from base plus 2nd 16-bit delta-offset.

Destination register-file: *ARF* only

Effective addresses:

- 2 load addresses provided directly from *MRF* as a common base register plus independent 16-bit delta-offsets (packed into a single *MRF* register)

Data format:

- Results are 2 x *f16v2* values stored in a naturally aligned register pair. Each *f16v2* vector is created from a broadcast operation on a single 16-bit value loaded from *Tile Memory*.

Address auto-increment:

- The two 16-bit delta-offsets are post-incremented independently:
 - One is incremented by 2 (bytes)
 - The other is incremented according to the 4-bit mini-delta value in the lsb of the 4th operand.
- The mini-delta operand is also right-shifted by 4.

Table 3.220: *ldb16b16* instruction definition

<i>ldb16b16</i>		worker	main
Syntax	ldb16b16 \$aDst0:Dst0+1, \$mBase0, \$mDelta0++, \$mMiniD0>>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mMiniD0; array<HalfDataWord,2> op2 = { ((\$mDelta0[0] >> 0) & 0xffff), ((\$mDelta0[0] >> 16) & 0xffff) }; uint16_t dataDelta = op2[0]; uint16_t weightDelta = op2[1]; EA[0] = op0 + weightDelta; EA[1] = op0 + dataDelta;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>uint16_t miniDelta = ((op1 & 0xf) + 1) * 2; // Increment weight and data deltas HalfDataWord nextDataOffset = dataDelta + miniDelta; HalfDataWord nextWeightOffset = weightDelta + 2; // Consume 4-bit delta DataWord nextMiniDelta = op1 >> 4;</pre>	
	Memory	<pre>uint16_t denseData = loadHalf(EA[0]); uint16_t sparseData = loadHalf(EA[1]);</pre>	
	Commit	<pre>\$mMiniD0 = nextMiniDelta; \$mDelta0 = { ((nextDataOffset & 0xffff) << 0) ((nextWeightOffset & 0xffff) << 16) }; \$aDst0:Dst0+1 = { ((denseData & 0xffff) << 0) ((denseData & 0xffff) << 16), ((sparseData & 0xffff) << 0) ((sparseData & 0xffff) << 16) };</pre>	

Function references: [TMem_IsValidAddress](#)

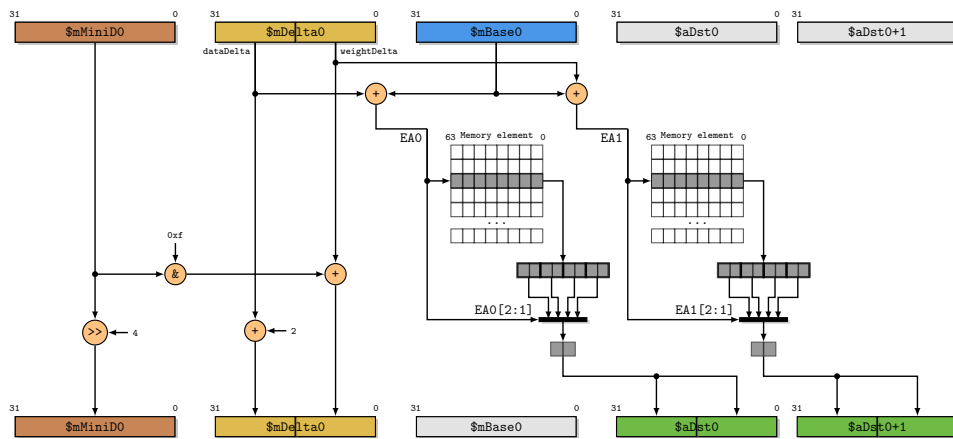


Fig. 3.33: ldb16b16

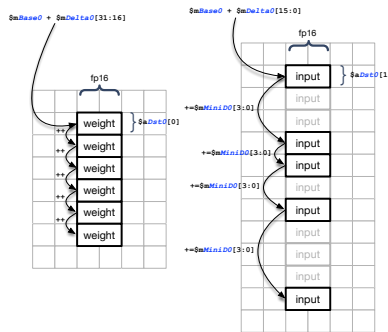


Fig. 3.34: ldb16b16

ldb16b16 occurs in the following code examples:

- *ldb16b16* example

Listing 3.13: ldb16b16 example

```

// Combine the two offsets into a single register
shl    $offsets, $weightOffset, 16
or     $offsets, $offsets, $inputOffset

// Initialise the weight and data to zero as they are used
// before they are loaded in the first f16v2cmac
setzi  $weight, 0
setzi  $data, 0

// There are 16 words of data to work through in this example and each
// repeat works through 8
ldconst $numRepeats, (16 / 8)

```

```

.align 8
{
  rpt          $numRepeats, ((_loop_end - _loop_start) / 8) - 1
  fnop
}
// Consume 8 x 4-bit mini-deltas per rpt loop
_loop_start:
{
  // Load next mini-delta set
  ld32step    $deltas, $ptrBase, $deltaOffset++, 1
  fnop
}
.rept 8
{
  ldb16b16    $weightAndData, $ptrBase, $offsets++, $deltas>>
  f16v2cmac   $weight, $data
}
.endr
_loop_end:

// Final fmac
f16v2cmac    $weight, $data

// Read out $AACC[0]
f32v2gina    $a0:1, $a14:15, 0

// Divide the result by 2 because the loads are broadcasting the result to
// both halves
ldconst     $a1, 2
f32fromui32  $a1, $a1
f32div      $a0, $a0, $a1

```

3.7.5.2.6 ld64a32

Post-incrementing load of dense 64-bit value plus a sparse 32-bit value.

Destination register-file: *ARF* only

Effective addresses:

1. A full-pointer register value (notionally a pointer into an array of dense values).
2. A base address register value added to a 16-bit delta-offset located in the lsbs of a third source register value.

Data format:

- Results are:
 - A naturally aligned 64-bit value, written to the top half of the destination register quad.
 - A naturally aligned 32-bit data value, written to the 2nd element of the destination register quad.

Address auto-increment:

- The full-pointer register is post-incremented by 8 (bytes)

Note: The first element of the destination register quad is unmodified.

Table 3.221: *ld64a32* instruction definition

<i>ld64a32</i>		worker	main
Syntax	<i>ld64a32 \$aDst0+1:Dst0+3, \$mAddr0++, \$mBase0, \$mDelta0</i>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mAddr0; HalfDataWord op2 = ((\$mDelta0 >> 0) & 0xffff); DataWord delta = op2; EA[0] = op0 + delta; EA[1] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Increment ptr address DataWord nextAddr = op1 + 8;</pre>	
	Memory	<pre>uint32_t sparseData = loadWord(EA[0]); uint64_t weightsPair = loadDoubleWord(EA[1]);</pre>	
	Commit	<pre>\$mAddr0 = nextAddr; \$aDst0+1:Dst0+3 = { sparseData, weightsPair & 0xffffffffFULL, (weightsPair >> 32ULL) & 0xffffffffFULL };</pre>	

Function references: *TMem_IsValidAddress*

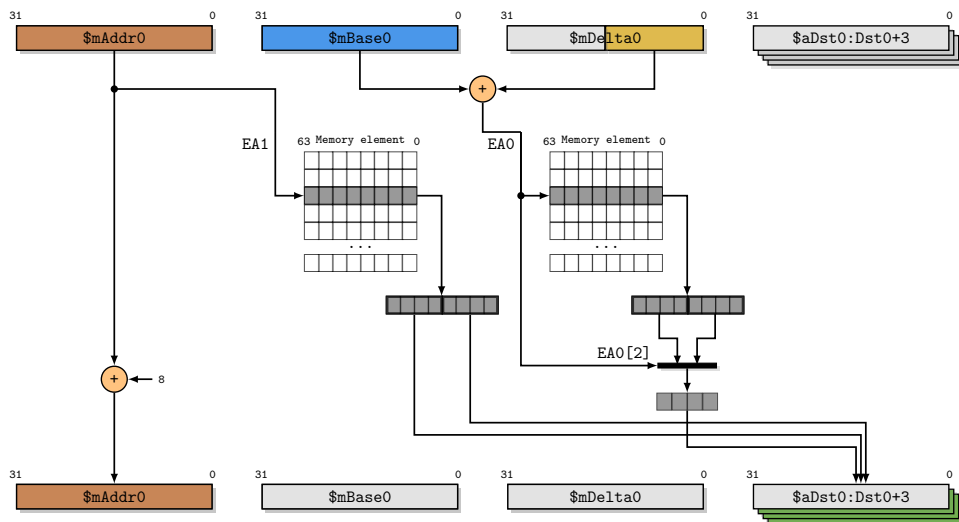


Fig. 3.35: `ld64a32`

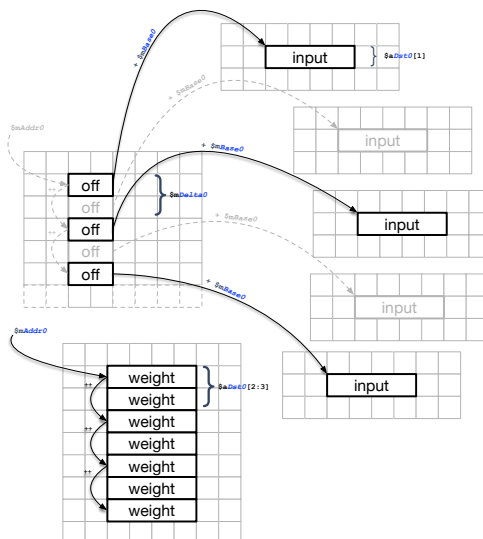


Fig. 3.36: `ld64a32`

`ld64a32` occurs in the following code examples:

- *f32mac example*

3.7.5.2.7 `ld64b16pace`

Naturally aligned 64-bit and broadcast 16-bit load, with dual independent post-incrementing addresses.

Destination register-file: *ARF* only

Effective addresses:

- 2 independent full load addresses
- provided directly from *MRF* as a register pair

-
- lower register provides 1st load address
 - upper register provides 2nd load address

Data format:

- Results are:
 - 1 unmodified 64-bit value stored in a naturally aligned register pair
 - 1 16-bit value broadcast (duplicated) into a single ARF register

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Note that a `TEXCPT_INVALID_OP` exception will occur if the two destination register pairs are not distinct.

Table 3.222: *ld64b16pace* instruction definition

<i>ld64b16pace</i>		worker	main
Syntax	<i>ld64b16pace</i> \$aDst0:Dst0+1, \$aDst1, \$mAddr0:Addr0+1+=\$, \$mStride0, Strimm2x2		
Semantics	Prepare	<pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; DataWord op4 = Strimm2x2; array<DataWord,3> addr;</pre> <pre>EA[0] = Tile_ExtractPackedAddress(op1, 0); EA[1] = Tile_ExtractPackedAddress(op1, 1);</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Post-increment both addresses as specified by the immediate int32_t stride0 = Tile_ExtractPackedStride(op0, op4); int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2)); addr[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[1] = (EA[1] + (stride1 * 2)) & TMEM_FULL_ADDRESS_MASK; addr[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged</pre>	
	Memory	<pre>uint64_t data0 = loadDoubleWord(EA[0]); uint16_t data1 = loadHalf(EA[1]);</pre>	
	Commit	<pre>\$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) }; \$aDst0:Dst0+1 = { data0 & 0xffffffffFULL, (data0 >> 32ULL) & 0xffffffffFULL }; \$aDst1 = { ((data1 & 0xffff) << 0) ((data1 & 0xffff) << 16) };</pre>	

Function references: *Tile_ExtractPackedAddress* , *Tile_ExtractPackedStride* , *Tile_TripleAddressPack_Lower* , *Tile_TripleAddressPack_Upper* , *TMem_IsValidAddress*

3.7.5.2.8 *ld64a32pace*

Naturally aligned dual 64/32-bit load, with dual independent post-incrementing addresses.

Destination register-file: *ARF* only

Effective addresses:

- 2 independent load addresses
- provided directly from *MRF* as a register pair
- lower register provides 1st load address
- upper register provides 2nd load address

Data format:

- Results are:
 - 1 unmodified 64-bit value stored in a naturally aligned register pair
 - 1 unmodified 32-bit value stored in a single *ARF* register

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Note that a `TEXCPT_INVALID_OP` exception will occur if the two destination register pairs are not distinct.

Table 3.223: *ld64a32pace* instruction definition

<i>ld64a32pace</i>		worker	main
Syntax	<i>ld64a32pace</i> \$aDst0:Dst0+1, \$aDst1, \$mAddr0:Addr0+1+=\$mStride0, Strimm2x2		
Semantics	Prepare	<pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; DataWord op4 = Strimm2x2; array<DataWord,3> addr;</pre> <pre>EA[0] = Tile_ExtractPackedAddress(op1, 0); EA[1] = Tile_ExtractPackedAddress(op1, 1);</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Post-increment both addresses as specified by the immediate int32_t stride0 = Tile_ExtractPackedStride(op0, op4); int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2)); addr[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[1] = (EA[1] + (stride1 * 4)) & TMEM_FULL_ADDRESS_MASK; addr[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged</pre>	
	Memory	<pre>uint64_t data0 = loadDoubleWord(EA[0]); uint32_t data1 = loadWord(EA[1]);</pre>	
	Commit	<pre>\$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) }; \$aDst0:Dst0+1 = { data0 & 0xffffffffFULL, (data0 >> 32ULL) & 0xffffffffFULL }; \$aDst1 = data1;</pre>	

Function references: [Tile_ExtractPackedAddress](#) , [Tile_ExtractPackedStride](#) , [Tile_TripleAddressPack_Lower](#) , [Tile_TripleAddressPack_Upper](#) , [TMem_IsValidAddress](#)

3.7.5.2.9 ld2x64pace

Naturally aligned dual 64-bit load, with dual independent post-incrementing addresses.

Destination register-file: *ARF* only

Effective addresses:

-
- 2 independent load addresses
 - provided directly from *MRF* as a register pair
 - lower register provides 1st load address
 - upper register provides 2nd load address

Data format:

- Results are 2 unmodified 64-bit values stored in 2 naturally aligned register pairs.

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Note that a `TEXCPT_INVALID_OP` exception will occur if the two destination register pairs are not distinct.

Table 3.224: *ld2x64pace* instruction definition

<i>ld2x64pace</i>		worker	main
Syntax	<code>ld2x64pace</code>	<code>\$aDst0:Dst0+1, \$aDst1:Dst1+1, \$mAddr0:Addr0+1+=, \$mStride0, Strimm2x2</code>	
Semantics	Prepare	<pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; DataWord op4 = Strimm2x2; array<DataWord,3> addr;</pre> <pre>EA[0] = Tile_ExtractPackedAddress(op1, 0); EA[1] = Tile_ExtractPackedAddress(op1, 1);</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (!TMem_IsValidAddress(EA[1])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[1] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } else if (hadMemoryConflict(EA[1])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Post-increment both addresses as specified by the immediate int32_t stride0 = Tile_ExtractPackedStride(op0, op4); int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2)); addr[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[1] = (EA[1] + (stride1 * 8)) & TMEM_FULL_ADDRESS_MASK; addr[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged</pre>	
	Memory	<pre>uint64_t data0 = loadDoubleWord(EA[0]); uint64_t data1 = loadDoubleWord(EA[1]);</pre>	
	Commit	<pre>\$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) }; \$aDst0:Dst0+1 = { data0 & 0xffffffffFULL, (data0 >> 32ULL) & 0xffffffffFULL }; \$aDst1:Dst1+1 = { data1 & 0xffffffffFULL, (data1 >> 32ULL) & 0xffffffffFULL };</pre>	

Function references: [Tile_ExtractPackedAddress](#) , [Tile_ExtractPackedStride](#) , [Tile_TripleAddressPack_Lower](#) , [Tile_TripleAddressPack_Upper](#) , [TMem_IsValidAddress](#)

ld2x64pace occurs in the following code examples:

- [f16v4cmac example](#)
- [f16v4hihoamp example](#)
- [f16v4sisoslic example part 1](#)
- [f16v4sisoslic example part 2](#)
- [f32sisoslic example](#)

3.7.5.3 Single-load

3.7.5.3.1 atom

Copy an *ARF* register value to *MRF*. No format conversion is performed.

Table 3.225: *atom* instruction definition

<i>atom</i>	worker	main
Syntax	atom \$mDst0, \$aSrc0	
Semantics	Prepare	DataWord op1 = \$aSrc0;
	Commit	\$mDst0 = op1;

3.7.5.3.2 ldz8

Load and zero-extend a single, 8-bit quantity from *Tile Memory*.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Data format:

- Result is a 32-bit value formed by zero-extending the 8-bit loaded data value.

Table 3.226: *ldz8* instruction definition

<i>ldz8</i>	both	main
Syntax	ldz8 \$mDst0, \$mBase0, \$mDelta0, \$mOff0 ldz8 \$mDst0, \$mBase0, \$mDelta0, zimm12	
Semantics	Prepare	DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(\$mOff0, zimm12)>; EA[0] = op0 + op1 + op2;
	Except In	if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }
	Memory	uint8_t data = loadByte(EA[0]);
	Commit	\$mDst0 = Tile_ZeroExtend(data, 8);

Function references: [Tile_ZeroExtend](#), [TMem_IsValidAddress](#)

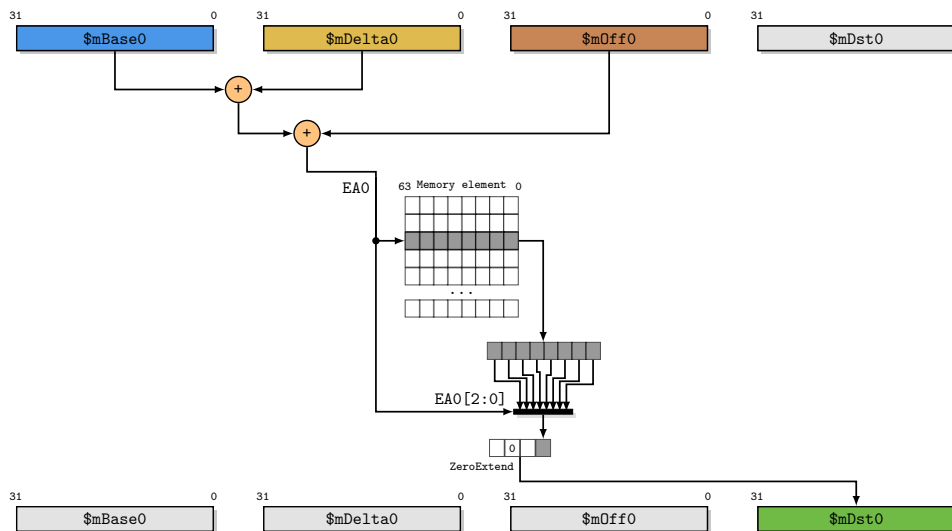


Fig. 3.37: ldz8

3.7.5.3.3 lds8

Load and sign-extend a single, 8-bit quantity from *Tile Memory*.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (*\$m* register)
- Unsigned address delta (*\$m* register)
- Unsigned scaled offset (*\$m* register or immediate)

Data format:

- Result is a 32-bit value formed by sign-extending the 8-bit loaded data value.

Table 3.227: *lds8* instruction definition

<i>lds8</i>	both	main
Syntax	<code>lds8 \$mDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>lds8 \$mDst0, \$mBase0, \$mDelta0, zimm12</code>	
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(\$mOff0, zimm12)>; EA[0] = op0 + op1 + op2;</pre>
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>
	Memory	<code>uint8_t data = loadByte(EA[0]);</code>
	Commit	<code>\$mDst0 = Tile_SignExtend(data, 8);</code>

Function references: *Tile_SignExtend*, *TMem_IsValidAddress*

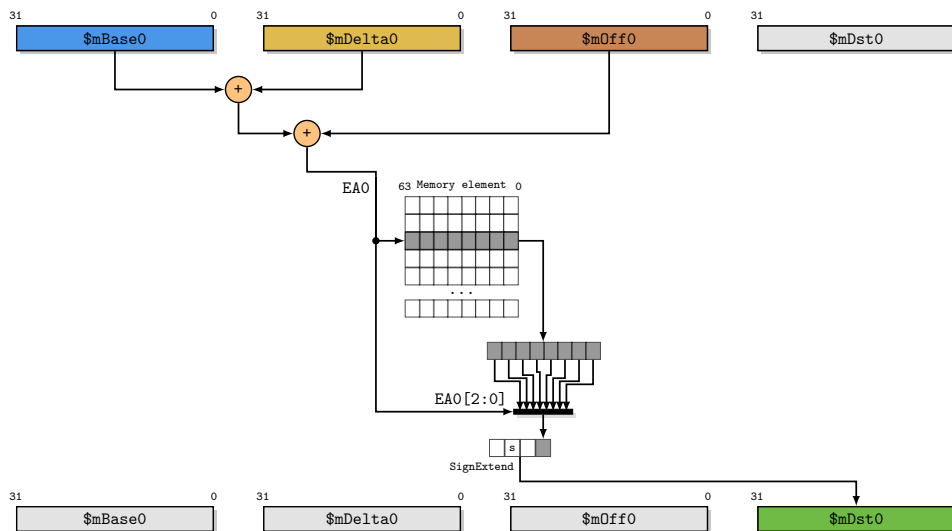


Fig. 3.38: lds8

3.7.5.3.4 ldz16

Load and zero-extend a single, naturally aligned 16-bit quantity from *Tile Memory*.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (*\$m* register)
- Unsigned address delta (*\$m* register)
- Unsigned scaled offset (*\$m* register or immediate)

Data format:

- Result is a 32-bit value formed by zero-extending the 16-bit data value.

Table 3.228: *ldz16* instruction definition

<i>ldz16</i>	both	main
Syntax	<code>ldz16 \$mDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>ldz16 \$mDst0, \$mBase0, \$mDelta0, zimm12</code>	
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <<((\$mOff0 << 1), (zimm12 << 1))>>; EA[0] = op0 + op1 + op2;</pre>
	Except In	<pre>if (!Mem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>
	Memory	<code>uint16_t data = loadHalf(EA[0]);</code>
	Commit	<code>\$mDst0 = Tile_ZeroExtend(data, 16);</code>

Function references: *Tile_ZeroExtend* , *TMem_IsValidAddress*

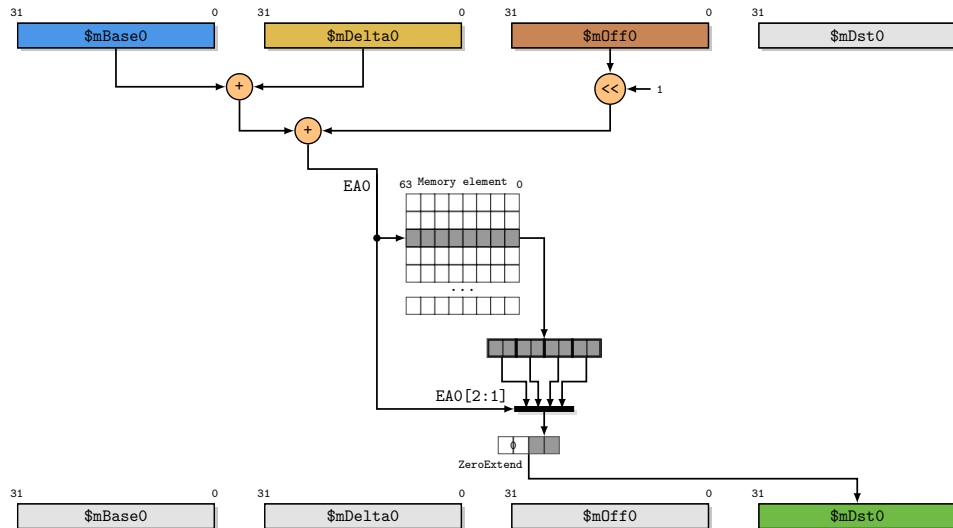


Fig. 3.39: ldz16

3.7.5.3.5 lds16

Load and sign-extend a single, naturally aligned 16-bit quantity from *Tile Memory*.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Data format:

- Result is a 32-bit value formed by sign-extending the 16-bit data value.

Table 3.229: *lds16* instruction definition

<i>lds16</i>	both	main
Syntax	<code>lds16 \$mDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>lds16 \$mDst0, \$mBase0, \$mDelta0, zimm12</code>	
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <<(\$mOff0 << 1), (zimm12 << 1)>>; EA[0] = op0 + op1 + op2;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } Memory <code>uint16_t data = loadHalf(EA[0]);</code></pre> <p>Commit <code>\$mDst0 = Tile_SignExtend(data, 16);</code></p>	

Function references: *Tile_SignExtend*, *TMem_IsValidAddress*

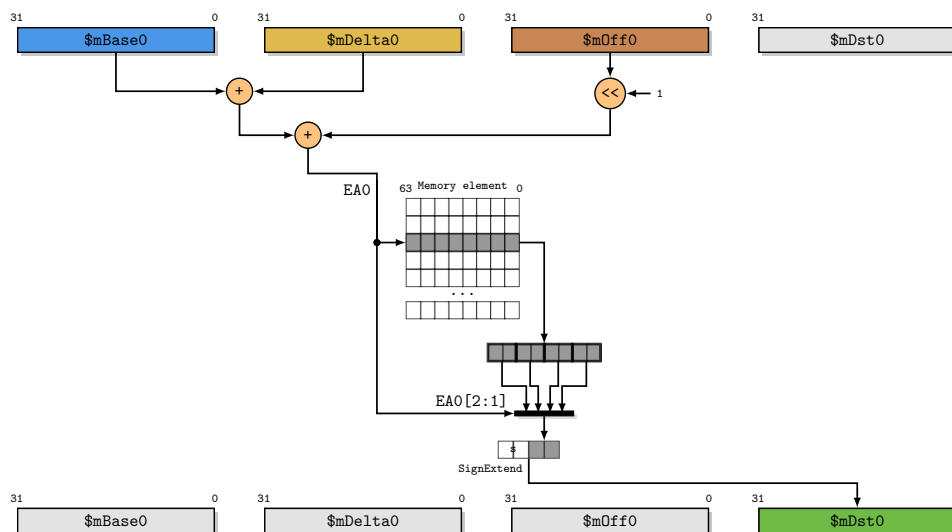


Fig. 3.40: *lds16*

3.7.5.3.6 *ldb16*

Load and broadcast a single, naturally aligned 16-bit quantity from *Tile Memory*.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (*\$m* register)
- Unsigned address delta (*\$m* register)
- Unsigned scaled offset (*\$m* register or immediate)

Data format:

- Result is a 32-bit value formed by broadcasting (duplicating) the 16-bit data value.

Table 3.230: *ldb16* instruction definition

<i>ldb16</i>		worker	main
Syntax	<code>ldb16 \$aDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>ldb16 \$aDst0, \$mBase0, \$mDelta0, zimm12</code>		
Semantics	Prepare	DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(((\$mOff0 << 1), (zimm12 << 1))>>); EA[0] = op0 + op1 + op2;	
	Except In	<pre> if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } </pre>	
	Memory	<code>uint16_t data = loadHalf(EA[0]);</code>	
	Commit	<code>\$aDst0 = {</code> <code> ((data & 0xffff) << 0) </code> <code> ((data & 0xffff) << 16)</code> <code>};</code>	

Function references: *TMem_IsValidAddress*

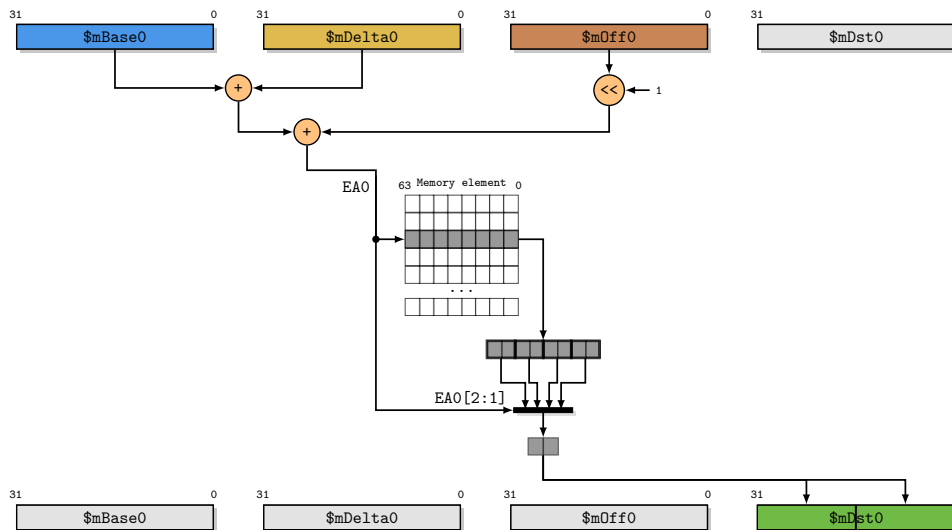


Fig. 3.41: *ldb16*

3.7.5.3.7 *ld32*

Load a single, naturally aligned 32-bit value from *Tile Memory*.

Destination register-file: *MRF* or *ARF*

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

- Unsigned scaled offset (\$m register or immediate)

Data format:

- Result is an unmodified 32-bit value.

Table 3.231: *ld32* instruction definition

<i>ld32</i>		both	main
Syntax	<code>ld32 \$mDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>ld32 \$mDst0, \$mBase0, \$mDelta0, zimm12</code>		
<i>ld32</i>		worker	main
Syntax	<code>ld32 \$aDst0, \$mBase0, \$mDelta0, \$mOff0</code> <code>ld32 \$aDst0, \$mBase0, \$mDelta0, zimm12</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((\$mOff0 << 2), (zimm12 << 2))>; EA[0] = op0 + op1 + op2;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } }</pre>	
	Memory	<pre>uint32_t data = loadWord(EA[0]);</pre>	
	Commit	<pre><(\$mDst0, \$aDst0)> = data;</pre>	

Function references: *TMem_IsValidAddress*

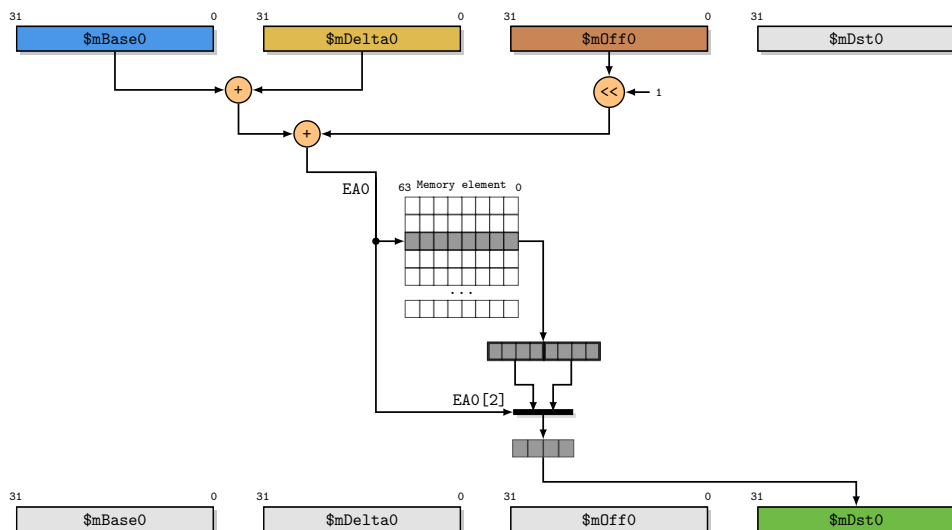


Fig. 3.42: *ld32*

3.7.5.3.8 *ld64*

Load a single, naturally aligned 64-bit quantity from *Tile Memory*.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Data format:

- Result is an unmodified 64-bit value stored in a naturally aligned register-pair.

Table 3.232: *ld64* instruction definition

<i>ld64</i>		worker	main
Syntax	<code>ld64 \$aDst0:Dst0+1, \$mBase0, \$mDelta0, \$mOff0</code> <code>ld64 \$aDst0:Dst0+1, \$mBase0, \$mDelta0, zimm12</code>		
Semantics	Prepare	DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(((\$mOff0 << 3), (zimm12 << 3))>); EA[0] = op0 + op1 + op2;	
	Except In	<pre> if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } </pre>	
	Memory	<code>uint64_t data = loadDoubleWord(EA[0]);</code>	
	Commit	<code>\$aDst0:Dst0+1 = {</code> <code> data & 0xffffffffFULL,</code> <code> (data >>32ULL) & 0xffffffffFULL</code> <code>};</code>	

Function references: *TMem_IsValidAddress*

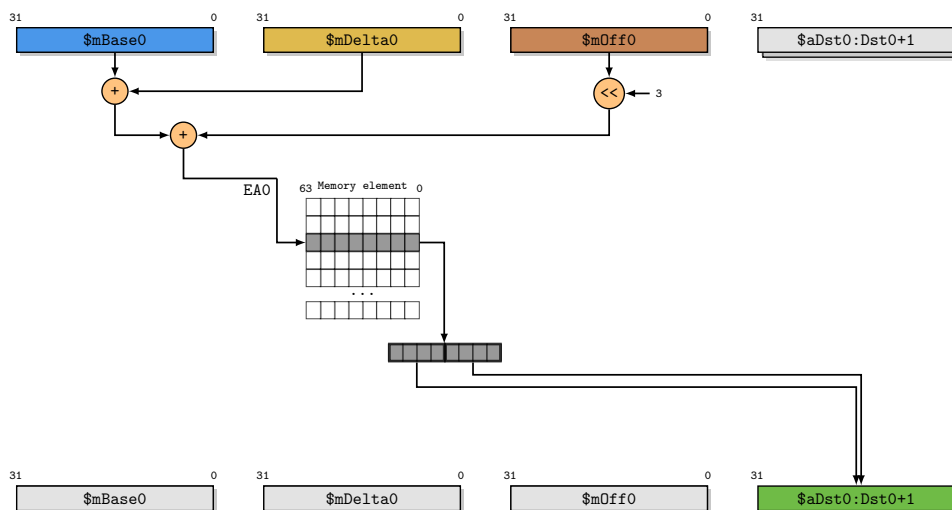


Fig. 3.43: *ld64*

3.7.5.3.9 ld128

Load a single, naturally aligned 128-bit quantity from an interleaved region of *Tile Memory*.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Data format:

- Result is an unmodified 128-bit value stored in a naturally aligned register-quad.

Table 3.233: *ld128* instruction definition

<i>ld128</i>		worker	main
Syntax	ld128 \$aDst0:Dst0+3, \$mBase0, \$mDelta0, \$mOff0 ld128 \$aDst0:Dst0+3, \$mBase0, \$mDelta0, zimm12		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((\$mOff0 << 4), (zimm12 << 4))>; array<uint64_t,2> data;</pre> <p>EA[0] = op0 + op1 + op2; EA[1] = (op0 + op1 + op2) 0x8;</p> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0xf) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (TMem_AddressInterleaveFactor(EA[0]) < 2) { // This access will cause a bank clash EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre> <p>Memory</p> <pre>data[0] = loadDoubleWord(EA[0]); data[1] = loadDoubleWord(EA[1]);</pre> <p>Commit</p> <pre>\$aDst0:Dst0+3 = { data[0] & 0xffffffffFULL, (data[0] >>32ULL) & 0xffffffffFULL, data[1] & 0xffffffffFULL, (data[1] >>32ULL) & 0xffffffffFULL };</pre>		

Function references: *TMem_IsValidAddress*, *TMem_AddressInterleaveFactor*

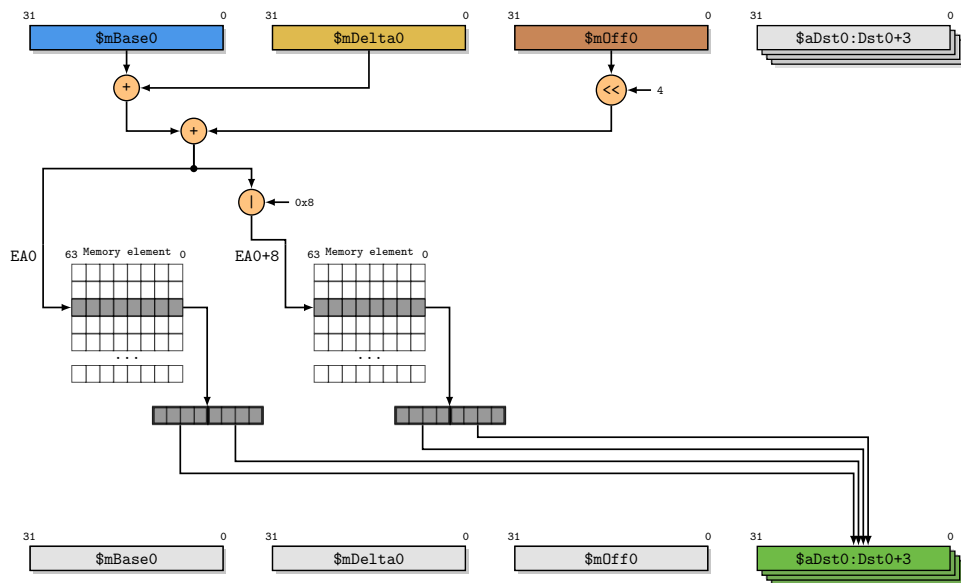


Fig. 3.44: ld128

3.7.5.3.10 ldz8step

Zero-extending 8-bit load with post-incrementing address.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Data format:

- Result is a 32-bit value formed by zero-extending the 8-bit data value.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register.

Table 3.234: *ldz8step* instruction definition

<i>ldz8step</i>		both	main
Syntax	<code>ldz8step \$mDst0, \$mBase0, \$mDelta0+ =, \$mStride0</code> <code>ldz8step \$mDst0, \$mBase0, \$mDelta0+ =, <i>simm8</i></code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(SignExtend32(<i>simm8</i>, 8), \$mStride0)>; EA[0] = op0 + op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>DataWord nextAddr = op1 + op2;</pre>	
	Memory	<pre>uint8_t data = loadByte(EA[0]);</pre>	
	Commit	<pre>\$mDelta0 = nextAddr; \$mDst0 = Tile_ZeroExtend(data, 8);</pre>	

Function references: [Tile_ZeroExtend](#) , [TMem_IsValidAddress](#)

3.7.5.3.11 *lds8step*

Sign-extending 8-bit load with post-incrementing address.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (*\$m* register)
- Unsigned address delta (*\$m* register)

Data format:

- Result is a 32-bit value formed by sign-extending the 8-bit data value.

Address auto-increment:

- The unsigned address delta *MRF* register operand is post-incremented by the signed immediate or stride register.

Table 3.235: *lds8step* instruction definition

<i>lds8step</i>		both	main
Syntax	<pre>lds8step \$mDst0, \$mBase0, \$mDelta0+ =, \$mStride0 lds8step \$mDst0, \$mBase0, \$mDelta0+ =, <i>simm8</i></pre>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <(SignExtend32(<i>simm8</i>, 8), \$mStride0)>; EA[0] = op0 + op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>DataWord nextAddr = op1 + op2;</pre>	
	Memory	<pre>uint8_t data = loadByte(EA[0]);</pre>	
	Commit	<pre>\$mDelta0 = nextAddr; \$mDst0 = Tile_SignExtend(data, 8);</pre>	

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

3.7.5.3.12 *ldz16step*

Zero-extending, naturally aligned 16-bit load with scaled post-incrementing address.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Data format:

- Result is a 32-bit value formed by zero-extending the 16-bit data value.

Address auto-increment:

- The unsigned address delta *MRF* register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.236: *ldz16step* instruction definition

<i>ldz16step</i>		both	main
Syntax	<code>ldz16step \$mDst0, \$mBase0, \$mDelta0+=, \$mStride0</code> <code>ldz16step \$mDst0, \$mBase0, \$mDelta0+=, <i>simm8</i></code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(<i>simm8</i>, 8) << 1), (\$mStride0 << 1))>; EA[0] = op0 + op1;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } Compute DataWord nextAddr = op1 + op2; Memory <code>uint16_t</code> data = loadHalf(EA[0]); Commit \$mDelta0 = nextAddr; \$mDst0 = Tile_ZeroExtend(data, 16);</pre>		

Function references: *Tile_ZeroExtend*, *TMem_IsValidAddress*

3.7.5.3.13 *lds16step*

Sign-extending, naturally aligned 16-bit load with scaled post-incrementing address.

Destination register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Data format:

- Result is a 32-bit value formed by sign-extending the 16-bit data value.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.237: *lds16step* instruction definition

<i>lds16step</i>		both	main
Syntax	<code>lds16step \$mDst0, \$mBase0, \$mDelta0+=, \$mStride0</code> <code>lds16step \$mDst0, \$mBase0, \$mDelta0+=, <i>simm8</i></code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(<i>simm8</i>, 8) << 1), (\$mStride0 << 1))>; EA[0] = op0 + op1;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } Compute DataWord nextAddr = op1 + op2; Memory <code>uint16_t</code> data = loadHalf(EA[0]); Commit \$mDelta0 = nextAddr; \$mDst0 = Tile_SignExtend(data, 16);</pre>		

Function references: *Tile_SignExtend*, *TMem_IsValidAddress*

3.7.5.3.14 *ldb16step*

Naturally aligned 16-bit load and broadcast with scaled post-incrementing address.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Data format:

- Result is a 32-bit value formed by broadcasting (duplicating) the 16-bit data value.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.238: *ldb16step* instruction definition

<i>ldb16step</i>		worker	main
Syntax	<code>ldb16step \$aDst0, \$mBase0, \$mDelta0+=, \$mStride0</code> <code>ldb16step \$aDst0, \$mBase0, \$mDelta0+=, <i>simm8</i></code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(simm8, 8) << 1), (\$mStride0 << 1))>; EA[0] = op0 + op1;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x1) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } Compute DataWord nextAddr = op1 + op2; Memory uint16_t data = loadHalf(EA[0]); Commit \$mDelta0 = nextAddr; \$aDst0 = { ((data & 0xffff) << 0) ((data & 0xffff) << 16) };</pre>		

Function references: [TMem_IsValidAddress](#)

3.7.5.3.15 *ld32step*

Naturally aligned single 32-bit load with scaled post-incrementing address.

Destination register-file: [MRF](#) or [ARF](#)

Effective address formed from:

- Base address ($\$m$ register)
- Unsigned address delta ($\$m$ register)

Data format:

- Result is an unmodified 32-bit value.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.239: *ld32step* instruction definition

<i>ld32step</i>		both	main
Syntax	ld32step \$mDst0, \$mBase0, \$mDelta0+ =, \$mStride0 ld32step \$mDst0, \$mBase0, \$mDelta0+ =, <i>simm8</i>		
<i>ld32step</i>		worker	main
Syntax	ld32step \$aDst0, \$mBase0, \$mDelta0+ =, \$mStride0 ld32step \$aDst0, \$mBase0, \$mDelta0+ =, <i>simm8</i>		
Semantics	Prepare	DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <<((SignExtend32(<i>simm8</i> , 8) << 2), (\$mStride0 << 2))>>; EA[0] = op0 + op1;	
	Except In	<pre> if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } </pre>	
	Compute	DataWord nextAddr = op1 + op2;	
	Memory	uint32_t data = loadWord(EA[0]);	
	Commit	\$mDelta0 = nextAddr; <(\$mDst0, \$aDst0)> = data;	

Function references: [TMem_IsValidAddress](#)

ld32step occurs in the following code examples:

- [ldb16b16 example](#)

3.7.5.3.16 ld64step

Naturally aligned 64-bit load with scaled post-incrementing address.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Data format:

- Result is an unmodified 64-bit value stored in a naturally aligned register-pair.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.240: *ld64step* instruction definition

<i>ld64step</i>		worker	main
Syntax	<code>ld64step \$aDst0:Dst0+1, \$mBase0, \$mDelta0+=, \$mStride0</code> <code>ld64step \$aDst0:Dst0+1, \$mBase0, \$mDelta0+=, <i>simm8</i></code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(<i>simm8</i>, 8) << 3), (\$mStride0 << 3))>; EA[0] = op0 + op1;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre> <p>Compute</p> <pre>DataWord nextAddr = op1 + op2;</pre> <p>Memory</p> <pre>uint64_t data = loadDoubleWord(EA[0]);</pre> <p>Commit</p> <pre>\$mDelta0 = nextAddr; \$aDst0:Dst0+1 = { data & 0xffffffffFULL, (data >> 32ULL) & 0xffffffffFULL };</pre>		

Function references: [TMem_IsValidAddress](#)

ld64step occurs in the following code examples:

- [f16v4stacc example](#)

3.7.5.3.17 ld128step

Naturally aligned 128-bit load from interleaved memory region with scaled post-incrementing address.

Destination register-file: *ARF* only

Effective address formed from:

- Base address (*\$m* register)
- Unsigned address delta (*\$m* register)

Data format:

- Result is an unmodified 128-bit value stored in a naturally aligned register-quad.

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.241: *ld128step* instruction definition

<i>ld128step</i>		worker	main
Syntax	<code>ld128step \$aDst0:Dst0+3, \$mBase0, \$mDelta0+=, \$mStride0</code> <code>ld128step \$aDst0:Dst0+3, \$mBase0, \$mDelta0+=, <i>simm8</i></code>		
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(<i>simm8</i>, 8) << 4), (\$mStride0 << 4))>; array<uint64_t,2> data;</pre> <p><code>EA[0] = op0 + op1;</code> <code>EA[1] = (op0 + op1) 0x8;</code></p> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0xf) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (TMem_AddressInterleaveFactor(EA[0]) < 2) { // This access will cause a bank clash EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } }</pre> <p>Compute</p> <pre>DataWord nextAddr = op1 + op2;</pre> <p>Memory</p> <pre>data[0] = loadDoubleWord(EA[0]); data[1] = loadDoubleWord(EA[1]);</pre> <p>Commit</p> <pre>\$mDelta0 = nextAddr; \$aDst0:Dst0+3 = { data[0] & 0xffffffffFULL, (data[0] >> 32ULL) & 0xffffffffFULL, data[1] & 0xffffffffFULL, (data[1] >> 32ULL) & 0xffffffffFULL };</pre>		

Function references: *TMem_IsValidAddress* , *TMem_AddressInterleaveFactor*

3.7.5.3.18 *ld64putcs*

Load a naturally-aligned 64-bit quantity and write the value to the *common compute configuration space*. The load address is provided by *\$CCCSLOAD*, which is automatically post-incremented by 8.

Table 3.242: *ld64putcs* instruction definition

<i>ld64putcs</i>		supervisor	main
Syntax	<code>ld64putcs <i>zimm8</i></code>		
Semantics	Prepare	<pre>DataWord op0 = zimm8; EA[0] = \$CCCSLOAD;</pre>	
	Except In	<pre>// Check that the common configuration space address is valid if (!TReg_IsValidCCCS(op0)) { EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } }</pre>	
	Memory	<pre>uint64_t data = loadDoubleWord(EA[0]);</pre>	
	Commit	<pre>setCCCSState(op0, data); \$CCCSLOAD = \$CCCSLOAD + 8;</pre>	

Architectural state references: *\$CCCSLOAD*

Function references: *TMem_IsValidAddress* , *TReg_IsValidCCCS*

3.7.5.3.19 *ld128putcs*

Load a naturally aligned 128-bit quantity, from a *memory region* with an *interleave factor* of at least 2 and write the value to the *common compute configuration space*. The load address is provided by *\$CCCSLOAD*, which is automatically post-incremented by 16.

Table 3.243: *ld128putcs* instruction definition

<i>ld128putcs</i>		supervisor	main
Syntax	ld128putcs <i>zimm8</i>		
Semantics	Prepare	DataWord op0 = zimm8; array<uint64_t,2> data; EA[0] = \$CCCSLOAD; EA[1] = (\$CCCSLOAD) 0x8;	
	Except In	<pre> // Check that the common configuration space address is valid if (!TReg_IsValidCCCS(op0)) { EXCEPT(TEXCPT_INVALID_OP); } else if (!TReg_IsValidCCCS(op0 + 1)) { EXCEPT(TEXCPT_INVALID_OP); } else if (op0 & 1) { // CCCS address must be even EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0xf) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (TMem_AddressInterleaveFactor(EA[0]) < 2) { // This access will cause a bank clash EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } </pre>	
	Memory	data[0] = loadDoubleWord(EA[0]); data[1] = loadDoubleWord(EA[1]);	
	Commit	setCCCSState(op0, data[0]); setCCCSState(op0 1, data[1]); \$CCCSLOAD = \$CCCSLOAD + 16;	

Architectural state references: *\$CCCSLOAD*

Function references: *TMem_IsValidAddress*, *TMem_AddressInterleaveFactor*, *TReg_IsValidCCCS*

3.7.5.4 Single-store

3.7.5.4.1 st32

Store a single 32-bit register value to *Tile Memory*.

Source register-file: *MRF* or *ARF*

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Table 3.244: *st32* instruction definition

<i>st32</i>	both	main
Syntax	<i>st32 \$mSrc0, \$mBase0, \$mDelta0, zimm12</i>	
<i>st32</i>	worker	main
Syntax	<i>st32 \$aSrc0, \$mBase0, \$mDelta0, \$mOffset0</i> <i>st32 \$aSrc0, \$mBase0, \$mDelta0, zimm12</i>	
Semantics	<p>Prepare</p> <pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((\$mOffset0 << 2), (zimm12 << 2))>; DataWord op3 = <(\$aSrc0, \$mSrc0)>; EA[0] = op0 + op1 + op2;</pre> <p>Except In</p> <pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre> <p>Memory</p> <pre>uint32_t data = op3; storeWord(EA[0], data);</pre>	

Function references: *TMem_IsValidAddress*

3.7.5.4.2 *stm32*

Store a single 32-bit MRF register value to *Tile Memory*.

Source register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned scaled offset (\$m register)

Table 3.245: *stm32* instruction definition

<i>stm32</i>		both	main
Syntax	<code>stm32 \$mSrc0, \$mBase0, \$mOffset0</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = (\$mOffset0 << 2); DataWord op2 = \$mSrc0; EA[0] = op0 + op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Memory	<pre>uint32_t data = op2; storeWord(EA[0], data);</pre>	

Function references: [TMem_IsValidAddress](#)

3.7.5.4.3 *st64*

Store a single 64-bit value, from a naturally aligned register pair to *Tile Memory*.

Source register-file: *ARF* only

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)
- Unsigned scaled offset (\$m register or immediate)

Table 3.246: *st64* instruction definition

<i>st64</i>	worker	main
Syntax	<pre>st64 \$aSrc0:Src0+1, \$mBase0, \$mDelta0, \$mOffset0 st64 \$aSrc0:Src0+1, \$mBase0, \$mDelta0, zimm12</pre>	
Semantics	<pre>Prepare DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((\$mOffset0 << 3), (zimm12 << 3))>; array<DataWord,2> op3 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; EA[0] = op0 + op1 + op2; Except In if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } Memory uint64_t data = (((uint64_t)op3[1]) << 32ULL) ((uint64_t)op3[0]); storeDoubleWord(EA[0], data);</pre>	

Function references: [TMem_IsValidAddress](#)

3.7.5.4.4 *st64pace*

Naturally aligned 64-bit store with scaled post-incrementing address.

Source register-file: *ARF* only (a naturally aligned register-pair)

Effective address:

- Absolute address 2 from a triple packed address register pair

Address auto-increment:

- Specified by a 2-bit immediate:
 - 0b00: 1 (atom)
 - 0b01: The (scaled) signed 10-bit value $\$mStride0[9:0]$
 - 0b10: The (scaled) signed 10-bit value $\$mStride0[19:10]$
 - 0b11: The (scaled) signed 10-bit value $\$mStride0[29:20]$

See *Striding support* for details.

Table 3.247: *st64pace* instruction definition

<i>st64pace</i>		worker	main
Syntax	<code>st64pace \$aSrc0:Src0+1, \$mAddr0:Addr0+1 +=, \$mStride0, Strimm2</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mStride0; array<DataWord,2> op1 = { \$mAddr0:Addr0+1[0], \$mAddr0:Addr0+1[1] }; DataWord op2 = Strimm2; array<DataWord,2> op3 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; array<DataWord,3> addr;</pre> <pre>EA[2] = Tile_ExtractPackedAddress(op1, 2);</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[2])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[2] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[2])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>// Auto-increment address int32_t stride = Tile_ExtractPackedStride(op0, op2 & 0x3); addr[0] = Tile_ExtractPackedAddress(op1, 0); // Unchanged addr[1] = Tile_ExtractPackedAddress(op1, 1); // Unchanged addr[2] = (EA[2] + (stride * 8)) & TMem_FULL_ADDRESS_MASK;</pre>	
	Memory	<pre>uint64_t data = (((uint64_t)op3[1]) << 32ULL ((uint64_t)op3[0])); storeDoubleWord(EA[2], data);</pre>	
	Commit	<pre>\$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addr), Tile_TripleAddressPack_Upper(addr) };</pre>	

Function references: [Tile_ExtractPackedAddress](#) , [Tile_ExtractPackedStride](#) , [Tile_TripleAddressPack_Lower](#) , [Tile_TripleAddressPack_Upper](#) , [TMem_IsValidAddress](#)

st64pace occurs in the following code examples:

- [f16v4sisoslic example part 1](#)
- [f32sisoslic example](#)
- [f16v4sisoslic example part 2](#)

3.7.5.4.5 *st32step*

Naturally aligned 32-bit store with scaled post-incrementing address.

Source register-file: *MRF* or *ARF*

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.248: *st32step* instruction definition

<i>st32step</i>		both	main
Syntax	<i>st32step</i> \$mSrc0, \$mBase0, \$mDelta0+ =, <i>simm8</i>		
<i>st32step</i>		worker	main
Syntax	<i>st32step</i> \$aSrc0, \$mBase0, \$mDelta0+ =, \$mStride0 <i>st32step</i> \$aSrc0, \$mBase0, \$mDelta0+ =, <i>simm8</i>		
Semantics	Prepare	<pre>DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(simm8, 8) << 2), (\$mStride0 << 2))>; DataWord op3 = <(\$mSrc0, \$aSrc0)>; EA[0] = op0 + op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	DataWord nextAddr = op1 + op2;	
	Memory	<pre>uint32_t data = op3; storeWord(EA[0], data);</pre>	
	Commit	\$mDelta0 = nextAddr;	

Function references: [TMem_IsValidAddress](#)

3.7.5.4.6 *stm32step*

Naturally aligned 32-bit store from MRF with scaled post-incrementing address.

Source register-file: *MRF* only

Effective address formed from:

- Base address (\$m register)

Address auto-increment:

- The base address register operand is post-incremented by the signed stride register value (after the value has been scaled to atom size).

Table 3.249: *stm32step* instruction definition

<i>stm32step</i>		both	main
Syntax	<code>stm32step \$mSrc0, \$mBase0+ =, \$mStride0</code>		
Semantics	Prepare	<pre>DataWord op2 = \$mSrc0; DataWord op1 = \$mBase0; SignedDataWord op0 = (\$mStride0 << 2); EA[0] = op1;</pre>	
	Except In	<pre>if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x3) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); }</pre>	
	Compute	<pre>DataWord nextAddr = op1 + op0;</pre>	
	Memory	<pre>uint32_t data = op2; storeWord(EA[0], data);</pre>	
	Commit	<pre>\$mBase0 = nextAddr;</pre>	

Function references: [*TMem_IsValidAddress*](#)

3.7.5.4.7 *st64step*

Naturally aligned 64-bit store with scaled post-incrementing address.

Source register-file: *ARF* only (a naturally aligned register-pair)

Effective address formed from:

- Base address (\$m register)
- Unsigned address delta (\$m register)

Address auto-increment:

- The unsigned address delta MRF register operand is post-incremented by the signed immediate or stride register (after the value has been scaled to atom size).

Table 3.250: *st64step* instruction definition

<i>st64step</i>		worker	main
Syntax	st64step \$aSrc0:Src0+1, \$mBase0, \$mDelta0+=, \$mStride0 st64step \$aSrc0:Src0+1, \$mBase0, \$mDelta0+=, <i>simm8</i>		
Semantics	Prepare	<pre> DataWord op0 = \$mBase0; DataWord op1 = \$mDelta0; DataWord op2 = <((SignExtend32(simm8, 8) << 3), (\$mStride0 << 3))>; array<DataWord,2> op3 = { \$aSrc0:Src0+1[0], \$aSrc0:Src0+1[1] }; EA[0] = op0 + op1; </pre>	
	Except In	<pre> if (!TMem_IsValidAddress(EA[0])) { EXCEPT(TEXCPT_INVALID_ADDR); } else if (EA[0] & 0x7) { // Misaligned address EXCEPT(TEXCPT_INVALID_ADDR); } else if (hadMemoryConflict(EA[0])) { EXCEPT(TEXCPT_CONFLICT); } </pre>	
	Compute	<pre> DataWord nextAddr = op1 + op2; </pre>	
	Memory	<pre> uint64_t data = (((uint64_t)op3[1]) << 32ULL) ((uint64_t)op3[0]); storeDoubleWord(EA[0], data); </pre>	
	Commit	<pre> \$mDelta0 = nextAddr; </pre>	

Function references: [TMem_IsValidAddress](#)

3.7.6 System

Table 3.251: system instructions summary

Mnemonic	Implemented?	Super?	Worker?	main?	aux?	Brief
<i>get</i>	✓	✓	✓	✓	✗	Lower control register read
<i>put</i>	✓	✓	✓	✓	✗	Write to a lower control register
<i>run</i>	✓	✓	✗	✓	✗	Launch a worker thread
<i>runall</i>	✓	✓	✗	✓	✗	Launch a batch of worker threads
<i>trap</i>	✓	✓	✓	✓	✗	Patched BREAKPOINT
<i>uget</i>	✓	✗	✓	✗	✓	Upper control register read
<i>uput</i>	✓	✗	✓	✗	✓	Write to an upper control register

3.7.6.1 get

Read the value of a control/status register into a general purpose register. See *Control and Status registers*.

Table 3.252: *get* instruction definition

<i>get</i>	both	main
Syntax	get <i>\$mDst0</i> , <i>zimm8</i>	
Semantics	Prepare DataWord op1 = zimm8; LOADED_STATE = readState(op1); Except In if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute get in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } if (!TReg_IsValidCSR(op1, isSupervisor())) { EXCEPT(TEXCPT_INVALID_OP); } Commit \$mDst0 = LOADED_STATE;	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TReg_IsValidCSR*

3.7.6.2 put

Write to a control register. See *Control and Status registers*.

Table 3.253: `put` instruction definition

<code>put</code>		both	main
Syntax	<code>put <i>zimm8</i>, \$mSrc0</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mSrc0; DataWord op1 = <i>zimm8</i>;</pre>	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT) && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } if (isSupervisor()) { if ((op1 >= CSR_S_WORKER0_BASE__INDEX) && (op1 < (CSR_S_WORKER0_BASE__INDEX + CTXT_WORKERS))) { // Cannot modify WORKERn_BASE when Worker n is active unsigned wid = op1 - CSR_S_WORKER0_BASE__INDEX; if (((CTXT_STS >> (2 * (wid + 1))) & 0x3) != TCTXT_STATUS_INACTIVE) { EXCEPT(TEXCPT_INVALID_INSTR); } } } TileException_t writeException = TReg_WriteException(op1, isSupervisor()); switch (writeException) { case TEXCPT_INVALID_OP: EXCEPT(TEXCPT_INVALID_OP); break; } if ((op1 == CSR_S_INCOMING_MUX__INDEX op1 == CSR_S_INCOMING_MUXPAIR__INDEX) && pairedTileUpdatedMuxPair()) { EXCEPT(TEXCPT_EXCONF); } if (TEXCH_TLINK_PUT_IMUX_EXCEPTION && isReceivingTlinkPacket() && ((op1 == CSR_S_INCOMING_MUX__INDEX) (op1 == CSR_S_INCOMING_MUXPAIR__INDEX))) { EXCEPT(TEXCPT_EXCONF); } </pre>	
	Commit	<code>writeState(op1, op0);</code>	

Architectural state references: `$REPEAT_COUNT`

Function references: `TReg_WriteException`

3.7.6.3 run

Launch a worker *thread*.

Allocate execution time and *context* state to the *thread* whose entry point is given by the register operand `$mEntry0`, using the vertex address calculated by summing:

1. the register operand `$mVBase0`
2. the 16-bit immediate offset `zimm16 × 4`
3. the constant `TMEM_REGION0_BASE_ADDR`

(i.e. the address formed by adding the register value `$mVBase0` to the scaled immediate offset `zimm16` is relative to `TMEM_REGION0_BASE_ADDR`)

Exception events will be raised for any of the following conditions:

- `$mEntry0` is not 4-byte aligned
- `$mVBase0` is not 4-byte aligned
- `$mEntry0` is not a valid, executable address

In order to ensure that all worker contexts are inactive a:

`sync TEXCH_SYNCZONE_LOCAL`

instruction should be executed following the `run`.

Table 3.254: `run` instruction definition

<code>run</code>		supervisor	main
Syntax	<code>run \$mEntry0, \$mVBase0, zimm16</code>		
Semantics	Prepare	<pre>DataWord op0 = \$mEntry0; DataWord op1 = \$mVBase0; DataWord op2 = (zimm16 << 2);</pre>	
	Except In	<pre>if (op0 & 0x3) { // Misaligned Worker \$PC EXCEPT(TEXCPT_INVALID_OP); } else if (op1 & 0x3) { // Misaligned Worker \$VERTEX_BASE EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_IsValidAddress(op0)) { // Bad address for Worker \$PC EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_AddressIsExecutable(op0)) { // Bad address for Worker \$PC EXCEPT(TEXCPT_INVALID_OP); }</pre>	
	Commit	<pre>DataWord vertexBase = op1 + op2 + TMem_REGION0_BASE_ADDR; Context &worker = getNextFreeWorker(); worker.\$PC = op0; worker.\$VERTEX_BASE = vertexBase; worker.\$FP_STS = CSR_W_FP_STS__RESET; worker.\$FP_CTL = \$FP_ICTL; worker.setRunMode(TRUNM_EXECUTING);</pre>	

Architectural state references: `$PC`, `$VERTEX_BASE`, `$FP_STS`, `$FP_CTL`, `$FP_ICTL`

Function references: `TMem_IsValidAddress`, `TMem_AddressIsExecutable`

3.7.6.4 `runall`

Allocate execution time and context state to a batch of worker *threads*:

- The total number of *threads* launched is equal to the number of hardware *worker* contexts (`CTXT_WORKERS`)
- All *threads* use the same entry point (`$mEntry0`)
- The value of `$VERTEX_BASE` assigned to the first allocated *worker* is provided by the register `$mVBase0`
- The value of `$VERTEX_BASE` assigned to every other *worker* is:

$$- \$mVBase0 + (n \times zimm16 \times 4) \quad (n \in [1, CTXT_WORKERS - 1])$$

Exception events will be raised for any of the following conditions:

- `$mEntry0` is not 4-byte aligned
- `$mVBase0` is not 4-byte aligned
- `$mEntry0` is not a valid, executable address
- There are any active Worker contexts

If there are active Worker contexts, **\$\$\$SR.RAERR** will also be set to 0b1 and all active Workers will raise a **TEXCPT_INVALID_INSTR** exception during the retirement of their next instruction.

In order to ensure that all worker contexts are inactive a:

sync TEXCH_SYNCZONE_LOCAL

instruction should be executed following the *runall*.

Table 3.255: *runall* instruction definition

<i>runall</i>	supervisor	main
Syntax	<code>runall \$mEntry0, \$mVBase0, zimm16</code>	
Semantics	Prepare	<pre>DataWord op0 = \$mEntry0; DataWord op1 = \$mVBase0; DataWord op2 = zimm16;</pre>
	Except In	<pre>// Raise an exception if there are any active worker contexts bool raiseExcpt = false; for (int w = 0; w < CTXT_WORKERS; w++) { if (((CTXT_STS >> (2 * (w + 1))) & 0x3) != TCTXT_STATUS_INACTIVE) { raiseExcpt = true; } } if (raiseExcpt) { EXCEPT(TEXCPT_INVALID_INSTR); if (isSupervisor()) { // Remove reference to SSR from codelet ISA // Force all active Worker contexts to also except (at retirement) \$\$\$SR.RAERR = 1; } } if (op0 & 0x3) { // Misaligned Worker \$PC EXCEPT(TEXCPT_INVALID_OP); } else if (op1 & 0x3) { // Misaligned Worker \$VERTEX_BASE EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_IsValidAddress(op0)) { // Bad address for Worker \$PC EXCEPT(TEXCPT_INVALID_OP); } else if (!TMem_AddressIsExecutable(op0)) { // Bad address for Worker \$PC EXCEPT(TEXCPT_INVALID_OP); } Commit for (int w = 0; w < CTXT_WORKERS; w++) { uint32_t stride = op2 << 2; uint32_t vertexBase = op1 + (stride * w); Context &worker = getNextFreeWorker(); worker.\$PC = op0; worker.\$VERTEX_BASE = vertexBase; worker.\$FP_STS = CSR_W_FP_STS__RESET; worker.\$FP_CTL = \$FP_ICTL; worker.setRunMode(TRUNM_EXECUTING); }</pre>

Architectural state references: *\$PC*, *\$VERTEX_BASE*, *\$FP_STS*, *\$FP_CTL*, *\$CTXT_STS*, *\$FP_ICTL*

Function references: *TMem_IsValidAddress*, *TMem_AddressIsExecutable*

3.7.6.5 trap

Unconditionally raise a *patched breakpoint exception event*.

Table 3.256: *trap* instruction definition

<i>trap</i>		both	main
Syntax	trap <i>zimm4</i>		
Semantics	Prepare	DataWord op0 = zimm4;	
	Except In	<pre>if (!isSupervisor() && (0 != \$REPEAT_COUNT)) { // Cannot execute trap in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } else { if ((op0 & 1) == 0) { EXCEPT(TEXCPT_PBRK0); } else { EXCEPT(TEXCPT_PBRK1); } }</pre>	

Architectural state references: *\$REPEAT_COUNT*

3.7.6.6 uget

Read the value of a control/status register into a general purpose register. See *Control and Status registers*.

Table 3.257: *uget* instruction definition

<i>uget</i>		worker	aux
Syntax	uget \$aDst0, <i>zimm8</i>		
Semantics	Prepare	DataWord op1 = zimm8; LOADED_STATE = readState(TREG_UPPER_CSR_SPACE_BASE + op1);	
	Except In	<pre>if (0 != \$REPEAT_COUNT && !isExecutingTDI()) { // Cannot execute get in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } if (!TReg_IsValidCSR(TREG_UPPER_CSR_SPACE_BASE + op1, false)) { EXCEPT(TEXCPT_INVALID_OP); }</pre>	
	Commit	\$aDst0 = LOADED_STATE;	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TReg_IsValidCSR*

3.7.6.7 uput

Write to a control register in the upper CSR address space. See *Worker CSRs*

Table 3.258: *uput* instruction definition

<i>uput</i>		worker	aux
Syntax	uput <i>zimm8</i> , \$aSrc0		
Semantics	Prepare	DataWord op0 = \$aSrc0; DataWord op1 = <i>zimm8</i> ;	
	Except In	<pre> if (0 != \$REPEAT_COUNT && !isExecutingTDI()) { // Cannot execute this instruction in the body of rpt EXCEPT(TEXCPT_INVALID_INSTR); } if (!TReg_IsValidCSR(TREG_UPPER_CSR_SPACE_BASE + op1, false)) { EXCEPT(TEXCPT_INVALID_OP); } </pre>	
	Commit	writeState(TREG_UPPER_CSR_SPACE_BASE + op1, op0);	

Architectural state references: *\$REPEAT_COUNT*

Function references: *TReg_IsValidCSR*

3.8 Instructions by attribute

3.8.1 Full instruction list per pipeline

The following sections list all instructions applicable to the given execution pipeline.

3.8.1.1 main

- *abs*
- *add*
- *and*
- *andc*
- *atom*
- *bitrev8*
- *br*
- *bri*
- *brneg*
- *brnz*
- *brnzdec*
- *brpos*
- *brz*
- *call*
- *clz*
- *cmpeq*
- *cmpne*
- *cmpslt*
- *cmpult*
- *cms*
- *exitneg*
- *exitnz*
- *exitpos*
- *exitz*
- *get*
- *ld128*
- *ld128putcs*
- *ld128step*
- *ld2x64pace*
- *ld2xst64pace*
- *ld32*
- *ld32step*
- *ld64*
- *ld64a32*
- *ld64a32pace*
- *ld64b16pace*
- *ld64putcs*
- *ld64step*
- *ldb16*
- *ldb16b16*
- *ldb16step*
- *ldd16a32*
- *ldd16a64*
- *ldd16b16*
- *ldd16v2a32*
- *lds16*
- *lds16step*
- *lds8*
- *lds8step*
- *ldst64pace*
- *ldz16*
- *ldz16step*
- *ldz8*
- *ldz8step*
- *max*
- *min*
- *movz*
- *mul*
- *or*
- *popc*
- *put*
- *roll16*
- *roll8l*
- *roll8r*
- *rpt*
- *run*
- *runall*
- *setzi*
- *shl*
- *shr*
- *shrs*
- *shuf8x8hi*
- *shuf8x8lo*
- *sort4x16hi*
- *sort4x16lo*
- *sort8*
- *sort8x8hi*
- *sort8x8lo*
- *st32*
- *st32step*
- *st64*
- *st64pace*
- *st64step*
- *stm32*
- *stm32step*
- *sub*
- *swap8*
- *tapack*
- *trap*
- *xnor*
- *xor*

3.8.1.2 aux

- *and*
- *and64*
- *andc*
- *andc64*
- *f16tof32*
- *f16v2absadd*
- *f16v2absmax*
- *f16v2add*
- *f16v2clamp*
- *f16v2class*
- *f16v2cmac*
- *f16v2cmpeq*
- *f16v2cmpge*
- *f16v2cmpgt*
- *f16v2cmple*
- *f16v2cmplt*
- *f16v2cmpne*
- *f16v2exp*
- *f16v2exp2*
- *f16v2gina*
- *f16v2grand*
- *f16v2ln*
- *f16v2log2*
- *f16v2max*
- *f16v2maxc*
- *f16v2min*
- *f16v2mul*
- *f16v2sigm*
- *f16v2sub*
- *f16v2sufromui*
- *f16v2sum*
- *f16v2tanh*
- *f16v2tof32*
- *f16v4absacc*
- *f16v4absadd*
- *f16v4absmax*
- *f16v4acc*
- *f16v4add*
- *f16v4clamp*
- *f16v4class*

- *f16v4cmac*
- *f16v4cmpeq*
- *f16v4cmpge*
- *f16v4cmpgt*
- *f16v4cmple*
- *f16v4cmplt*
- *f16v4cmpne*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4istacc*
- *f16v4max*
- *f16v4maxc*
- *f16v4min*
- *f16v4mix*
- *f16v4mul*
- *f16v4rmask*
- *f16v4sihoamp*
- *f16v4sihoslic*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4stacc*
- *f16v4sub*
- *f16v4sufromui*
- *f16v4sum*
- *f16v8absacc*
- *f16v8acc*
- *f16v8sqacc*
- *f32absadd*
- *f32absmax*
- *f32add*
- *f32clamp*
- *f32class*
- *f32cmpeq*
- *f32cmpge*
- *f32cmpgt*
- *f32cmple*
- *f32cmplt*
- *f32cmpne*
- *f32div*
- *f32exp*
- *f32exp2*
- *f32fromi32*
- *f32fromui32*
- *f32int*
- *f32ln*
- *f32log2*
- *f32mac*
- *f32max*
- *f32min*
- *f32mul*
- *f32oorx*
- *f32oox*
- *f32sigm*
- *f32sisoamp*
- *f32sisoslic*
- *f32sisov2amp*
- *f32sisov2slic*
- *f32sqrt*
- *f32sub*
- *f32sufromui*
- *f32tanh*
- *f32tof16*
- *f32toi32*
- *f32toui32*
- *f32v2absadd*
- *f32v2absmax*
- *f32v2add*
- *f32v2aop*
- *f32v2axpy*
- *f32v2clamp*
- *f32v2class*
- *f32v2cmpeq*
- *f32v2cmpge*
- *f32v2cmpgt*
- *f32v2cmple*
- *f32v2cmplt*
- *f32v2cmpne*
- *f32v2gina*
- *f32v2grand*
- *f32v2mac*
- *f32v2max*
- *f32v2min*
- *f32v2mul*
- *f32v2rmask*
- *f32v2sub*
- *f32v2sufromui*
- *f32v2tof16*
- *f32v4absacc*
- *f32v4acc*
- *f32v4sqacc*
- *f32v4tof16*
- *not*
- *not64*
- *or*
- *or64*
- *roll16*
- *roll32*
- *setzi*
- *sort4x16hi*
- *sort4x16lo*
- *sort4x32hi*
- *sort4x32lo*
- *uget*
- *uput*
- *urand32*
- *urand64*

3.8.1.3 both

- *and*
- *andc*
- *or*
- *roll16*
- *setzi*
- *sort4x16hi*

3.8.2 Instructions by FP exception

3.8.2.1 TFPEXCPT_INV

The following instructions can directly raise *TFPEXCPT_INV* floating point exceptions:

- *f16v2absadd*
- *f16v2add*
- *f16v2exp*
- *f16v2exp2*
- *f16v2gina*
- *f16v2ln*
- *f16v2log2*
- *f16v2mul*
- *f16v2sigm*
- *f16v2sub*
- *f16v2sum*
- *f16v2tanh*
- *f16v4absadd*
- *f16v4add*
- *f16v4clamp*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*

- *f16v4mix*
- *f16v4mul*
- *f16v4sihoamp*
- *f16v4sihoslic*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4sub*
- *f16v4sum*
- *f32absadd*
- *f32absmax*
- *f32add*
- *f32clamp*
- *f32div*
- *f32exp*
- *f32exp2*
- *f32int*
- *f32ln*
- *f32log2*
- *f32mac*
- *f32max*
- *f32min*
- *f32mul*
- *f32oorx*
- *f32oox*
- *f32sigm*
- *f32sisoamp*
- *f32sisoslic*
- *f32sisov2amp*
- *f32sisov2slic*
- *f32sqrt*
- *f32sub*
- *f32tanh*
- *f32tof16*
- *f32v2absadd*
- *f32v2add*
- *f32v2aop*
- *f32v2axpy*
- *f32v2gina*
- *f32v2mac*
- *f32v2mul*
- *f32v2sub*
- *f32v2tof16*
- *f32v4tof16*

3.8.2.2 TFPEXCPT_DIV0

The following instructions can directly raise *TFPEXCPT_DIV0* floating point exceptions:

- *f16v2ln*
- *f16v2log2*
- *f32div*
- *f32ln*
- *f32log2*
- *f32oorx*
- *f32oox*

3.8.2.3 TFPEXCPT_OFLO

The following instructions can directly raise *TFPEXCPT_OFLO* floating point exceptions:

- *f16v2absadd*
- *f16v2add*
- *f16v2exp*
- *f16v2exp2*
- *f16v2gina*
- *f16v2mul*
- *f16v2sub*
- *f16v4absadd*
- *f16v4add*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4mix*
- *f16v4mul*
- *f16v4sihoamp*
- *f16v4sihoslic*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4sub*
- *f32absadd*
- *f32add*
- *f32div*
- *f32exp*
- *f32exp2*
- *f32mul*
- *f32sisoamp*
- *f32sisoslic*
- *f32sisov2amp*
- *f32sisov2slic*
- *f32sub*
- *f32tof16*
- *f32v2absadd*
- *f32v2add*
- *f32v2axpy*
- *f32v2gina*
- *f32v2mul*
- *f32v2sub*
- *f32v2tof16*
- *f32v4tof16*

3.8.3 Instructions with broadcast

The following instructions support the *broadcast* operation on at least 1 register source operand:

- *f16v2add*
- *f16v2cmpeq*
- *f16v2cmpge*
- *f16v2cmpgt*
- *f16v2cmple*
- *f16v2cmplt*
- *f16v2cmpne*
- *f16v2mul*
- *f16v2sub*
- *f16v4add*
- *f16v4cmpeq*
- *f16v4cmpge*
- *f16v4cmpgt*
- *f16v4cmple*
- *f16v4cmplt*
- *f16v4cmpne*
- *f16v4mul*
- *f16v4sub*
- *f32v2add*
- *f32v2cmpeq*
- *f32v2cmpge*
- *f32v2cmpgt*
- *f32v2cmple*
- *f32v2cmplt*
- *f32v2cmpne*
- *f32v2mul*
- *f32v2sub*
- *sort4x16hi*
- *sort4x16lo*

3.8.4 Floating-point operations x number format and vector length

Table 3.259: Floating-point operations

Operation	f16	f16v2	f16v4	f16v8	f32	f32v2	f32v4
absacc			✓	✓			✓
acc			✓	✓			✓
aop						✓	
axpy						✓	
cmac		✓	✓				
gacc			✓				
gina		✓				✓	
hihoamp			✓				
hihoslic			✓				
hihov4amp			✓				
hihov4slic			✓				
istacc			✓				
mac					✓	✓	
mix			✓				
sihoamp			✓				
sihoslic			✓				
sisoamp			✓		✓		
sisoslic			✓		✓		
sisov2amp					✓		
sisov2slic					✓		
sqacc				✓			✓
stacc			✓				
grand		✓				✓	
rmask			✓			✓	
absadd		✓	✓		✓	✓	
absmax		✓	✓		✓	✓	
add		✓	✓		✓	✓	
clamp		✓	✓		✓	✓	
class		✓	✓		✓	✓	
div					✓		
exp		✓			✓		
exp2		✓			✓		
int					✓		
ln		✓			✓		
log2		✓			✓		
max		✓	✓		✓	✓	
maxc		✓	✓				
min		✓	✓		✓	✓	
mul		✓	✓		✓	✓	
oorx					✓		
oox					✓		

Continued on next page

Table 3.259 – continued from previous page

Operation	f16	f16v2	f16v4	f16v8	f32	f32v2	f32v4
sigm		✓			✓		
sqrt					✓		
sub		✓	✓		✓	✓	
sum		✓	✓				
tanh		✓			✓		
cmpeq		✓	✓		✓	✓	
cmpge		✓	✓		✓	✓	
cmpgt		✓	✓		✓	✓	
cmple		✓	✓		✓	✓	
cmplt		✓	✓		✓	✓	
cmpne		✓	✓		✓	✓	
fromi32					✓		
fromui32					✓		
sufromui		✓	✓		✓	✓	
tof16					✓	✓	✓
tof32	✓	✓					
toi32					✓		
toui32					✓		

IMPLEMENTATION SPECIFICS

4.1 [IPU2]

4.1.1 General

4.1.1.1 Tile_ZeroExtend

```
1: uint32_t Tile_ZeroExtend(uint32_t value, unsigned bitpos)
2: {
3:     uint32_t maskBits = (1 << bitpos) - 1;
4:     value = value & maskBits;
5:     return value;
6: }
```

4.1.1.2 Tile_SignExtend

```
1: uint32_t Tile_SignExtend(uint32_t value, unsigned bitpos)
2: {
3:     unsigned signBit = (value >> (bitpos - 1)) & 1;
4:     if (signBit) {
5:         uint32_t maskBits = ~0U << bitpos;
6:         value = value | maskBits;
7:     } else {
8:         value = Tile_ZeroExtend(value, bitpos);
9:     }
10:    return value;
11: }
```

4.1.1.3 Tile_ExtractPackedStride

Returns: the sign extended, signed 10-bit value as extracted from *strideRegValue*

```
1: int32_t Tile_ExtractPackedStride(uint32_t strideRegValue, unsigned id)
2: {
3:     switch (id & 0x3) {
4:         case 0: return 1; break;
5:         case 1: return LSU_PACKED_X3_STRIDES__STRIDE0__GET(strideRegValue); break;
6:         case 2: return LSU_PACKED_X3_STRIDES__STRIDE1__GET(strideRegValue); break;
7:         case 3: return LSU_PACKED_X3_STRIDES__STRIDE2__GET(strideRegValue); break;
8:     }
9: }
```

4.1.1.4 Tile_ExtractPackedAddress

Returns: full range memory address extracted from register pair

```
1: uint32_t Tile_ExtractPackedAddress(std::array<uint32_t,2> regPair, unsigned id)
2: {
3:     id &= 0x3; // 2-bit id
4:     uint32_t addr;
5:
6:     switch (id) {
7:         case 1:
8:             addr = regPair[1] & TMEM_FULL_ADDRESS_MASK;
9:             break;
10:
11:         case 2:
12:             // Lower 11 address bits are packed into the msbs of the lower register
13:             // Upper 10 address bits are packed into the msbs of the upper register
14:             addr = ((regPair[0] >> TMEM_BYTE_MAX_ADDRESS_WIDTH) & 0x7ff)
15:                 | (((regPair[1] >> TMEM_BYTE_MAX_ADDRESS_WIDTH) & 0x3ff) << 11);
16:             break;
17:
18:         default:
19:             addr = regPair[0] & TMEM_FULL_ADDRESS_MASK;
20:             break;
21:     }
22:
23:     return addr;
24: }
```

4.1.1.5 Tile_TripleAddressPack_Lower

Returns: register-pair lower register value for triple packed address

```
1: uint32_t Tile_TripleAddressPack_Lower(uint32_t addr[3])
2: {
3:     // Pack the lsbs of the third address into the unused upper bits
4:     return (addr[0] & TMem_FULL_ADDRESS_MASK)
5:         | ((addr[2] & 0x7ff) << TMem_BYTE_MAX_ADDRESS_WIDTH);
6: }
```

4.1.1.6 Tile_TripleAddressPack_Upper

Returns: register-pair upper register value for triple packed address

```
1: uint32_t Tile_TripleAddressPack_Upper(uint32_t addr[3])
2: {
3:     // Pack the msbs of the third address into the unused upper bits
4:     return (addr[1] & TMem_FULL_ADDRESS_MASK)
5:         | (((addr[2] >> 11) & 0x3ff) << TMem_BYTE_MAX_ADDRESS_WIDTH);
6: }
```

4.1.2 Instructions

4.1.2.1 TInstr_GetLatency

```
1: unsigned TInstr_GetLatency(const Opcode opcode, const float src0[], const float src1[], const unsigned brTargetPc)
2: {
3:     // Only worker instructions are supported
4:
5:     unsigned latency = 1;
6:     if (isFtuOpcode(opcode)) {
7:         latency = TInstr_GetFtuLatency(opcode, (float *)src0, (float *)src1);
8:     } else if (isBranchOpcode(opcode) || opcode == Opcode::call_mmmn_zi) {
9:         latency = TInstr_GetBranchLatency(brTargetPc);
10:    } else if (opcode == Opcode::get_mmmn_zi) {
11:        latency = TInstr_GetGetLatency();
12:    }
13:
14:    return latency;
15: }
```

4.1.3 Floating_Point

4.1.3.1 Transcendental, divide, square-root and reciprocal instructions

Table 4.1: Transcendental, divide, square-root and reciprocal instructions

Instruction	Implemented?	Latency	Domain	Monotonic?	Accuracy
<i>f32div</i>	✓	3		✓	<i>TFPU_ACCURATE</i>
<i>f32sqrt</i>	✓	5	$x \in [0, +\infty]$	✓	<i>TFPU_ACCURATE</i>
<i>f32oox</i>	✓	3	$x \in [-\infty, +\infty]$	✓	<i>TFPU_ACCURATE</i>
<i>f32oorx</i>	✓	4	$x \in (0, +\infty]$	✗	<i>TFPU_INACCURATE_1P5</i>
<i>f32log2</i>	✓	6	$x \in [0, +\infty]$	✓	<i>TFPU_INACCURATE_1P5</i>
<i>f32ln</i>	✓	6	$x \in [0, +\infty]$	✓	<i>TFPU_INACCURATE_1P5</i>
<i>f32exp2</i>	✓	3	$x \in [-\infty, +\infty]$	✓	<i>TFPU_INACCURATE_1P5</i>
<i>f32exp</i>	✓	3	$x \in [-\infty, +\infty]$	✓	<i>TFPU_INACCURATE_1P5</i>
<i>f32sigm</i>	✓	5	$x \in [-\infty, +\infty]$	✓	<i>TFPU_INACCURATE_2P5</i>
<i>f32tanh</i>	✓	5	$x \in [-\infty, +\infty]$	✓	<i>TFPU_INACCURATE_1P5</i>
<i>f16v2log2</i>	✓	2	$x \in [0, 65504]$	✓	<i>TFPU_ACCURATE</i>
<i>f16v2ln</i>	✓	2	$x \in [0, 65504]$	✓	<i>TFPU_ACCURATE</i>
<i>f16v2exp2</i>	✓	2	$x \in [-65504, 65504]$	✓	<i>TFPU_ACCURATE</i>
<i>f16v2exp</i>	✓	2	$x \in [-65504, 65504]$	✓	<i>TFPU_ACCURATE</i>
<i>f16v2sigm</i>	✓	2	$x \in [-65504, 65504]$	✓	<i>TFPU_INACCURATE_16P5</i>
<i>f16v2tanh</i>	✓	1	$x \in [-65504, 65504]$	✓	<i>TFPU_ACCURATE</i>

Note: Latency and accuracy figures given are worst case across the entire input domain. The minimum latency for floating point instructions is 1 cycle. See *TInstr_GetLatency* for further details on FTU instruction latency.

4.1.3.2 Parameters

Table 4.2: Floating_Point parameters

Parameter name	Value	Description
TFPU_FP32_FALSE	0	False representation for <i>Single-precision</i> .
TFPU_NUM_ACCUM	32/0x20	The total number of individual accumulators

4.1.3.3 TileRoundMode

Floating point rounding modes.

Table 4.3: Enumeration: TileRoundMode

Identifier	Value	Description
TFPU_ROUND_EVEN	0	Round-to-nearest, ties-to-even.
TFPU_ROUND_POSINF	1	Round-toward-positive-infinity.
TFPU_ROUND_NEGINF	2	Round-toward-negative-infinity.

Continued on next page

Table 4.3 – continued from previous page

Identifier	Value	Description
TFPU_ROUND_ZERO	3	Round-toward-zero.
TFPU_ROUND_AWAY	4	Round-to-nearest, with ties rounded away from zero.

4.1.3.4 TileFPAccuracy

The accuracy of a floating-point operation relative to the infinitely precise result.

Table 4.4: Enumeration: TileFPAccuracy

Identifier	Value	Description
TFPU_ACCURATE	0	The accuracy of the result matches that of an infinitely precise value, rounded accordingly to the result format.
TFPU_INACCURATE_1P5	1	The accuracy of the result is within 1.5 ULPs of the infinitely precise value, rounded accordingly to the result format.
TFPU_INACCURATE_2P5	2	The accuracy of the result is within 2.5 ULPs of the infinitely precise value, rounded accordingly to the result format.
TFPU_INACCURATE_4P5	4	The accuracy of the result is within 4.5 ULPs of the infinitely precise value, rounded accordingly to the result format.
TFPU_INACCURATE_8P5	8	The accuracy of the result is within 8.5 ULPs of the infinitely precise value, rounded accordingly to the result format.
TFPU_INACCURATE_16P5	16	The accuracy of the result is within 16.5 ULPs of the infinitely precise value, rounded accordingly to the result format.

4.1.3.5 TFPU_F32FromBits

32-bit bit-field to float conversion.

Returns: f32 as represented by *bits* with denorms flushed to zero.

```

1: float TFPU_F32FromBits(const uint32_t bits, const bool flushDenorms=true)
2: {
3:     float result;
4:     std::memcpy(&result, &bits, sizeof(result));
5:
6:     if (flushDenorms) {
7:         result = TFPU_FlushFP32DenormToZero(result);
8:     }
9:     return result;
10: }
```

4.1.3.6 TFPU_BitsFromF32

float to raw 32-bit bit-field conversion.

Returns: raw bit representation of f32 input val

```

1: uint32_t TFPU_BitsFromF32(float val, const bool flushDenorms=true)
2: {
3:     uint32_t result;
4:
5:     if (flushDenorms) {
6:         val = TFPU_FlushFP32DenormToZero(val);
7:     }
```



```

8:
9:  std::memcpy(&result, &val, sizeof(result));
10:  return result;
11: }

```

4.1.3.7 TFPU_F32_QNan

Returns: internally generated single-precision quiet NaN

```

1: float TFPU_F32_QNan()
2: {
3:  return TFPU_F32FromBits(TFPU_F32_GEN_QNAN);
4: }

```

4.1.3.8 TFPU_F32_IsQNan

Returns: true if the single-precision value *op0* is a quiet NaN. false otherwise

```

1: bool TFPU_F32_IsQNan(const float op0)
2: {
3:  uint32_t uival;
4:  std::memcpy(&uival, &op0, sizeof(uival));
5:  return std::isnan(op0) && (((uival >> (TFPU_F32_E_OFFSET - 1)) & 1) != 0);
6: }

```

4.1.3.9 TFPU_F32_IsSNan

Returns: true if the single-precision value *op0* is a signaling NaN. false otherwise

```

1: bool TFPU_F32_IsSNan(const float op0)
2: {
3:  uint32_t uival;
4:  std::memcpy(&uival, &op0, sizeof(uival));
5:  return std::isnan(op0) && (((uival >> (TFPU_F32_E_OFFSET - 1)) & 1) == 0);
6: }

```

4.1.3.10 TFPU_RoundFP64ToFmt

Round a double-precision value to half/single-precision accuracy, using specified rounding mode. Note that only round-to-nearest-ties-to-even rounding mode is currently supported.

Returns: the single-precision representation of *value* rounded to half/single-precision (for half-precision, the bottom 13-bits of the significand will be zero).

```

1: float TFPU_RoundFP64ToFmt(double value, TileFloatFormat_t fmt, TileRoundMode_t mode)
2: {
3:  uint64_t sign, mantissa, uexponent;
4:  int sexponent, oflow, effectivePrecision, minNormExp, minDenormExp;
5:
6:  if (TFPU_FP16 == fmt) {
7:    effectivePrecision = TFPU_F16_M_SIZE;
8:    minNormExp = TFPU_F16_MIN_NORM_EXP;
9:    minDenormExp = TFPU_F16_MIN_DENORM_EXP;
10:
11:  } else {
12:    effectivePrecision = TFPU_F32_M_SIZE;
13:    minNormExp = TFPU_F32_MIN_NORM_EXP;
14:    minDenormExp = TFPU_F32_MIN_DENORM_EXP;
15:
16:  }
17:
18:  // Extract fields from double-precision value
19:  TFPU_F64_Decompose(value, &sign, &uexponent, &mantissa);
20:  sexponent = uexponent - TFPU_F64_BIAS;
21:
22:  // Check for fmt denorm range - the precision will effectively
23:  // be reduced for these.
24:  if (sexponent >= (minDenormExp - 1)) {
25:    if (sexponent == (minDenormExp - 1)) {
26:      if (mantissa != 0) {
27:        // Round up to smallest denorm

```

```

28:     uexponent += 1;
29:     mantissa = 0;
30: }
31:
32: } else if (sexponent < minNormExp) {
33:     effectivePrecision -= -(sexponent - minNormExp);
34: }
35:
36: // Round the significand to the required precision
37: mantissa = TFPU_RoundMantissa(mantissa, TFPU_F64_M_SIZE,
38:                               effectivePrecision, TFPU_ROUND_EVEN,
39:                               &oflow);
40:
41: // Did we overflow the mantissa?
42: if (oflow) {
43:     if (TFPU_F64_Exp(value) == TFPU_F64_MAX_NORM_EXP) {
44:         // Round to infinity
45:         uexponent = TFPU_MASK(F64_E);
46:         mantissa = 0;
47:     } else {
48:         uexponent++;
49:     }
50: }
51: }
52: }
53:
54: // Construct a double value, with the reduced mantissa
55: value = TFPU_F64FromFields(sign, uexponent, mantissa);
56:
57: // Convert to float format
58: return static_cast<float>(value);
59: }

```

4.1.3.11 TFPU_RoundFP32ToIntegral

Round a *Single-Precision* value to an integral, rounding according to *mode*.

```

1: float TFPU_RoundFP32ToIntegral(const float op0, TileRoundMode_t mode)
2: {
3:     float rounded;
4:
5:     if (std::isnan(op0)) {
6:         rounded = TFPU_F32_QuietenNan(op0);
7:     }
8:     } else if (std::isinf(op0)) {
9:         // IEEE 754-2008: 6.1
10:        rounded = op0;
11:    }
12:    } else if (op0 > std::exp2(24)) {
13:        // No rounding required for large values (integers)
14:        rounded = op0;
15:    }
16:    } else {
17:        switch (mode) {
18:            case TFPU_ROUND_POSINF:
19:                rounded = std::ceil(op0);
20:                break;
21:
22:            case TFPU_ROUND_NEGINF:
23:                rounded = std::floor(op0);
24:                break;
25:
26:            case TFPU_ROUND_ZERO:
27:                rounded = std::trunc(op0);
28:                break;
29:
30:            case TFPU_ROUND_AWAY:
31:                rounded = std::round(op0);
32:                break;
33:
34:            default: // TFPU_ROUND_EVEN
35:            {
36:                int currentRoundM = std::fegetround();
37:                std::fesetround(FE_TONEAREST);
38:                rounded = std::nearbyint(op0);

```

```

39:     std::fesetround(currentRoundM);
40:     break;
41:   }
42: }
43: }
44:
45: return rounded;
46: }

```

4.1.3.12 TFPU_F32_QuietenNan

Produce a single-precision quiet Nan

```

1: float TFPU_F32_QuietenNan(const float op0)
2: {
3:
4: // Tile produces a fixed qNan bit-pattern, rather than propagating input Nan mantissas
5: // (except in the cases of min/max operations and format conversions)
6: return TFPU_F32_QNan();
7:
8: }

```

4.1.3.13 TFPU_SNanCheck

Check an op for the presence of a signaling NaN. Return invalid operation exception flag if signaling NaN detected.

```

1: TileFPEException_t TFPU_SNanCheck(const float op)
2: {
3:   if (TFPU_F32_IsSNan(op)) {
4:     // IEEE 754-2008: 6.2
5:     return TFPEXCPT_INV;
6:   }
7:
8:   return TFPEXCPT_NONE;
9: }

```

4.1.3.14 TFPU_GenSNanCheck

Check ops for the presence of a signaling NaN. Return invalid operation exception flag if signaling NaN detected.

```

1: TileFPEException_t TFPU_GenSNanCheck(const float ops[], int numOps)
2: {
3:   for (int o = 0; o < numOps; o++) {
4:     if (TFPU_F32_IsSNan(ops[o])) {
5:       // IEEE 754-2008: 6.2
6:       return TFPEXCPT_INV;
7:     }
8:   }
9:
10:  return TFPEXCPT_NONE;
11: }

```

4.1.3.15 TFPU_GenOFLOCheck

Check for overflow.

Returns: TFPEXCPT_NONE if no overflow condition detected. Otherwise returns TFPEXCPT_OFLO or TFPEXCPT_OFLO | TFPEXCPT_INV if nanoo is set.

Parameters:

- *result*: The operation result, to double-precision accuracy
- *format*: The precision of the final result
- *nanoo*: The value of NaN On Overflow (NANOO) flag

```

1: uint32_t TFPU_GenOFLOCheck(double result, TileFloatFormat_t format, bool nanoo)
2: {
3:     uint64_t osign, omantissa, oexponent, nmantissa, nexponent;
4:     double rresult, maxNorm;
5:     int tmSize;
6:     int oflow = 0;
7:
8:     if (std::isnan(result)) {
9:         return TFEXCPT_NONE;
10:    }
11:
12:    if (std::fabs(result) == 0.0) {
13:        return TFEXCPT_NONE;
14:    }
15:
16:    // Extract the various parts of the double-precision result
17:    TFPU_F64_Decompose(result, &osign, &oexponent, &omantissa);
18:
19:    switch (format) {
20:        case TFPU_FP16:
21:            tmSize = TFPU_F16_M_SIZE;
22:            maxNorm = exp2(TFPU_F16_MAX_NORM_EXP + 1) -
23:                exp2(TFPU_F16_MAX_NORM_EXP - TFPU_F16_M_SIZE);
24:            break;
25:
26:        default:
27:            // TFPU_FP32:
28:            tmSize = TFPU_F32_M_SIZE;
29:            maxNorm = std::numeric_limits<float>::max();
30:            break;
31:    }
32:
33:    // Construct a double-precision result with a mantissa rounded to target precision
34:    nexponent = oexponent;
35:    nmantissa = TFPU_RoundMantissa(omantissa, TFPU_F64_M_SIZE, tmSize,
36:        TFPU_ROUND_EVEN, &oflow);
37:
38:    // Did we overflow the mantissa?
39:    if (oflow) {
40:        nexponent++;
41:    }
42:
43:    rresult = TFPU_F64FromFields(osign, nexponent, nmantissa);
44:
45:    if (std::abs(rresult) > maxNorm) {
46:        // IEEE 754-2008: 7.4
47:        if (nanoo) {
48:            return TFEXCPT_OFLO | TFEXCPT_INV;
49:        }
50:        return TFEXCPT_OFLO;
51:    }
52:
53:    return TFEXCPT_NONE;
54: }

```

4.1.3.16 TFPU_AACCResetValue

Returns: false if the accumulator registers \$AACC[n] do not have a well-defined post-reset value. Otherwise returns true and sets *value* to the reset value for all accumulator registers.

```

1: bool TFPU_AACCResetValue(uint32_t &value)
2: {
3:     // Every $AACC register reset to zero
4:     value = 0;
5:     return true;
6: }

```

4.1.3.17 TFPU_AACCReadFlags

Check for exception conditions on accumulator read

Returns: TFEXCPT_NONE if no exception condition detected. Otherwise returns combination of exception flags as appropriate

Parameters:

- *aacc*: The accumulator output value(s)
- *format*: The precision of the final result
- *nanoo*: NANOO mode enable flag

```
1: uint32_t TFPU_AACCReadFlags(std::vector<float> &aacc, TileFloatFormat_t format, bool nanoo)
2: {
3:     uint32_t fpExcpt = TFPEXCPT_NONE;
4:
5:     for (auto iter = aacc.begin(); iter != aacc.end(); ++iter) {
6:         float value = *iter;
7:
8:         if (std::isinf(value)) {
9:             // Assume that the infinity is the result of an arithmetic overflow
10:            // (which is otherwise invisible in Tile's AMP engine)
11:            fpExcpt |= TFPEXCPT_OFLO;
12:
13:            if (TFPU_FP16 == format) {
14:                // For f16 results the infinity result is converted to qNaN
15:                // and Tile sets the INVALID_OPERATION flag
16:                fpExcpt |= TFPEXCPT_INV;
17:            }
18:        } else {
19:            fpExcpt |= TFPU_GenOFLOCheck(value, format, nanoo);
20:        }
21:    }
22:
23:    return fpExcpt;
24: }
```

4.1.3.18 TFPU_F16DotProduct

Perform an inner-product on the two *half-precision* input vectors *x* and *y* as per the Tile AMP hardware.

Returns: the rounded, single-precision result of the inner-product.

Parameters:

- *x*: an array of half-precision floating-point values (represented as single-precision floats)
- *y*: an array of half-precision floating-point values (represented as single-precision floats). The number of elements in *y* must be at least that of *x*.

```
1: float TFPU_F16DotProduct(const std::vector<float> &x, const std::vector<float> &y)
2: {
3: }
```

4.1.3.19 TFPU_DoAddPreExecute

```
1: bool TFPU_DoAddPreExecute(float op0, float op1, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:
5:     if (TFPU_F32_IsSNan(op0) || TFPU_F32_IsSNan(op1)) {
6:         // IEEE 754-2008: 6.2
7:         *fpExcpt |= TFPEXCPT_INV;
8:         *result = TFPU_F32_QNan();
9:     }
10:    } else if (TFPU_F32_IsQNaN(op0) || TFPU_F32_IsQNaN(op1)) {
11:        // IEEE 754-2008: 6.2.3
12:        *result = TFPU_F32_QNan();
13:    }
14:    } else if (_InvCheckADD(op0, op1)) {
15:        *fpExcpt |= TFPEXCPT_INV;
16:        *result = TFPU_F32_QNan();
17:    }
18:    } else if (std::isinf(op0)) {
19:        // IEEE 754-2008: 6.1
20:        *result = op0;
21:    }
22:    } else if (std::isinf(op1)) {
```

```

23: // IEEE 754-2008: 6.1
24: *result = op1;
25:
26: } else {
27:     specialCase = false;
28: }
29:
30: return specialCase;
31: }

```

4.1.3.20 TFPU_Add

```

1: float TFPU_Add(float op0, float op1, TileFloatFormat_t fmt, bool abs=false, bool subNotAdd=false)
2: {
3:     if (TFPU_F32_IsSNan(op0)) {
4:         // IEEE 754-2008: 6.2
5:         return TFPU_F32_QuietenNan(op0);
6:     }
7:     } else if (TFPU_F32_IsQNan(op0)) {
8:         // IEEE 754-2008: 6.2.3
9:         return TFPU_F32_QuietenNan(op0);
10:    }
11:    } else if (TFPU_F32_IsSNan(op1)) {
12:        // IEEE 754-2008: 6.2
13:        return TFPU_F32_QuietenNan(op1);
14:    }
15:    } else if (TFPU_F32_IsQNan(op1)) {
16:        // IEEE 754-2008: 6.2.3
17:        return TFPU_F32_QuietenNan(op1);
18:    }
19:
20:    if (abs) {
21:        // Convert operands to absolute values
22:        op0 = std::fabs(op0);
23:        op1 = std::fabs(op1);
24:    }
25:
26:    if (subNotAdd) {
27:        op1 = -op1;
28:    }
29:
30:    // Denorms are flushed-to-zero
31:    op0 = TFPU_FlushFP32DenormToZero(op0);
32:    op1 = TFPU_FlushFP32DenormToZero(op1);
33:
34:    if (_InvCheckADD(op0, op1)) {
35:        // IEEE 754-2008: 7.2
36:        return TFPU_F32_QNan();
37:    }
38:    } else if (std::isinf(op0)) {
39:        // IEEE 754-2008: 6.1
40:        return op0;
41:    }
42:    } else if (std::isinf(op1)) {
43:        // IEEE 754-2008: 6.1
44:        return op1;
45:    }
46:    } else {
47:        double result = static_cast<double>(op0) + static_cast<double>(op1);
48:        return TFPU_FlushFP32DenormToZero(
49:            TFPU_RoundFP64ToFmt(result, fmt, TFPU_ROUND_EVEN));
50:    }
51: }

```

4.1.3.21 TFPU_DoAxpbyPreExecute

Floating-point exception and special-case checking for $ax + by$ operation.

```

1: bool TFPU_DoAxpbyPreExecute(float a, float x, float b, float y, uint32_t *fpExcpt, float *result)
2: {
3:     // sNan input check
4:     if (TFPU_F32_IsSNan(a) || TFPU_F32_IsSNan(x) ||
5:         TFPU_F32_IsSNan(b) || TFPU_F32_IsSNan(y)) {
6:         // IEEE 754-2008: 6.2

```

```

7:     *fpExcpt |= TFPEXCPT_INV;
8:     *result = TFPU_F32_QNaN();
9:     return true;
10:
11: // inf * 0 check
12: } else if ((std::isinf(a) && std::fabs(x) == 0.0) ||
13:           (std::isinf(x) && std::fabs(a) == 0.0) ||
14:           (std::isinf(b) && std::fabs(y) == 0.0) ||
15:           (std::isinf(y) && std::fabs(b) == 0.0)) {
16:     // IEEE 754-2008: 7.2
17:     *fpExcpt |= TFPEXCPT_INV;
18:     *result = TFPU_F32_QNaN();
19:     return true;
20:
21: // qNaN input check
22: } else if (TFPU_F32_IsQNaN(a) || TFPU_F32_IsQNaN(x) ||
23:           TFPU_F32_IsQNaN(b) || TFPU_F32_IsQNaN(y)) {
24:     // IEEE 754-2008: 6.2.3
25:     *result = TFPU_F32_QNaN();
26:     return true;
27: }
28:
29: int sia = std::signbit(a);
30: int sib = std::signbit(b);
31: int six = std::signbit(x);
32: int siy = std::signbit(y);
33: int sii0 = sia ^ six;
34: int sii1 = sib ^ siy;
35:
36: if ( (std::isinf(a) || std::isinf(x))
37:     && (std::isinf(b) || std::isinf(y))
38:     && (sii0 != sii1) ) {
39:     // Attempted addition of opposite signed infinities
40:     // IEEE 754-2008:
41:     *result = TFPU_F32_QNaN();
42:     return true;
43: }
44:
45: // No check for magnitude subtraction of infinities
46:
47: // No check for overflow into accumulators
48:
49: return false;
50: }

```

4.1.3.22 TFPU_DoMacPreExecute

```

1: bool TFPU_DoMacPreExecute(float x, float y, float z, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:
5:     int six = std::signbit(x);
6:     int siy = std::signbit(y);
7:     int siz = std::signbit(z);
8:     int sii = six ^ siy;
9:
10:    if (TFPU_F32_IsSNan(x) || TFPU_F32_IsSNan(y)) {
11:        // IEEE 754-2008: 6.2
12:        *fpExcpt |= TFPEXCPT_INV;
13:        *result = TFPU_F32_QNaN();
14:    }
15:    } else if (TFPU_F32_IsQNaN(x) || TFPU_F32_IsQNaN(y)) {
16:        // IEEE 754-2008: 6.2.3
17:        *result = TFPU_F32_QNaN();
18:    }
19:    } else if (std::isinf(x) && (fabs(y) == 0.0)) {
20:        // IEEE 754-2008: 7.2 (c)
21:        // Raise INVALID_OP even when accum is qNaN
22:        *fpExcpt |= TFPEXCPT_INV;
23:        *result = TFPU_F32_QNaN();
24:    }
25:    } else if (std::isinf(y) && (fabs(x) == 0.0)) {
26:        // IEEE 754-2008: 7.2 (c)
27:        // Raise INVALID_OP even when z is qNaN
28:        *fpExcpt |= TFPEXCPT_INV;

```

```

29:     *result = TFPU_F32_QNan();
30:
31: } else if (TFPU_F32_IsSNan(z) || TFPU_F32_IsQNan(z)) {
32:     // IEEE 754-2008: 6.2.3
33:     *result = TFPU_F32_QNan();
34:
35: } else if (std::isinf(x) && std::isinf(z) && (sii != siz)) {
36:     // IEEE 754-2008: 7.2 (d)
37:     *result = TFPU_F32_QNan();
38:
39: } else if (std::isinf(y) && std::isinf(z) && (sii != siz)) {
40:     // IEEE 754-2008: 7.2 (d)
41:     *result = TFPU_F32_QNan();
42:
43: } else {
44:     specialCase = false;
45: }
46:
47: // No check for magnitude subtraction of infinities
48: // (IEEE 754-2008: 7.2(d))
49:
50: return specialCase;
51: }

```

4.1.3.23 TFPU_Mac

```

1: float TFPU_Mac(float x, float y, float z, TileFloatFormat_t fmt, TileRoundMode_t rmode)
2: {
3:     int six = std::signbit(x);
4:     int siy = std::signbit(y);
5:     int siz = std::signbit(z);
6:     int sii = six ^ siy;
7:
8:     if (TFPU_F32_IsSNan(x)) {
9:         // IEEE 754-2008: 6.2
10:        return TFPU_F32_QuietenNan(x);
11:
12:    } else if (TFPU_F32_IsQNan(x)) {
13:        // IEEE 754-2008: 6.2.3
14:        return TFPU_F32_QuietenNan(x);
15:
16:    } else if (TFPU_F32_IsSNan(y)) {
17:        // IEEE 754-2008: 6.2
18:        return TFPU_F32_QuietenNan(y);
19:
20:    } else if (TFPU_F32_IsQNan(y)) {
21:        // IEEE 754-2008: 6.2.3
22:        return TFPU_F32_QuietenNan(y);
23:
24:    } else if (TFPU_F32_IsSNan(z)) {
25:        // IEEE 754-2008: 6.2
26:        return TFPU_F32_QuietenNan(z);
27:
28:    } else if (TFPU_F32_IsQNan(z)) {
29:        // IEEE 754-2008: 6.2.3
30:        return TFPU_F32_QuietenNan(z);
31:
32:    } else if (std::isinf(x)) {
33:        if (fabs(y) == 0.0) {
34:            // IEEE 754-2008: 7.2 (c)
35:            return TFPU_F32_QNan();
36:
37:        } else if ((std::isinf(z) && (sii != siz)) {
38:            // IEEE 754-2008: 7.2 (d)
39:            return TFPU_F32_QNan();
40:
41:        }
42:    } else if (std::isinf(y)) {
43:        if (fabs(x) == 0.0) {
44:            // IEEE 754-2008: 7.2 (c)
45:            return TFPU_F32_QNan();
46:
47:        } else if ((std::isinf(z) && (sii != siz)) {
48:            // IEEE 754-2008: 7.2 (d)
49:            return TFPU_F32_QNan();

```



```

50:
51:     }
52: }
53:
54: // MAC isn't "fused" in the sense that
55: // Tile rounds to the accumulator format
56: // prior to the addition.
57: float mRes = TFPU_Mul(x, y, TFPU_FP32, rmode);
58: return TFPU_Add(mRes, z, TFPU_FP32);
59: }

```

4.1.3.24 TFPU_DoMulPreExecute

```

1: bool TFPU_DoMulPreExecute(float op0, float op1, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:
5:     if (TFPU_F32_IsSNan(op0) || TFPU_F32_IsSNan(op1)) {
6:         // IEEE 754-2008: 6.2
7:         *fpExcpt |= TFPEXCPT_INV;
8:         *result = TFPU_F32_QNan();
9:
10:    } else if (TFPU_F32_IsQNaN(op0) || TFPU_F32_IsQNaN(op1)) {
11:        // IEEE 754-2008: 6.2.3
12:        *result = TFPU_F32_QNan();
13:
14:    } else if (isinf(op0)) {
15:        if (fabs(op1) == 0.0) {
16:            // IEEE 754-2008: 7.2
17:            *fpExcpt |= TFPEXCPT_INV;
18:            *result = TFPU_F32_QNan();
19:
20:        } else {
21:            bool op2IsNeg = signbit(op1);
22:            if (op2IsNeg) {
23:                // IEEE 754-2008: 6.1
24:                *result = -op0;
25:
26:            } else {
27:                // IEEE 754-2008: 6.1
28:                *result = op0;
29:            }
30:        }
31:
32:    } else if (isinf(op1)) {
33:        if (fabs(op0) == 0.0) {
34:            // IEEE 754-2008: 7.2
35:            *fpExcpt |= TFPEXCPT_INV;
36:            *result = TFPU_F32_QNan();
37:
38:        } else {
39:            bool op1IsNeg = signbit(op0);
40:            if (op1IsNeg) {
41:                // IEEE 754-2008: 6.1
42:                *result = -op1;
43:
44:            } else {
45:                // IEEE 754-2008: 6.1
46:                *result = op1;
47:            }
48:        }
49:
50:    } else {
51:        specialCase = false;
52:    }
53:
54:    return specialCase;
55: }

```

4.1.3.25 TFPU_Mul

```

1: double TFPU_Mul(float op0, float op1, TileFloatFormat_t fmt, TileRoundMode_t rmode)
2: {
3:     bool isNeg0 = std::signbit(op0);

```

```

4:  bool isNeg1 = std::signbit(op1);
5:
6:  if (TFPU_F32_IsSNaN(op0)) {
7:      // IEEE 754-2008: 6.2
8:      return TFPU_F32_QuietenNan(op0);
9:
10: } else if (TFPU_F32_IsQNaN(op0)) {
11:     // IEEE 754-2008: 6.2.3
12:     return TFPU_F32_QuietenNan(op0);
13:
14: } else if (TFPU_F32_IsSNaN(op1)) {
15:     // IEEE 754-2008: 6.2
16:     return TFPU_F32_QuietenNan(op1);
17:
18: } else if (TFPU_F32_IsQNaN(op1)) {
19:     // IEEE 754-2008: 6.2.3
20:     return TFPU_F32_QuietenNan(op1);
21:
22: } else if (_InvCheckMUL(op0, op1)) {
23:     return TFPU_F32_QNaN();
24:
25: } else if (std::isinf(op0)) {
26:     if (isNeg1) {
27:         // IEEE 754-2008: 6.1
28:         return -op0;
29:
30:     } else {
31:         // IEEE 754-2008: 6.1
32:         return op0;
33:
34:     }
35: } else if (std::isinf(op1)) {
36:     if (isNeg0) {
37:         // IEEE 754-2008: 6.1
38:         return -op1;
39:
40:     } else {
41:         // IEEE 754-2008: 6.1
42:         return op1;
43:
44:     }
45: } else {
46:     double result = static_cast<double>(op0) * static_cast<double>(op1);
47:
48:     if ( (TFPU_FP16 == fmt) || (TFPU_FP32 == fmt) ) {
49:         return TFPU_RoundFP64ToFmt(result, fmt, rmode);
50:
51:     } else {
52:         return result;
53:     }
54: }
55: }

```

4.1.3.26 TFPU_DoDivPreExecute

```

1:  bool TFPU_DoDivPreExecute(float op0, float op1, uint32_t *fpExcpt, float *result)
2:  {
3:      bool specialCase = true;
4:      bool isNeg0 = std::signbit(op0);
5:      bool isNeg1 = std::signbit(op1);
6:
7:      if (TFPU_F32_IsSNaN(op0) || TFPU_F32_IsSNaN(op1)) {
8:          // IEEE 754-2008: 6.2
9:          *fpExcpt |= TFPEXCPT_INV;
10:         *result = TFPU_F32_QNaN();
11:
12:     } else if (TFPU_F32_IsQNaN(op0) || TFPU_F32_IsQNaN(op1)) {
13:         // IEEE 754-2008: 6.2.3
14:         *result = TFPU_F32_QNaN();
15:
16:     } else if (std::isinf(op0)) {
17:         if (std::isinf(op1)) {
18:             // IEEE 754-2008: 7.2
19:             *fpExcpt |= TFPEXCPT_INV;
20:             *result = TFPU_F32_QNaN();

```

```

21:
22:     } else if (isNeg0 == isNeg1) {
23:         // IEEE 754-2008: 6.1
24:         // Signs are the same - positive result
25:         *result = std::fabs(op0);
26:
27:     } else {
28:         // Signs are opposite, result is negative
29:         *result = std::copysign(op0, -1);
30:
31:     }
32:
33: } else if (std::isinf(op1)) {
34:     // IEEE 754-2008: 6.1
35:     if (isNeg0 == isNeg1) {
36:         *result = 0.0;
37:
38:     } else {
39:         *result = -0.0;
40:     }
41:
42: } else if (fabs(op1) == 0.0) {
43:     if (fabs(op0) == 0.0) {
44:         // IEEE 754-2008: 7.2 (e)
45:         *fpExcpt |= TFPEXCPT_INV;
46:         *result = TFPU_F32_QNaN();
47:
48:     } else {
49:         // IEEE 754-2008: 7.3
50:         *fpExcpt |= TFPEXCPT_DIV0;
51:         if (isNeg0 == isNeg1) {
52:             *result = std::numeric_limits<float>::infinity();
53:         } else {
54:             *result = -std::numeric_limits<float>::infinity();
55:         }
56:     }
57:
58: } else {
59:     specialCase = false;
60: }
61:
62: return specialCase;
63: }

```

4.1.3.27 TFPU_DoSqrtPreExecute

```

1: bool TFPU_DoSqrtPreExecute(float op0, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:     if (TFPU_F32_IsSNan(op0)) {
5:         // IEEE 754-2008: 6.2
6:         *fpExcpt |= TFPEXCPT_INV;
7:         *result = TFPU_F32_QNaN();
8:
9:     } else if (TFPU_F32_IsQNaN(op0)) {
10:        // IEEE 754-2008: 6.2.3
11:        *result = TFPU_F32_QNaN();
12:
13:    } else if (op0 == -0.0) {
14:        // IEEE 754-2008: 5.4.1/6.3
15:        *result = op0;
16:
17:    } else if (std::signbit(op0)) {
18:        // IEEE 754-2008: 7.2
19:        *fpExcpt |= TFPEXCPT_INV;
20:        *result = TFPU_F32_QNaN();
21:
22:    } else if (std::isinf(op0)) {
23:        // IEEE 754-2008: 6.1
24:        *result = op0;
25:
26:    } else {
27:        specialCase = false;
28:    }
29:

```

```
30: return specialCase;
31: }
```

4.1.3.28 TFPU_DoRecipPreExecute

```
1: bool TFPU_DoRecipPreExecute(float op0, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:
5:     if (TFPU_F32_IsSNan(op0)) {
6:         // IEEE 754-2008: 6.2
7:         *fpExcpt |= TFPEXCPT_INV;
8:         *result = TFPU_F32_QNan();
9:     }
10:    } else if (TFPU_F32_IsQNan(op0)) {
11:        // IEEE 754-2008: 6.2.3
12:        *result = TFPU_F32_QNan();
13:    }
14:    } else if (fabs(op0) == 0.0) {
15:        // IEEE 754-2008: 9.2.1
16:        *fpExcpt |= TFPEXCPT_DIV0;
17:        *result = std::copysign(std::numeric_limits<float>::infinity(), op0);
18:    }
19:    } else if (std::isinf(op0)) {
20:        *result = std::copysign(0.0, op0);
21:    }
22:    } else {
23:        specialCase = false;
24:    }
25:
26:    return specialCase;
27: }
```

4.1.3.29 TFPU_DoRSqrtPreExecute

```
1: bool TFPU_DoRSqrtPreExecute(float op0, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:     if (TFPU_F32_IsSNan(op0)) {
5:         // IEEE 754-2008: 9.2
6:         *fpExcpt |= TFPEXCPT_INV;
7:         *result = TFPU_F32_QNan();
8:     }
9:    } else if (TFPU_F32_IsQNan(op0)) {
10:        // IEEE 754-2008: 6.2
11:        *result = TFPU_F32_QNan();
12:    }
13:    } else if (fabs(op0) == 0.0) {
14:        // IEEE 754-2008: 9.2
15:        *fpExcpt |= TFPEXCPT_DIV0;
16:        *result = std::copysign(std::numeric_limits<float>::infinity(), op0);
17:    }
18:    } else if (op0 < 0.0) {
19:        // IEEE 754-2008: 7.2
20:        *fpExcpt |= TFPEXCPT_INV;
21:        *result = TFPU_F32_QNan();
22:    }
23:    } else if (std::isinf(op0)) {
24:        // IEEE 754-2008: 9.2.1
25:        *result = 0.0;
26:    }
27:    } else {
28:        specialCase = false;
29:    }
30:
31:    return specialCase;
32: }
```

4.1.3.30 TFPU_DoExpPreExecute

```
1: bool TFPU_DoExpPreExecute(float op0, uint32_t *fpExcpt, float *result)
2: {
```

```

3:  bool specialCase = true;
4:  if (TFPU_F32_IsSNan(op0)) {
5:      // IEEE 754-2008: 6.2
6:      *fpExcpt |= TFPEXCPT_INV;
7:      *result = TFPU_F32_QNan();
8:  }
9:  } else if (TFPU_F32_IsQNan(op0)) {
10:     // IEEE 754-2008: 6.2
11:     *result = TFPU_F32_QNan();
12:  }
13:  } else if (std::isinf(op0)) {
14:     if (std::signbit(op0) == 0) {
15:         // IEEE 754-2008: 9.2.1
16:         *result = op0;
17:     } else {
18:         *result = 0.0;
19:     }
20:  }
21:  } else {
22:     specialCase = false;
23:  }
24:
25:  return specialCase;
26: }

```

4.1.3.31 TFPU_DoLogPreExecute

```

1:  bool TFPU_DoLogPreExecute(float op0, uint32_t *fpExcpt, float *result)
2:  {
3:     bool specialCase = true;
4:     if (TFPU_F32_IsSNan(op0)) {
5:         // IEEE 754-2008: 6.2
6:         *fpExcpt |= TFPEXCPT_INV;
7:         *result = TFPU_F32_QNan();
8:     }
9:     } else if (TFPU_F32_IsQNan(op0)) {
10:        // IEEE 754-2008: 6.2
11:        *result = TFPU_F32_QNan();
12:    }
13:    } else if (std::fabs(op0) == 0.0) {
14:        // IEEE 754-2008: 9.2.1
15:        *fpExcpt |= TFPEXCPT_DIV0;
16:        *result = -std::numeric_limits<float>::infinity();
17:    }
18:    } else if (op0 < 0.0) {
19:        // IEEE 754-2008 Table 9.1 and section 7.2
20:        *fpExcpt |= TFPEXCPT_INV;
21:        *result = TFPU_F32_QNan();
22:    }
23:    } else if (std::isinf(op0)) {
24:        // IEEE 754-2008: 9.2.1
25:        *result = std::numeric_limits<float>::infinity();
26:    }
27:    } else {
28:        specialCase = false;
29:    }
30:
31:    return specialCase;
32: }

```

4.1.3.32 TFPU_DoTanhPreExecute

```

1:  bool TFPU_DoTanhPreExecute(float op0, uint32_t *fpExcpt, float *result)
2:  {
3:     bool specialCase = true;
4:     if (TFPU_F32_IsSNan(op0)) {
5:         // IEEE 754-2008: 6.2
6:         *fpExcpt |= TFPEXCPT_INV;
7:         *result = TFPU_F32_QNan();
8:     }
9:     } else if (TFPU_F32_IsQNan(op0)) {
10:        // IEEE 754-2008: 6.2
11:        *result = TFPU_F32_QNan();
12:    }

```

```

13: } else if (std::fabs(op0) == 0.0) {
14:     // IEEE 754-2008: 9.2.1
15:     *result = op0;
16:
17: } else {
18:     specialCase = false;
19: }
20:
21: return specialCase;
22: }

```

4.1.3.33 TFPU_DoSigmoidPreExecute

```

1: bool TFPU_DoSigmoidPreExecute(float op0, uint32_t *fpExcpt, float *result)
2: {
3:     bool specialCase = true;
4:
5:     if (TFPU_F32_IsSNan(op0)) {
6:         // IEEE 754-2008: 6.2
7:         *fpExcpt |= TFPEXCPT_INV;
8:         *result = TFPU_F32_QuietenNan(op0);
9:
10:    } else if (TFPU_F32_IsQNan(op0)) {
11:        // IEEE 754-2008: 6.2.3
12:        *result = TFPU_F32_QuietenNan(op0);
13:
14:    } else if (isinf(op0)) {
15:        // IEEE 754-2008: 9.2.1
16:        bool op1IsNeg = signbit(op0);
17:        if (op1IsNeg) {
18:            *result = 0.0;
19:        } else {
20:            *result = 1.0;
21:        }
22:
23:    } else if (fabs(op0) == 0.0) {
24:        // IEEE 754-2008: 9.2.1
25:        *result = 0.5;
26:
27:    } else if (fabs(op0) < exp2(-24)) {
28:        // inexact 0.5
29:        *result = 0.5;
30:
31:    } else if (op0 < -104.0) {
32:        // values in this range will return an inexact zero
33:        *result = 0.0;
34:
35:    } else if (op0 >= 18.0) {
36:        // values in this range will return an inexact one
37:        *result = 1.0;
38:
39:    } else {
40:        specialCase = false;
41:    }
42:
43:    return specialCase;
44: }

```

4.1.3.34 TFPU_F32Sigmoid

```

1: float TFPU_F32Sigmoid(float op0, TileRoundMode_t rmode)
2: {
3:     return _FTU_F32sigm(op0, rmode);
4: }

```

4.1.3.35 TFPU_F16Sigmoid

```

1: float TFPU_F16Sigmoid(float op0, TileRoundMode_t rmode)
2: {
3:     return _FTU_F16sigm(op0, rmode);
4: }

```

4.1.3.36 TFPU_F32Exp

```
1: float TFPU_F32Exp(float op0, TileFPBase_t base, TileRoundMode_t rmode)
2: {
3:   if (TFPU_BASE_2 == base) {
4:     return _FTU_F32exp2(op0, rmode);
5:   } else {
6:     return _FTU_F32exp(op0, rmode);
7:   }
8: }
```

4.1.3.37 TFPU_F16Exp

```
1: float TFPU_F16Exp(float op0, TileFPBase_t base, TileRoundMode_t rmode)
2: {
3:   if (TFPU_BASE_2 == base) {
4:     return _MPFR_F16exp2(op0, rmode);
5:   } else {
6:     return _MPFR_F16exp(op0, rmode);
7:   }
8: }
```

4.1.3.38 TFPU_F32Log

```
1: float TFPU_F32Log(float op0, TileFPBase_t base, TileRoundMode_t rmode)
2: {
3:   if (TFPU_BASE_2 == base) {
4:     return _FTU_F32log2(op0, rmode);
5:   } else {
6:     return _FTU_F32ln(op0, rmode);
7:   }
8: }
```

4.1.3.39 TFPU_F16Log

```
1: float TFPU_F16Log(float op0, TileFPBase_t base, TileRoundMode_t rmode)
2: {
3:   if (TFPU_BASE_2 == base) {
4:     return _MPFR_F16log2(op0, rmode);
5:   } else {
6:     return _MPFR_F16ln(op0, rmode);
7:   }
8: }
```

4.1.3.40 TFPU_F32Tanh

```
1: float TFPU_F32Tanh(float op0, TileRoundMode_t rmode)
2: {
3:   return _FTU_F32tanh(op0, rmode);
4: }
```

4.1.3.41 TFPU_F16Tanh

```
1: float TFPU_F16Tanh(float op0, TileRoundMode_t rmode)
2: {
3:   return _MPFR_F16tanh(op0, rmode);
4: }
```

4.1.3.42 TFPU_F32Recip

```
1: float TFPU_F32Recip(float op0, TileRoundMode_t rmode)
2: {
3:   return _MPFR_F32recip(op0, rmode);
4: }
```

4.1.3.43 TFPU_F32Sqrt

```
1: float TFPU_F32Sqrt(float op0, TileRoundMode_t rmode)
2: {
3:     double res = sqrt(static_cast<double>(op0));
4:     return TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode);
5: }
```

4.1.3.44 TFPU_F32RSqrt

```
1: float TFPU_F32RSqrt(float op0, TileRoundMode_t rmode)
2: {
3:     return _FTU_F32rsqrt(op0, rmode);
4: }
```

4.1.3.45 TFPU_Min

Determine the minimum of two single-precision values.

Returns: A quietened *op0* if *op0* is a signaling NaN. A quietened *op1* if *op1* is a signaling NaN. *op0* if *op1* is a quiet NaN. *op1* if *op0* is a quiet NaN. Otherwise returns the (non NaN) minimum of *op0* and *op1*.

```
1: float TFPU_Min(float op0, float op1)
2: {
3:     bool isNeg0 = std::signbit(op0);
4:     bool isNeg1 = std::signbit(op1);
5:
6:     if (TFPU_F32_IsSNan(op0)) {
7:         // IEEE 754-2008: 6.2
8:         return TFPU_F32_QuietenNan(op0);
9:     }
10:    } else if (TFPU_F32_IsSNan(op1)) {
11:        // IEEE 754-2008: 6.2
12:        return TFPU_F32_QuietenNan(op1);
13:    }
14:    } else if (TFPU_F32_IsQNaN(op1)) {
15:        // IEEE 754-2008: 5.3.1
16:        return op0;
17:    }
18:    } else if (TFPU_F32_IsQNaN(op0)) {
19:        // IEEE 754-2008: 5.3.1
20:        return op1;
21:    }
22:    } else if (std::isinf(op0)) {
23:        // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
24:        // Also min(a,a) = a
25:        if (isNeg0) {
26:            return op0;
27:        } else {
28:            return op1;
29:        }
30:    }
31:    } else if (std::isinf(op1)) {
32:        // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
33:        if (isNeg1) {
34:            return op1;
35:        } else {
36:            return op0;
37:        }
38:    }
39:    } else if (std::abs(op0) == std::abs(op1)) {
40:        if (isNeg1) {
41:            // if a is positive, min(-a,-a) = -a; min(a,-a) = -a (including a == 0.0)
42:            return op1;
43:        } else {
44:            return op0;
45:        }
46:    }
47:    } else {
48:        return (op0 < op1 ? op0 : op1);
49:    }
50: }
```


4.1.3.46 TFPU_Max

Determine the maximum of two single-precision values.

Returns: A quietened *op0* if *op0* is a signaling NaN. A quietened *op1* if *op1* is a signaling NaN. *op0* if *op1* is a quiet NaN. *op1* if *op0* is a quiet NaN. Otherwise returns the (non NaN) maximum of *op0* and *op1*.

```
1: float TFPU_Max(float op0, float op1)
2: {
3:     bool isNeg0 = std::signbit(op0);
4:     bool isNeg1 = std::signbit(op1);
5:
6:     if (TFPU_F32_IsSNan(op0)) {
7:         // IEEE 754-2008: 6.2
8:         return TFPU_F32_QuietenNan(op0);
9:     }
10:    } else if (TFPU_F32_IsSNan(op1)) {
11:        // IEEE 754-2008: 6.2
12:        return TFPU_F32_QuietenNan(op1);
13:    }
14:    } else if (TFPU_F32_IsQNaN(op1)) {
15:        // IEEE 754-2008: 5.3.1
16:        return op0;
17:    }
18:    } else if (TFPU_F32_IsQNaN(op0)) {
19:        // IEEE 754-2008: 5.3.1
20:        return op1;
21:    }
22:    } else if (std::isinf(op0)) {
23:        // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
24:        // Also max(a,a) = a
25:        if (isNeg0) {
26:            return op1;
27:        } else {
28:            return op0;
29:        }
30:    }
31:    } else if (std::isinf(op1)) {
32:        // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
33:        if (isNeg1) {
34:            return op0;
35:        } else {
36:            return op1;
37:        }
38:    }
39:    } else if (std::abs(op0) == std::abs(op1)) {
40:        if (isNeg1) {
41:            return op0;
42:        } else {
43:            // if a is positive, max(-a,-a) = -a; max(-a, a) = a (including a == 0.0)
44:            return op1;
45:        }
46:    }
47:    } else {
48:        return (op0 > op1 ? op0 : op1);
49:    }
50: }
```

4.1.3.47 TFPU_Relation

```
1: TileFPRelation_t TFPU_Relation(float op0, float op1)
2: {
3:     if (std::isnan(op0)) {
4:         return TFPU_RELATION_UN;
5:     }
6:     } else if (std::isnan(op1)) {
7:         return TFPU_RELATION_UN;
8:     }
9:     } else if (std::isinf(op0)) {
10:    } if (std::isinf(op1)) {
11:        if (std::signbit(op0) == std::signbit(op1)) {
12:            // IEEE 754-2008: 5.11
13:            return TFPU_RELATION_EQ;
14:        }
15:    } else if (std::signbit(op0) == 0) {
```

```

16:     // IEEE 754-2008: 6.1
17:     return TFPU_RELATION_GT;
18:
19: } else {
20:     // IEEE 754-2008: 6.1
21:     return TFPU_RELATION_LT;
22:
23: }
24: } else if (std::signbit(op0) == 0) {
25:     // IEEE 754-2008: 6.1
26:     return TFPU_RELATION_GT;
27:
28: } else {
29:     // IEEE 754-2008: 6.1
30:     return TFPU_RELATION_LT;
31:
32: }
33: } else if (std::isinf(op1)) {
34:     if (std::signbit(op1) == 0) {
35:         // IEEE 754-2008: 6.1
36:         return TFPU_RELATION_LT;
37:
38:     } else {
39:         return TFPU_RELATION_GT;
40:
41:     }
42: } else if ((fabs(op0) == 0.0) && (fabs(op1) == 0.0)) {
43:     // IEEE 754-2008: 5.11
44:     return TFPU_RELATION_EQ;
45:
46: } else if (op0 > op1) {
47:     return TFPU_RELATION_GT;
48:
49: } else if (op0 < op1) {
50:     return TFPU_RELATION_LT;
51:
52: } else {
53:     return TFPU_RELATION_EQ;
54: }
55: }

```

4.1.3.48 TFPU_F32DivExceptIsImprecise

```

1: bool TFPU_F32DivExceptIsImprecise(uint32_t fpExcpt, uint32_t fpCtl)
2: {
3:     if ((fpExcpt == TFPEXCPT_OFLO) &&
4:         ((fpCtl & TFPEXCPT_OFLO) != 0)) {
5:         return true;
6:     }
7:
8:     return false;
9: }

```

4.1.3.49 TFPU_IsMalign

```

1: bool TFPU_IsMalign(uint32_t fpExcpt, uint32_t fpCtl)
2: {
3:     // $FP_CTL indicates that the floating-point exception is to be treated as malign.
4:     return ((fpExcpt & fpCtl) != 0);
5: }

```

4.1.4 Exceptions

4.1.4.1 Imprecise Exceptions

The only imprecise exception raised by *IPU2* is overflow (*TFPEXCPT_OFLO*), detected during the execution of *f32div*.

4.1.4.2 Super-imprecise Exceptions

The only Super-imprecise exception raised by *IPU2* is an uncorrectable ECC memory error on load instructions executed by the Supervisor context.

In this scenario, there is a maximum of 5 subsequent issue slots in which a store instruction could commit its data to *Tile Memory*. The actual number may be less than 5, depending on the presence of pipeline bubbles.

4.1.4.3 RBRK and imprecise-exceptions

If a Retirement BREAK request is active during the retirement phase of an instruction that would raise an imprecise exception, the RBRK *exception event* will be launched. On recovery from the RBRK *exception event*, the imprecise *exception event* will not be raised. Note that this only affects overflow (*TFPEXCPT_OFLO*), detected during the execution of *f32div*, in which case the overflow status flag *\$FP_STS.OFLO* will be set at the launch of the RBRK.

4.1.4.4 TileException

Tile exception identifiers.

Table 4.5: Enumeration: TileException

Identifier	Value	Description
TEXCPT_MEMERR	14	Memory error exception
TEXCPT_INVALID_INSTR	12	Invalid instruction exception

4.1.5 Contexts

4.1.5.1 TileCtxtStatus

Context status brief.

Table 4.6: Enumeration: TileCtxtStatus

Identifier	Value	Description
TCTXT_STATUS_INACTIVE	0	Worker context <i>run-mode</i> is <i>Inactive</i>
TCTXT_STATUS_ACTIVE	1	Context <i>run-mode</i> is not any of those covered by the other TCTXT_STATUS enumerations.
TCTXT_STATUS_EXCEPTED_DBG	2	Context <i>run-mode</i> is <i>TRUNM_EXCEPTED</i> (having raised a debug exception)
TCTXT_STATUS_EXCEPTED_NDBG	3	Context <i>run-mode</i> is <i>TRUNM_EXCEPTED</i> (having raised a non-debug exception)

4.1.6 Memory

4.1.6.1 Memory regions

IPU2's implemented memory space is split into two *Memory regions* of unequal size.

Table 4.7: *IPU2* Memory Regions

Region	Base address	Size	Interleave factor	Executable?	Comments
0	0x4c000	208KBytes	1	✓	Fully populated
1	0x80000	416KBytes	2	✗	Fully populated

4.1.6.2 Memory clashes

In addition to the memory access restrictions specified by the *Tile* architecture, (*Memory clashes*) data load/store accesses on *IPU2* have the potential to cause memory clashes with instruction fetches from the same context. The precise conditions are dependent on the context type:

The following memory access restriction applies to *Worker* contexts only:

- For data-accesses to/from region 0 only, the timing of instruction fetch means that the memory-element id of any such data access (*TMem_ElementId*), initiated by a memory instruction must not match the memory-element id of all addresses in the range:
 - [*\$REPEAT_FIRST*, *\$REPEAT_END*], when *\$REPEAT_COUNT* is non-zero
 - [*\$PC* + 4, *\$PC* + 16] otherwise

If a data access is made that violates this restriction, the memory instruction will raise a *TEXCPT_CONFLICT exception event*.

The following memory access restriction applies to *Supervisor* contexts only:

- For data-accesses to/from region 0 only, the timing of instruction fetch means that the memory-element id of any data access (*TMem_ElementId*), initiated by a memory instruction must not match the memory-element id of all addresses in the range [*PC* + 4, *PC* + (8 * 8)]. If a data access is made that violates this restriction, the memory instruction will raise a *TEXCPT_CONFLICT exception event*.

No such restriction exists for supervisor data accesses to region 1 since it's not possible to perform instruction fetches from that region.

4.1.6.3 Striding support

The *pace* instructions use a stride register which contains a set of strides packed into the 32-bit register value.

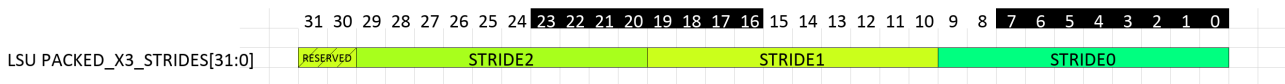


Fig. 4.1: `LSU_PACKED_X3_STRIDES` data-type format

Table 4.8: `LSU_PACKED_X3_STRIDES` data-type fields

Field name	Bit field	Description
STRIDE0	[9:0]	The first of three strides packed into a 32-bit register.
STRIDE1	[19:10]	The second of three strides packed into a 32-bit register.
STRIDE2	[29:20]	The third of three strides packed into a 32-bit register.

4.1.6.4 Parameters

Table 4.9: Memory parameters

Parameter name	Value	Unit	Description
TMEM_BASE_ADDR	311296/0x4c000	bytes	The logical base address of <i>Tile Memory</i> (in bytes).
TMEM_BASE_ADDR_WORD	77824/0x13000	words	The logical base address of <i>Tile Memory</i> , in 32-bit words.
TMEM_ATOMSIZE	64/0x40	Bits	Tile Memory read/write port widths (the natural access size).
TMEM_ELEMSIZE	16/0x10	KiBytes	The size of a single Tile Memory element.
TMEM_REGION0_SIZE	208/0xd0	KiBytes	The logical address space of <i>Tile Memory</i> region 0.
TMEM_REGION1_SIZE	416/0x1a0	KiBytes	The logical address space of <i>Tile Memory</i> region 1.
TMEM_SIZE	624/0x270	KiBytes	The total logical capacity of Tile Memory, per Tile instance.
TMEM_SIZE_WORDS	159744/0x27000	words	The total logical capacity of Tile Memory, per Tile instance, in 32-bit words.
TMEM_REGION0_BASE_ADDR	311296/0x4c000	bytes	The logical base address of <i>Tile Memory</i> region 0.
TMEM_FULL_ADDRESS_MASK	0x1fffff	Bits	Architectural (and therefore constant across all implementations) address space mask.
TMEM_NUM_REGIONS	2	Regions	The total number of distinct memory regions within the Tile Memory space.

4.1.6.5 TMem_RegionId

Pre-conditions: TMem_IsValidAddress(address) == true

Returns: the region Id of *address*

```
1: int32_t TMem_RegionId(uint32_t address)
2: {
3:     // |tilerev| has 2 memory regions. Region 1 is [0x80000,0xe7fff]
4:     return (address >> 19) & 1;
5: }
```

4.1.6.6 TMem_ElementId

Pre-conditions: TMem_IsValidAddress(address) == true

Returns: the absolute element Id of the *Tile Memory* address *address*.

```
1: uint32_t TMem_ElementId(uint32_t address)
2: {
3:     int32_t regionId = TMem_RegionId(address);
4:     uint32_t regionBase = TMem_RegionBaseAddress(regionId);
5:
6:     // The address -> element-id mapping is dependent on the interleave factor at address
7:     if (TMem_AddressInterleaveFactor(address) == 1) {
8:         return (address - regionBase) >> TMEM_ELEM_OFFSET_SIZE;
9:     } else {
10:         return (((address - regionBase) >> (TMEM_ELEM_OFFSET_SIZE + 1)) << 1)
11:             | ((address >> 3) & 1) + (TMem_RegionSizeKBytes(0) / TMEM_ELEMSIZE);
12:     }
```

```
13:
14: }
15: }
```

4.1.6.7 TMem_ElementOffset

Pre-conditions: TMem_IsValidAddress(address) == true

Returns: the offset of address within its memory element

```
1: uint32_t TMem_ElementOffset(uint32_t address)
2: {
3:     int32_t regionId = TMem_RegionId(address);
4:     uint32_t regionBase = TMem_RegionBaseAddress(regionId);
5:
6:     // Offset within element is dependent on the interleave factor of the memory region
7:     if (TMem_AddressInterleaveFactor(address) == 1) {
8:         return (address - regionBase) & ((1 << TMem_ELEM_OFFSET_SIZE) - 1);
9:
10:    } else {
11:        address = address - regionBase;
12:        return (address & 0x7)
13:            | (((address >> 4) & ((1 << (TMem_ELEM_OFFSET_SIZE - 3)) - 1)) << 3);
14:
15:    }
16: }
```

4.1.6.8 TMem_IsValidAddress

Returns: true iff *address* is within the valid *Tile Memory* address range. Note that any (unpopulated) area of memory below TMem_BASE_ADDR is considered invalid.

```
1: bool TMem_IsValidAddress(uint32_t address)
2: {
3:     return (address >= TMem_BASE_ADDR) && (address < (TMem_BASE_ADDR + (TMem_SIZE * 1024)));
4: }
```

4.1.6.9 TMem_RegionBaseAddress

Returns: 0 if *regionId* is invalid. Otherwise returns the base address of the specified memory region.

```
1: uint32_t TMem_RegionBaseAddress(int regionId)
2: {
3:     if (0 == regionId) {
4:         return TMem_REGION0_BASE_ADDR;
5:     } else if (1 == regionId) {
6:         return TMem_REGION0_BASE_ADDR + (TMem_RegionSizeKBytes(0) * 1024);
7:     }
8:
9:     return 0;
10: }
```

4.1.6.10 TMem_RegionInterleaveFactor

Returns: 0 if *regionId* is invalid. Otherwise returns the *interleave factor* of the specified memory region.

```
1: uint32_t TMem_RegionInterleaveFactor(int regionId)
2: {
3:     if (0 == regionId) {
4:         return 1;
5:     } else if (1 == regionId) {
6:         return 2;
7:     }
8:
9:     return 0;
10: }
```

4.1.6.11 TMem_AddressInterleaveFactor

Returns: 0 if the address is not contained within any particular region. Otherwise returns the *interleave factor* of the region containing *address*.

```
1: uint32_t TMem_AddressInterleaveFactor(uint32_t address)
2: {
3:     return TMem_RegionInterleaveFactor(TMem_RegionId(address));
4: }
```

4.1.6.12 TMem_RegionIsExecutable

Returns: true iff *Tile* can execute instructions from the specified memory region.

```
1: bool TMem_RegionIsExecutable(int regionId)
2: {
3:     return (TMem_RegionInterleaveFactor(regionId) == 1);
4: }
```

4.1.6.13 TMem_AddressIsExecutable

Returns: true iff *address* is within an executable region of memory.

```
1: bool TMem_AddressIsExecutable(uint32_t address)
2: {
3:     return TMem_RegionIsExecutable(TMem_RegionId(address)) && (address >= TMem_BASE_ADDR);
4: }
```


4.1.7 Registers

4.1.7.1 TileRFAccessSize

The width of a read or write access to a Tile register-file.

Table 4.10: Enumeration: TileRFAccessSize

Identifier	Value	Description
TRF_ACCESS_SIZE_SINGLE	1	Access to a single 32-bit register
TRF_ACCESS_SIZE_PAIR	2	Access to a naturally aligned register pair
TRF_ACCESS_SIZE_QUAD	4	Access to a naturally aligned register quad

4.1.7.2 TReg_RFIndices

Parameters:

- *indices*: is an empty vector to be populated with the register-file indices
- *baseIndex*: is the raw register field value extracted from the instruction encoding
- *size*: is the effective size of the register operand

```
1: void TReg_RFIndices(std::vector<unsigned> &indices, unsigned baseIndex, TileRFAccessSize_t size)
2: {
3:     switch (size) {
4:         case TRF_ACCESS_SIZE_QUAD:
5:             // Only supported for writes to ARF
6:             // Mask out the lsbs
7:             baseIndex &= ~0x3;
8:             break;
9:
10:        case TRF_ACCESS_SIZE_PAIR:
11:            // Supported for reads from and writes to the ARF and MRF
12:            // Mask out the lsbs
13:            baseIndex &= ~0x1;
14:            break;
15:
16:        default:
17:            // Supported for reads from and writes to the ARF and MRF
18:            // Single register access - no alignment issues
19:            break;
20:    }
21:
22:    indices.push_back(baseIndex);
23:
24:    if (size > TRF_ACCESS_SIZE_SINGLE) {
25:        indices.push_back(baseIndex | 1);
26:
27:        if (size > TRF_ACCESS_SIZE_PAIR) {
28:            indices.push_back(baseIndex | 2);
29:            indices.push_back(baseIndex | 3);
30:        }
31:    }
32: }
```

4.1.7.3 TReg_IsValidCCCS

```
1: bool TReg_IsValidCCCS(uint32_t index)
2: {
3:     if (index < (TREG_CCCS_WEIGHT_GROUP_SIZE * TREG_CCCS_NUM_WEIGHT_REG_GROUP)) {
4:         return true;
5:     } else {
6:         return false;
7:     }
8: }
```

4.1.7.4 TReg_WriteException

```
1: TileException_t TReg_WriteException(uint32_t index, bool supervisor)
2: {
3:
4:     if (!supervisor) {
5:         switch (index) {
6:             case 0: /* $PC */
7:             case 1: /* $WSR */
8:             case 2: /* $VERTEX_BASE */
9:             case 3: /* $WORKER_BASE */
10:            case 4: /* $REPEAT_COUNT */
11:            case 5: /* $REPEAT_FIRST */
12:            case 6: /* $REPEAT_END */
13:            case 96: /* $COUNT_L */
14:            case 97: /* $COUNT_U */
15:            case 112: /* $DBG_DATA */
16:            case 113: /* $DBG_BRK_ID */
17:            case 256: /* $FP_STS */
18:            case 257: /* $FP_CLR */
19:            case 258: /* $FP_CTL */
20:            case 259: /* $PRNG_0_0 */
21:            case 260: /* $PRNG_0_1 */
22:            case 261: /* $PRNG_1_0 */
23:            case 262: /* $PRNG_1_1 */
24:            case 263: /* $PRNG_SEED */
25:            case 264: /* $TAS */
26:                return TEXCPT_NONE;
27:            }
28:        }
29:
30:        // Default for unknown registers is INVALID_OP exception
31:        return TEXCPT_INVALID_OP;
32:    }
```

4.1.7.5 TReg_IsValidCSR

```
1: bool TReg_IsValidCSR(uint32_t index, bool supervisor)
2: {
3:
4:     if (!supervisor) {
5:         switch (index) {
6:             case 0: /* $PC */
7:             case 1: /* $WSR */
8:             case 2: /* $VERTEX_BASE */
9:             case 3: /* $WORKER_BASE */
10:            case 4: /* $REPEAT_COUNT */
11:            case 5: /* $REPEAT_FIRST */
12:            case 6: /* $REPEAT_END */
13:            case 96: /* $COUNT_L */
14:            case 97: /* $COUNT_U */
15:            case 112: /* $DBG_DATA */
16:            case 113: /* $DBG_BRK_ID */
17:            case 256: /* $FP_STS */
18:            case 257: /* $FP_CLR */
19:            case 258: /* $FP_CTL */
20:            case 259: /* $PRNG_0_0 */
21:            case 260: /* $PRNG_0_1 */
22:            case 261: /* $PRNG_1_0 */
23:            case 262: /* $PRNG_1_1 */
24:            case 263: /* $PRNG_SEED */
25:            case 264: /* $TAS */
26:                return true;
27:            }
28:        }
29:        return false;
30:    }
```

4.1.7.6 MRF

4.1.7.6.1 Parameters

Table 4.11: MRF parameters

Parameter name	Value	Description
MRF_GP_REGISTERS	12/0xc	The total number of populated MRF general-purpose registers, per context

4.1.7.6.2 TReg_MRFRresetValue

Returns: false if the populated MRF registers do not have a well-defined post-reset value. Otherwise returns true and sets *value* to the reset value for all populated MRF registers.

```

1: bool TReg_MRFRresetValue(uint32_t &value)
2: {
3:     // Every populated MRF register reset to zero
4:     value = 0;
5:     return true;
6: }
```

4.1.7.7 ARF

4.1.7.7.1 Parameters

Table 4.12: ARF parameters

Parameter name	Value	Description
ARF_GP_REGISTERS	8	The total number of populated ARF general-purpose registers, per context

4.1.7.7.2 TReg_ARFRresetValue

Returns: false if the populated ARF registers do not have a well-defined post-reset value. Otherwise returns true and sets *value* to the reset value for all populated ARF registers.

```

1: bool TReg_ARFRresetValue(uint32_t &value)
2: {
3:     // Every populated ARF register reset to zero
4:     value = 0;
5:     return true;
6: }
```


BIBLIOGRAPHY

[IEEE754] IEEE Std 754TM-2008 <http://ieeexplore.ieee.org/document/4610935/>

A

Accumulation, 52
 Address format, 36
 ARF, 4, 17
 Atomic Sections, 4
 aux, 4, 8, 31

B

Barrier Synchronisation, 4
 Benign, 58
 BREAK, 4
 BSP, 4

C

Co-issue, 10
 Codelet, 4
 Colossus, 4
 Commit, 4
 Compute, 4
 Context, 5
 Contexts, 8
 CSR, 5, 19

D

Debug, 59
 Delta, 37
 Delta Offset, 37
 Delta Pointer, 37
 Deltas, 37

E

ECC, 5
 Endianness, 37
 Except In, 5
 Except Out, 5
 Exception, 5
 Exception Event, 5
 Exception event, 58
 Exceptions, 58
 Exchange, 60
 Exchange Fabric, 5
 Exchange Phase, 5
 Execution Bundle, 5, 10
 Execution pipeline, 8
 External Exchange, 5

F

f16, 5
 f16v2, 5

f16v4, 5
 f16v8, 5
 f32, 5
 f32v2, 5
 f32v4, 5
 FAULT, 5
 Floating point exceptions, 51
 Format conversion, 49

G

get, 19

H

Half-precision, 5

I

Immediate, 5
 Imprecise, 5, 367
 Inactive, 5
 Instruction execution, 9
 Instruction fetch, 9, 39
 Instruction issue, 9
 Instruction retirement, 9
 Interleave factor, 5
 Internal Exchange, 5
 Internal state, 31
 IPU, 5
 IPU2, 343
 ISA, 5

M

main, 5, 8
 Malign, 58
 Memory, 5
 Memory errors, 40
 Memory map, 37
 Memory protection, 39
 Memory region, 369
 MRF, 5, 14

N

Naturally Aligned, 5

P

Patched Breakpoint, 5
 PC, 5
 ports, 16, 18
 Precise, 6
 Prepare, 6

PRNG, 61
Program order, 9
put, 19

Q

Quiescence, 13
Quiescent, 6

R

Receive, 6
Register File, 6
Register file, 14, 17
Register model, 14
Registers, 12, 14, 31
Retirement, 6
Rounding, 48
Run mode, 12

S

Sibling instruction, 6
Sign extended, 6
Single-precision, 6
Superstep, 6
Supervisor, 6, 8

T

TDI, 6
Thread, 6
Tile, 6
Transcendental, 56, 57, 347

U

Undefined, 6

V

Vertex, 6
Vertex state, 6

W

Word, 6
Worker, 6, 8

Z

Zero extended, 6
Zero tailed, 6

Trademarks & Copyright

Graphcore® and Poplar® are Registered Trademarks of Graphcore Ltd.

© Copyright 2016 - 2022, Graphcore Ltd