GRAPHCORE

Tile Vertex ISA

Release 1.3.1

IPU21

Dec 16, 2022

TABLE OF CONTENTS

CHAPTER

ONE

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 1.3.1

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:

- *IPU21*

1.3 Conventions

1.3.1 Arithmetic Notation and Operators

The textual descriptions use the following notation:

Table 1.2: Arithmetic operators

Symbol	Meaning
0x*value*	A value written using hexadecimal notation
0b*value*	A value written using binary notation
value	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

Active A *worker* state when it is in any *run mode* other than *Inactive*.

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*.

BFloat16 A 16-bit floating-point number format. See *Number Formats*

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 A phase of an *instruction lifespan*.

Execution Bundle A set of independently executable instructions that are issued together. See *Issue Group*.

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 16-bit floating-point value, in either *half-precision* or *BFloat16* format

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.

f8 A *quarter-precision* floating-point value.

f8v4 A 4-element vector of 8-bit floating-point values.

f8v8 An 8-element vector of 8-bit floating-point values.

FAULT An unrecoverable exception event.

Fetch The phase of *instruction lifespan* where instruction encodings are read from memory.

ff32 A mode of operation for certain *single-precision* operations where the 12 least significant bits of the mantissa are ignored (assumed to be zero).

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 event*s 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*

Issue The phase of an *instruction lifespan* where instructions are sent to execution units.

Issue Group Term to describe a set of one or more instructions that are issued together. See *Issue Group*.

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

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

Memory element The *64-bit wide building block* of the *Tile* memory.

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 event*s 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.

Quarter-precision A union 8-bit floating-point number format. See *Number Formats*

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 *instruction lifespan*.

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*.

Suspended The *run mode* the *supervisor* is in when all *worker*s are active.

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.

CHAPTER

TWO

MACHINE FRAMEWORK

2.1 Overview

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

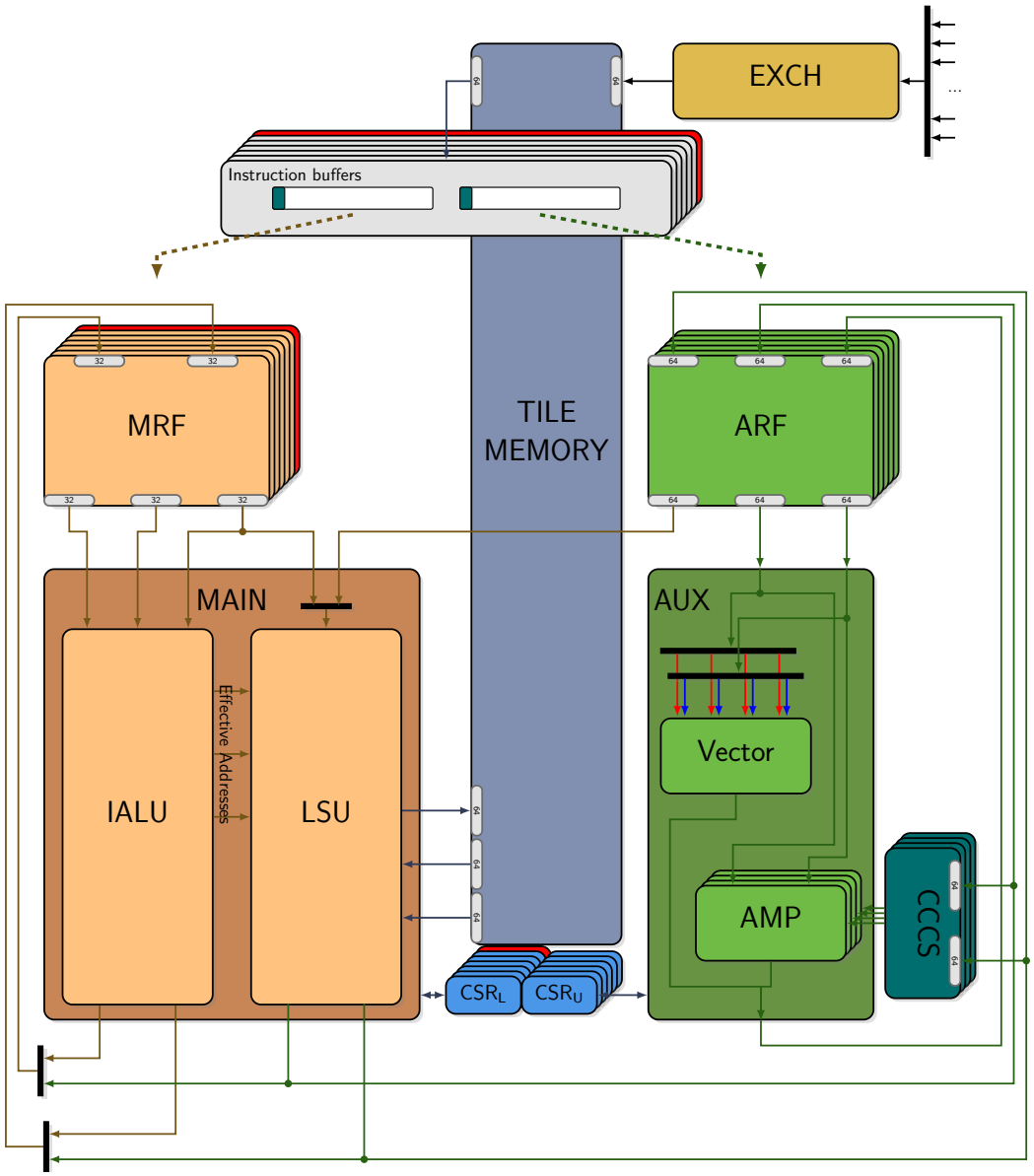


Fig. 2.1: *Tile* logical block structure

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* and *runall* instructions. Once all execution slots have been allocated to *worker*s, 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: The *supervisor* context cannot use the *aux* pipeline and its associated state.

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

- executable by the *main* pipeline only
- executable by the *aux* pipeline only
- executable by both execution pipelines[1]

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* 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:

- 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 TEXCPT_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 it 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.
- *atom* takes it source register operand from *ARF* and writes its result to *MRF*.

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

In addition to the *ARF*, the *aux* 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

2.4.1 Issue Group

Instructions are issued in *issue group*s which must be one of:

- Solo instructions to either pipeline. The execution pipeline is determined from the instruction opcode. Any attempt to execute a solo *aux* instruction by the *supervisor* context will result in a TEXCPT_INVALID_INSTR *exception*.
- An *execution bundle* containing two instructions. Each instruction in an *execution bundle* is executed independently, in parallel by the appropriate pipeline (see *Co-issue*).

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

Note: Since the *supervisor* context cannot use the *aux* pipeline, it is also incapable of parallel instruction execution.

2.5 Program Order

To define program order, the lifespan of instructions in an *issue group* are broken down into a number of phases:

1. *Instruction fetch* where the instruction encodings are read from *Tile Memory*.
2. *Instruction issue* where the instructions are dispatched to execution units 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 event*s. 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 event*s.

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

For all contexts, *issue group*s are always executed in the sequence defined by the updates of *$PC*. All *issue group*s that contain no *control class* instructions implicitly increment the value of *$PC* by the size of the *issue group* during *retirement*. Solo instructions increment the *$PC* by 4. Solo instructions with payload and *execution bundle*s increment the *$PC* by 8.

Where the *issue group* contains a *control class* instruction, the new value of *$PC* to be committed at *retirement* is defined by the control instruction semantics.

2.5.1 Co-issue

Worker contexts support *issue group*s that are either solo instructions or *execution bundle*s.

The semantics of parallel execution of an *execution bundle* are as follows:

- All *tile* architectural state is with that resulting from the *retirement* phase of the previous *issue group*.
- There is no guaranteed relative ordering of *retirement* between the co-issued instructions.
- *Tile* architectural state for the next *issue group* will be consistent with that resulting from the *retirement* phase for both co-issued instructions.
- *Precise exception*s raised by one execution pipeline will squash all architectural updates associated with **both** co-issued instructions.
- *Imprecise exception*s (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 exception*s (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:

- A *Co-issue* field value of 0 indicates that this instruction is not part of an *execution bundle*
- 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 or solo instructions with payload.
2. The *supervisor* context can only issue instructions to the *main* pipeline.

3. 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*.
4. 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*.
5. 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 *exception* being raised.

Table 2.1: Co-issue behaviour for adjacent instructions

Instruction *i*	**Instruction *i+1***	**Supervisor action**	**Worker action**
0 main	x main/aux	Solo issue of instruction *i* to main pipeline	If executing a repeat-loop, TEXCPT_INVALID_INSTR *exception* is raised. Otherwise, solo issue of instruction *i* to *main*
0 aux	x main/aux	TEXCPT_INVALID_INSTR *exception* raised	If executing a repeat-loop, TEXCPT_INVALID_INSTR *exception* is raised. Otherwise, solo issue of instruction *i* to *aux*
1 main	0 main/aux	TEXCPT_INVALID_INSTR *exception* raised	TEXCPT_INVALID_INSTR *exception* raised
1 main	1 main	TEXCPT_INVALID_INSTR *exception* raised	TEXCPT_INVALID_INSTR *exception* raised
1 main	1 aux	TEXCPT_INVALID_INSTR *exception* raised	Issue of *execution bundle* (comprised of *main* instruction *i* and *aux* instruction *i+1*)
1 aux	x main/aux	TEXCPT_INVALID_INSTR *exception* raised	TEXCPT_INVALID_INSTR *exception* raised

2.6 Worker Timing

The number of cycles taken by *worker* instructions to complete is defined by the *TInstr_GetLatency* function. If instructions are being issued together in an *execution bundle* then they will complete together after the maximum number of cycles of all instructions in the *execution bundle*.

2.7 Execution Semantics

2.7.1 Register Read Values

The value of all source registers is that following the *retirement* phase of the previous *issue group*.

Source register values for the *supervisor*'s first *issue group* are consistent with their architecturally defined reset values.

Source register values for *worker*'s first *issue group* are consistent with their architecturally defined reset values unless they are changed by the *supervisor*.

2.7.2 Run Modes

At any point in time a *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 *worker*s. The *worker* is not executing instructions or participating in any other type of activity. The *context* will remain *Inactive* until either: 1. it is allocated an execution slot by the *supervisor* context executing a *run* instruction, or 2. an instruction is *injected* into the *worker* context Entry to the *Inactive* run mode is via the *exit* instructions.
Executing	✗	The context is actively issuing instructions. *Worker*s are placed into the *Executing* run mode by the *supervisor* context executing *run*.
Excepted	✓	The *worker* has raised an *exception event*. For certain types of exception it is possible to transition back into the *Executing* run mode by clearing down (recovering from) the exception.
Repeating	✗	The *worker* has executed a *rpt* instruction and *$REPEAT_COUNT* is non-zero. The *worker* run mode will transition back to *Executing* when *$REPEAT_COUNT* becomes 0.
Executing injected instruction	✗	A *worker* will transition into this state from a quiescent run mode whenever an *injected* instruction for this *worker* enters the pipeline. If the injected instruction causes an *exception event* the run mode will transition to *Excepted*. Otherwise the run mode will transition back to its previous state.

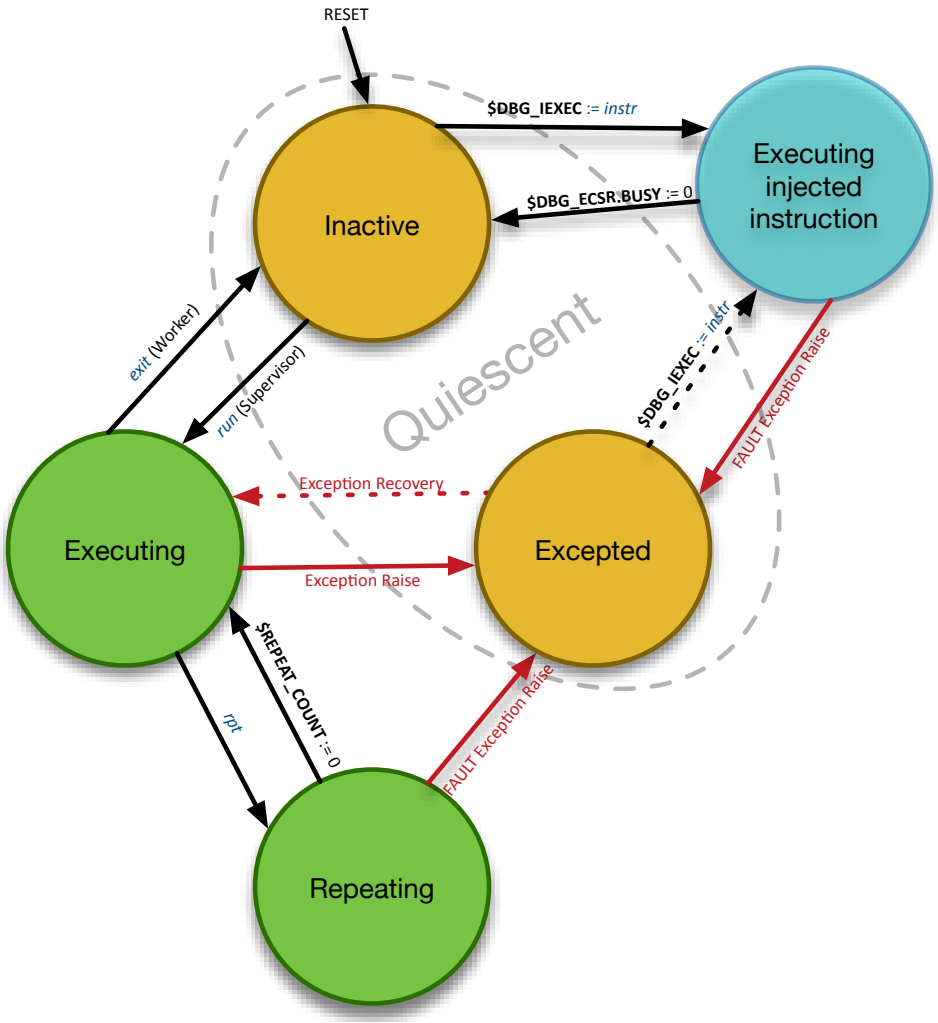


Fig. 2.2: Worker Context Run Mode Transitions

2.7.3 Quiescence

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

2.8 Register Model

2.8.1 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.	✗	$WSR.CTXTID_M1
RvO	Ignored	Reads yield current value of dynamic state.	✓	$COUNT_L.VALUE
RW	Write value committed to state.	Reads yield last value written (or reset value if never written to).	✗	*MRF*
RvW	Write value committed to state.	Reads yield current value of dynamic state.	✓	$FP_CTL.INV
WC	Write value committed to state.	Yields 0	✗	$PRNG_SEED.VALUE
W1C	Write of 0b1 initiates a background operation.	Yields 0	✗	$FP_CLR.INV
WO	Write value committed to state.	Not applicable. There is no mechanism available to directly read this state. It may be possible to read the state indirectly.	✗	*$CWEI_n_0*
RESERVED	Ignored	Yields 0	✗	*$PC*.**RESERVED**

2.8.2 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 applicable[2], 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* only and is implicitly used (read) by the *worker*s when performing certain operations. This common state is typically used when all *worker*s 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*'s architectural state.

2.8.3 MRF

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

1. provides register (or *constant valued*) source operand values to the *main execution pipeline*
2. accepts writes 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] The *supervisor* cannot issue instructions to the *aux* pipeline and so doesn't include an *ARF*. Any attempt made by the *supervisor* to execute an instruction that would require access to the *ARF* will result in a TEXCPT_INVALID_INSTR *exception*.

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. The behaviour of reads is *implementation dependent*. For example, reads may yield 0 or the contents of one of the populated indices.

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	*RW*	*Populated* region. Reset value is *implementation dependent* and given by *TReg_MRFResetValue*.
$m1	1			
$m2	2			
...	...			
$m*N-1*	*N-1*			
$m*N*	*N*		*RO*. Reads yield *implementation dependent* values.	*Unpopulated* region. Does not exist when MRF_GP_REGISTERS is 12.
$m*N+1*	*N+1*			
$m12	12		*RO*. Reads by *worker*s return *$WORKER_BASE*. Reads by *supervisor* return 0.	*Constant valued* register.
$m13	13		*RO*. Reads by *worker*s return *$VERTEX_BASE*. Reads by *supervisor* return 0.	*Constant valued* register.
$m14	14		*RO*. Reads yield *implementation dependent* values.	
$m15	15		*RO*. Reads return 0.	*Constant valued* 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*.

2.8.3.1 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

2.8.4 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*. 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)

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

Table 2.5: ARF

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

Note: *M* is ARF_GP_REGISTERS

Table 2.6: ARF as an array of 64-bit registers

Assembler Syntax	Register Indices	Width (bits)	Semantics	Notes
$ap0	*0:1*	64	*RW*	Populated region. Reset value is implementation dependent and given by *TReg_ARFResetValue*.
$ap1	*2:3*			
...				
$ap*M*	*2*M*:2*M+1*		*RO*. Reads yield *implementation dependent* values.	Unpopulated region. Will not exist if *ARF_GP_REGISTERS* >= 15.
...				
$ap15			*RO*. Reads return 0.	*Constant valued* register.

Note: *M* is ARF_GP_REGISTERS >> 1.

Note: In order maintain backwards compatibility, the assembler will map `$a14:15` to `$ap15`, which is the zero register pair. To utilise `$a14` and `$a15`, `$ap7` should be used instead.

Table 2.7: ARF as an array of 128-bit registers

Assembler Syntax	Register Indices	Width (bits)	Semantics	Notes
$aq0	*0:3*	128	*RW*	Populated region. Reset value is implementation dependent and given by *TReg_ARFResetValue*.
$aq1	*4:7*			
...				
$aq*M*	*4*M*:4*M+3*		*RO*. Reads yield *implementation dependent* values.	Unpopulated region. Will not exist if *ARF_GP_REGISTERS* >= 14.
...				
$aq15			*RO*. Reads return 0.	*Constant valued* register.

Note: *M* is ARF_GP_REGISTERS >> 2.

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*.

2.8.4.1 ARF Read and Write Ports

In order to satisfy all instruction source and destination register operand requirements efficiently, the ARF of *IPU21* provides:

- 3 x 64-bit independent read ports
- 3 x 64-bit independent write ports[3]

2.8.5 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 *worker*s (see *Worker CSRs*).

2.8.5.1 Supervisor CSRs

[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 2.8: Supervisor Control and Status registers

Index	Register name	Reset value	Description
0x20	*$FP_ICTL*	0x00000000	Floating-point control register initial value.
0x31	*$FP_INFMT*	0x00000000	Floating-point number format control register initial value.
0x33	*$FP_ISCL*	0x00000000	Floating-point instruction scale factor initial values.
0x50	*$CCCSLOAD*	0x00000000	Post-incrementing load address for *ld64putcs* and *ld128putcs* instructions.
0x72	*$CTXT_STS*	0x00000000	Context execution status register.

2.8.5.1.1 $FP_ICTL

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*

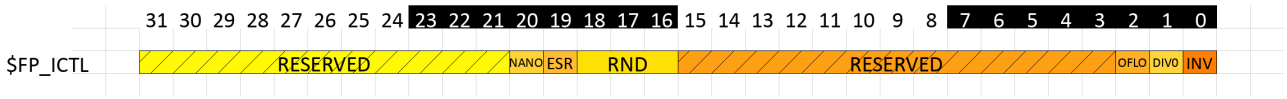


Fig. 2.3: **$FP_ICTL** register format

Table 2.9: **$FP_ICTL** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
INV	[0]	0x0	*RW*	• 0b0: Floating-point invalid operation exceptions treated as benign. • 0b1: Floating-point invalid operation exceptions treated as malign.
DIV0	[1]	0x0	*RW*	• 0b0: Floating-point divide-by-zero exceptions treated as benign. • 0b1: Floating-point divide-by-zero exceptions treated as malign.
OFLO	[2]	0x0	*RW*	• 0b0: Floating-point overflow exceptions treated as benign. • 0b1: Floating-point overflow exceptions treated as malign.

Continued on next page

Table 2.9 – continued from previous page

Field name	Bit field	Reset value	*Access semantics*	Description
RND	[18:16]	0x0	*RO*	Floating-point rounding behaviour. See *TileRoundMode*. **Note:** Not all rounding modes are supported by all implementations (see *Parameters*). Attempts to write values corresponding to RESERVED or unimplemented modes will be ignored. Reads will yield non-RESERVED, implemented modes.
ESR	[19]	0x0	*RW*	Enable stochastic rounding for those instructions that support it.
NANOO	[20]	0x0	*RW*	Enable NAN On Overflow mode. When enabled *half-precision* calculations that have overflowed will: 1. produce a quiet NaN result, rather than saturating to the f16 maximum/minimum values. 2. Set the invalid operation flag

2.8.5.1.2 $FP_INFMT

Register index: 49

Floating-point number format control register initial value. The Worker context floating-point number format control register *$FP_NFMT* is initialised with the contents of this register prior to worker activation.

See *run* and *runall*

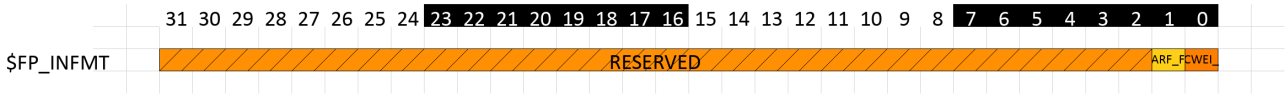


Fig. 2.4: **$FP_INFMT** register format

Table 2.10: **$FP_INFMT** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
CWEI_FMT	[0]	0x0	*RW*	Define the format of the CWEI inputs to the AMP unit for f8 operations • 0b0: f8 CWEI operands are treated as *Quart152* values • 0b1: f8 CWEI operands are treated as *Quart143* values

Continued on next page

Table 2.10 – continued from previous page

Field name	Bit field	Reset value	*Access semantics*	Description
ARF_FMT	[1]	0x0	*RW*	Define the format of the ARF based *quarter-precision* inputs to certain format conversion and AMP based operations • 0b0: f8 ARF operands are treated as *Quart152* values • 0b1: f8 ARF operands are treated as *Quart143* values

2.8.5.1.3 $FP_ISCL

Register index: 51

Floating-point instruction scale factor initial values.

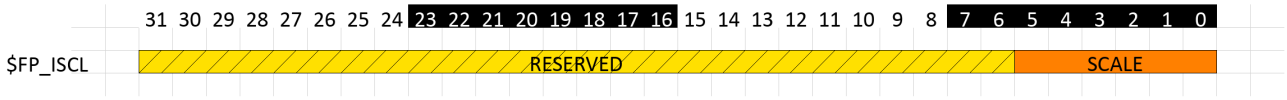


Fig. 2.5: **$FP_ISCL** register format

Table 2.11: **$FP_ISCL** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
SCALE	[5:0]	0x0	*RW*	Signed scale exponent for certain instructions with *half-precision* or *quarter-precision* input or output vectors.

2.8.5.1.4 $CCCSLOAD

Register index: 80

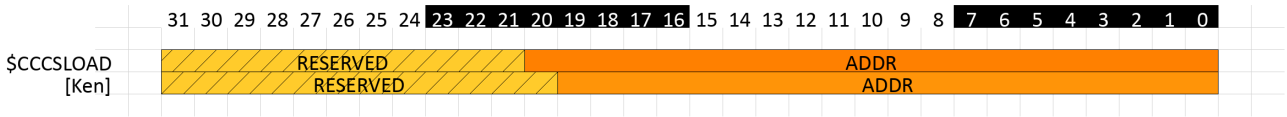


Fig. 2.6: **$CCCSLOAD** register format

Table 2.12: **$CCCSLOAD** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:0]	0x0	*RvW*	Byte address

$CCCSLOAD register field sizes are influenced by the following parameter: TMEM_BYTE_ADDRESS_WIDTH

2.8.5.1.5 $CTXT_STS

Register index: 114

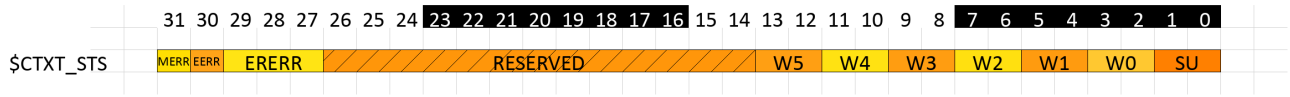


Fig. 2.7: **$CTXT_STS** register format

Table 2.13: **$CTXT_STS** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
SU	[1:0]	0x0	*RvO*	Supervisor context status, as specified by *TileCtxtStatus*
W0	[3:2]	0x0	*RvO*	Worker context 0 status, as specified by *TileCtxtStatus*
W1	[5:4]	0x0	*RvO*	Worker context 1 status, as specified by *TileCtxtStatus*
W2	[7:6]	0x0	*RvO*	Worker context 2 status, as specified by *TileCtxtStatus*
W3	[9:8]	0x0	*RvO*	Worker context 3 status, as specified by *TileCtxtStatus*
W4	[11:10]	0x0	*RvO*	Worker context 4 status, as specified by *TileCtxtStatus*
W5	[13:12]	0x0	*RvO*	Worker context 5 status, as specified by *TileCtxtStatus*
ERERR	[29:27]	0x0	*RvO*	Exchange receive error status flag.
EERR	[30]	0x0	*RvO*	Exchange error status flag • 0b0: No exchange error detected • 0b1: Exchange parity error detected
MERR	[31]	0x0	*RvO*	Memory error status flag • 0b0: No memory error detected • 0b1: Memory ECC/Parity error detected

2.8.5.2 Worker CSRs

Table 2.14: Worker Control and Status registers

Index	Register name	Reset value	Description
0x00	*$PC*	0x0004c000	Context Program Counter.
0x01	*$WSR*	0x00000000	Worker context status register.
0x02	*$VERTEX_BASE*	0x00000000	Vertex data structure pointer.
0x03	*$WORKER_BASE*	0x00000000	Worker context scratch space base address.
0x04	*$REPEAT_COUNT*	0x00000000	*rpt* loop down-counter.
0x05	*$REPEAT_FIRST*	0x00000000	*rpt* loop start address.
0x06	*$REPEAT_END*	0x00000000	*rpt* loop end address.
0x60	*$COUNT_L*	0x00000000	*Tile* cycle counter value. Lower 32-bits .
0x61	*$COUNT_U*	0x00000000	*Tile* cycle counter value. Upper 32-bits .
0x70	*$DBG_DATA*	0x00000000	Alias for **$DBG_DATA** debug register.

Continued on next page

Table 2.14 – continued from previous page

Index	Register name	Reset value	Description
0x71	*$DBG_BRK_ID*	0x00000000	Id of BRK channel which caused the last BREAK *exception event*. See *Debug*.
0x100	*$FP_STS*	0x00000000	Floating-point status register.
0x101	*$FP_CLR*	0x00000000	Floating-point exception/state clear.
0x102	*$FP_CTL*	0x00000000	Floating-point control register.
0x103	*$PRNG_0_0*	0x00000000	The least significant 32-bits of $PRNG_0.
0x104	*$PRNG_0_1*	0x00000000	The most significant 32-bits of $PRNG_0.
0x105	*$PRNG_1_0*	0x00000000	The least significant 32-bits of $PRNG_1.
0x106	*$PRNG_1_1*	0x00000000	The most significant 32-bits of $PRNG_1.
0x107	*$PRNG_SEED*	0x00000000	32-bit Pseudo-random-number-generator initialisation.
0x108	*$TAS*	0x00000000	**T**he **a**xpy **s**cale (or **T**emporary **a**mp **s**torage).
0x109	*$FP_NFMT*	0x00000000	Floating-point number format control register.
0x10A	*$FP_SCL*	0x00000000	Floating-point instruction scale factors

2.8.5.2.1 $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.

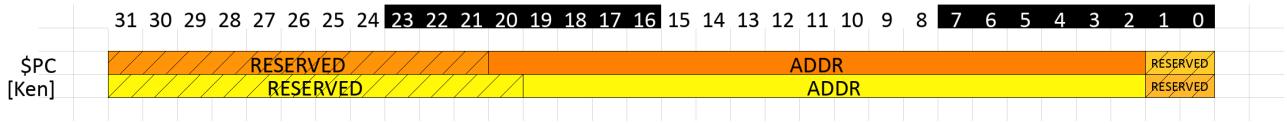


Fig. 2.8: **$PC** register format

Table 2.15: **$PC** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:2]	0x13000	*RvO*	Program counter address.

$PC register field sizes are influenced by the following parameter: TMEM_WORD_ADDRESS_WIDTH

2.8.5.2.2 $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.

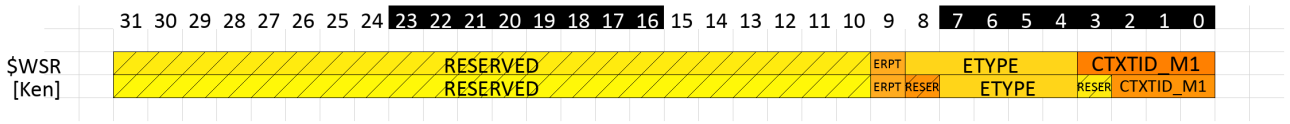


Fig. 2.9: **$WSR** register format

Table 2.16: **$WSR** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
CTXTID_M1	[2:0]	0x0	*RO*	Unique id for this hardware context, minus 1. Reset value specified is for the first worker context only.
ETYPE	[7:4]	0x0	*RvO*	Exception type (see *TileException*).
ERPT	[9]	0x0	*RvO*	Exception was raised within body of *rpt*. When set, the resulting *exception* is always *malign*.

$WSR register field sizes are influenced by the following parameters: CTXT_TOTAL_BITWIDTH, TEXCPT_ENUM_BITWIDTH

2.8.5.2.3 $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.

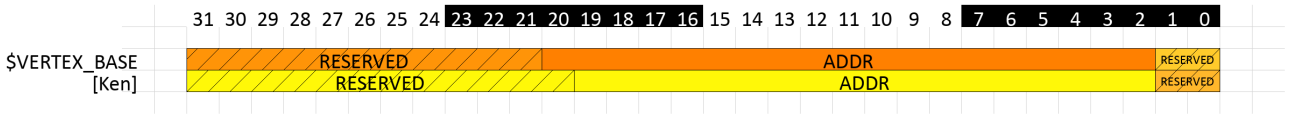


Fig. 2.10: **$VERTEX_BASE** register format

Table 2.17: **$VERTEX_BASE** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:2]	0x0	*RO*	32-bit aligned address

$VERTEX_BASE register field sizes are influenced by the following parameter: TMEM_WORD_ADDRESS_WIDTH

2.8.5.2.4 $WORKER_BASE

Register index: 3

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

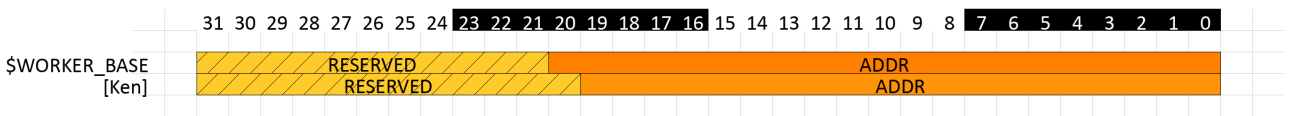


Fig. 2.11: **$WORKER_BASE** register format

Table 2.18: **$WORKER_BASE** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:0]	0x0	*RO*	Byte address

$WORKER_BASE register field sizes are influenced by the following parameter: TMEM_BYTE_ADDRESS_WIDTH

2.8.5.2.5 $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.

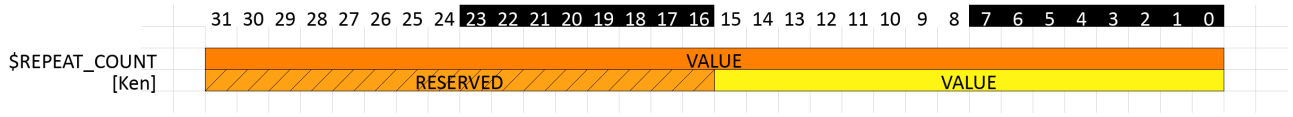


Fig. 2.12: **$REPEAT_COUNT** register format

Table 2.19: **$REPEAT_COUNT** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[15:0]	0x0	*RvO*	Value

$REPEAT_COUNT register field sizes are influenced by the following parameter: TREG_REPEAT_COUNT_WIDTH

2.8.5.2.6 $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

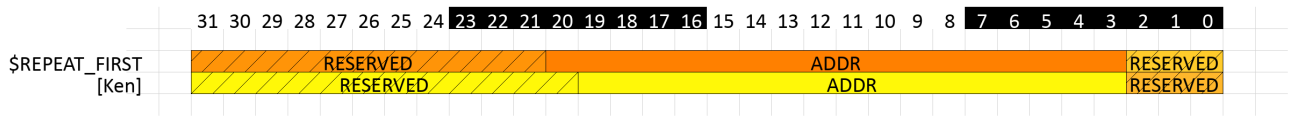


Fig. 2.13: **$REPEAT_FIRST** register format

Table 2.20: **$REPEAT_FIRST** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:3]	0x0	*RvO*	64-bit aligned address

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

2.8.5.2.7 $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

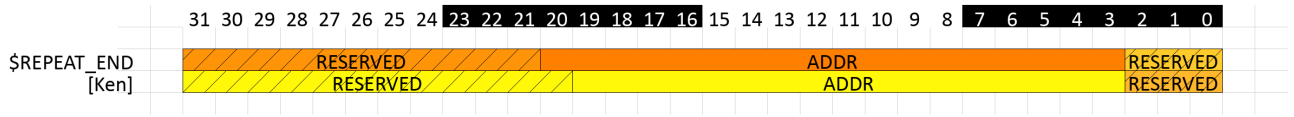


Fig. 2.14: **$REPEAT_END** register format

Table 2.21: **$REPEAT_END** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
ADDR	[19:3]	0x0	*RvO*	64-bit aligned address

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

2.8.5.2.8 $COUNT_L

Register index: 96

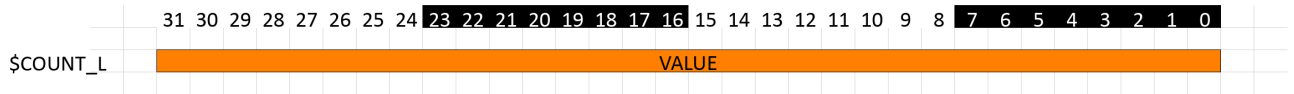


Fig. 2.15: **$COUNT_L** register format

Table 2.22: **$COUNT_L** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvO*	Value

2.8.5.2.9 $COUNT_U

Register index: 97

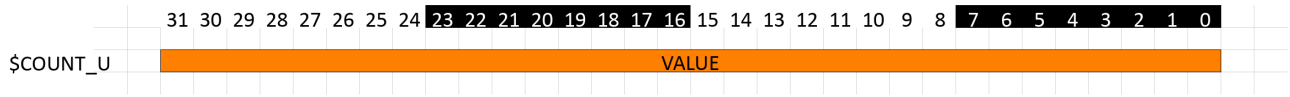


Fig. 2.16: **$COUNT_U** register format

Table 2.23: **$COUNT_U** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvO*	Value

2.8.5.2.10 $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.

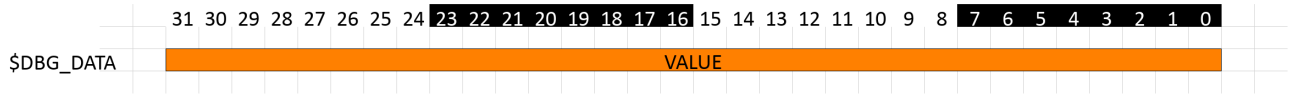


Fig. 2.17: **$DBG_DATA** register format

Table 2.24: **$DBG_DATA** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvW*	Value

2.8.5.2.11 $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.

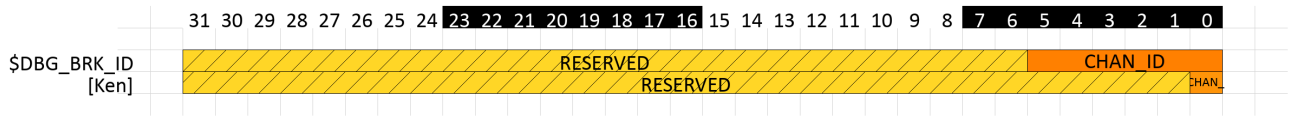


Fig. 2.18: **$DBG_BRK_ID** register format

Table 2.25: **$DBG_BRK_ID** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
CHAN_ID	[0]	0x0	*RvO*	Channel ID.

$DBG_BRK_ID register field sizes are influenced by the following parameter: TDBG_TOTAL_CHANNELS_CEIL_LOG2

2.8.5.2.12 $FP_STS

Register index: 256

Floating-point status register.

Note: This register is explicitly reset on worker launch

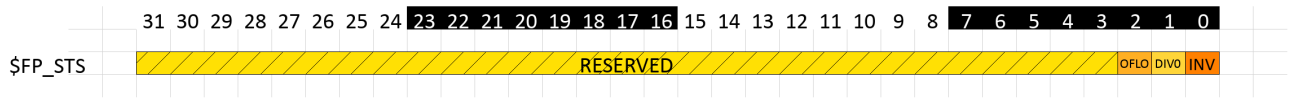


Fig. 2.19: **$FP_STS** register format

Table 2.26: **$FP_STS** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
INV	[0]	0x0	*RvO*	Floating-point invalid operation exception flag
DIV0	[1]	0x0	*RvO*	Floating-point divide-by-zero exception flag
OFLO	[2]	0x0	*RvO*	Floating-point overflow exception flag

2.8.5.2.13 $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.

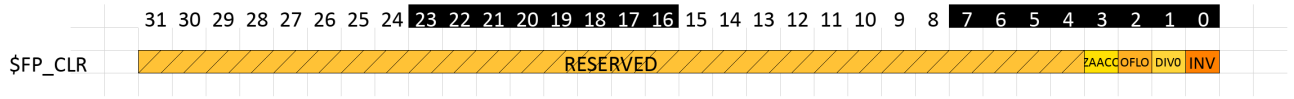


Fig. 2.20: **$FP_CLR** register format

Table 2.27: **$FP_CLR** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
INV	[0]	0x0	*W1C*	Floating-point invalid operation exception flag
DIV0	[1]	0x0	*W1C*	Floating-point divide-by-zero exception flag
OFLO	[2]	0x0	*W1C*	Floating-point overflow exception flag
ZAACC	[3]	0x0	*W1C*	Force all accumulators **$AACC[][]** to zero.

2.8.5.2.14 $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‘.

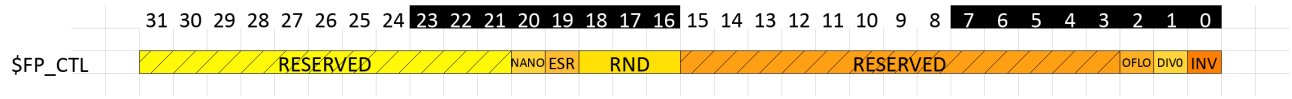


Fig. 2.21: **$FP_CTL** register format

Table 2.28: **$FP_CTL** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
INV	[0]	0x0	*RvW*	• 0b0: Floating-point invalid operation exceptions treated as benign. • 0b1: Floating-point invalid operation exceptions treated as malign.
DIV0	[1]	0x0	*RvW*	• 0b0: Floating-point divide-by-zero exceptions treated as benign. • 0b1: Floating-point divide-by-zero exceptions treated as malign.
OFLO	[2]	0x0	*RvW*	• 0b0: Floating-point overflow exceptions treated as benign. • 0b1: Floating-point overflow exceptions treated as malign.
RND	[18:16]	0x0	*RvO*	Floating-point rounding behaviour. See *TileRoundMode*. **Note:** Not all rounding modes are supported by all implementations (see *Parameters*). Attempts to write values corresponding to RESERVED or unimplemented modes will be ignored. Reads will yield non-RESERVED, implemented modes.
ESR	[19]	0x0	*RvW*	Enable stochastic rounding for those instructions that support it.
NANOO	[20]	0x0	*RvW*	Enable NAN On Overflow mode. When enabled *half-precision* calculations that have overflowed will: 1. produce a quiet NaN result, rather than saturating to the f16 maximum/minimum values. 2. Set the invalid operation flag

2.8.5.2.15 $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.

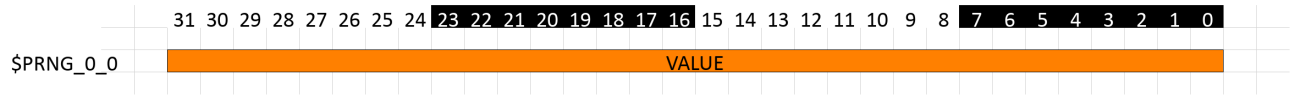


Fig. 2.22: **$PRNG_0_0** register format

Table 2.29: **$PRNG_0_0** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvW*	Value

2.8.5.2.16 $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.

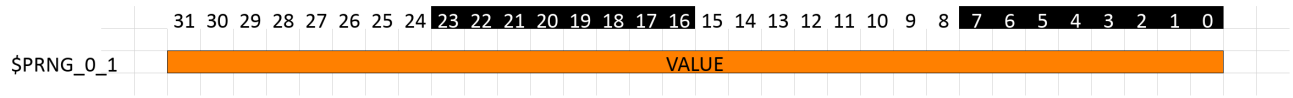


Fig. 2.23: **$PRNG_0_1** register format

Table 2.30: **$PRNG_0_1** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvW*	Value

2.8.5.2.17 $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.

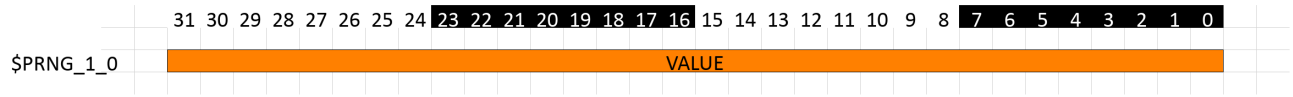


Fig. 2.24: **$PRNG_1_0** register format

Table 2.31: **$PRNG_1_0** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvW*	Value

2.8.5.2.18 $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.

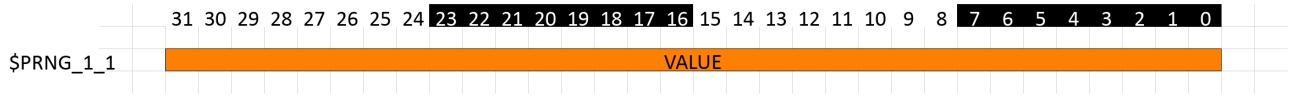


Fig. 2.25: **$PRNG_1_1** register format

Table 2.32: **$PRNG_1_1** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*RvW*	Value

2.8.5.2.19 $PRNG_SEED

Register index: 263

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

- *$PRNG_0_0* = *value*
- *$PRNG_0_1* = ~*value*
- *$PRNG_1_0* = (*value* << 13) | (~*value* >> 19)
- *$PRNG_1_1* = (~*value* << 13) | (*value* >> 19)

Reads return 0.

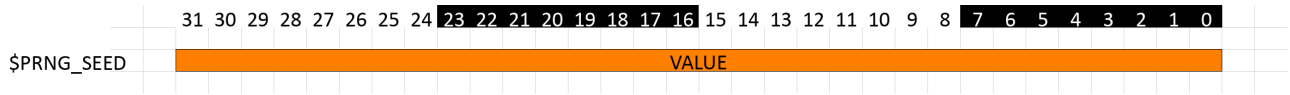


Fig. 2.26: **$PRNG_SEED** register format

Table 2.33: **$PRNG_SEED** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
VALUE	[31:0]	0x0	*WC*	Value

2.8.5.2.20 $TAS

Register index: 264

The **a**xpy **s**cale(s) (or **T**emporary **a**mp **s**torage). The scalar value(s) used by the various *axp(b)y instructions (*f32v2axpy*, *f16v4mix* for example), or temporary storage for certain AMP instructions (*f32sisoamp*, *f32sisoslic* for example)

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

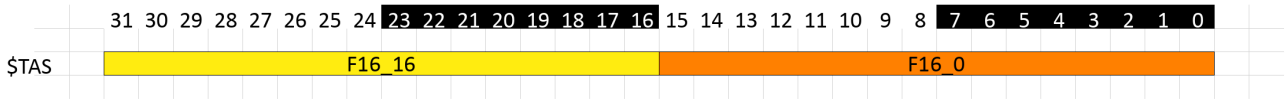


Fig. 2.27: **$TAS** register format

Table 2.34: **$TAS** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
F16_0	[15:0]	0x0	*RvW*	*Half-precision* floating-point value (or bottom 16-bits of a *single-precision* value).
F16_16	[31:16]	0x0	*RvW*	*Half-precision* floating-point value (or top 16-bits of a *single-precision* value).

2.8.5.2.21 $FP_NFMT

Register index: 265

Floating-point number format control register.

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

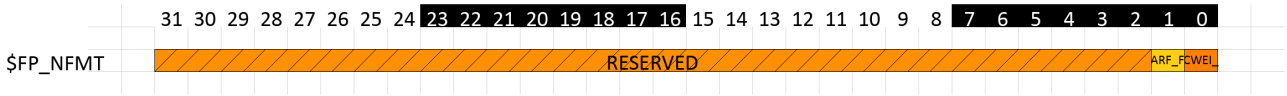


Fig. 2.28: **$FP_NFMT** register format

Table 2.35: **$FP_NFMT** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
CWEI_FMT	[0]	0x0	*RvO*	Define the format of the CWEI inputs to the AMP unit for f8 operations • 0b0: f8 CWEI operands are treated as *Quart152* values • 0b1: f8 CWEI operands are treated as *Quart143* values

Continued on next page

Table 2.35 – continued from previous page

Field name	Bit field	Reset value	*Access semantics*	Description
ARF_FMT	[1]	0x0	*RvW*	Define the format of the ARF based *quarter-precision* inputs to certain format conversion and AMP based operations • 0b0: f8 ARF operands are treated as *Quart152* values • 0b1: f8 ARF operands are treated as *Quart143* values

2.8.5.2.22 $FP_SCL

Register index: 266

Floating-point instruction scale factors

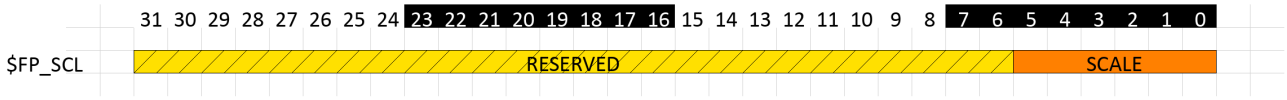


Fig. 2.29: **$FP_SCL** register format

Table 2.36: **$FP_SCL** register fields

Field name	Bit field	Reset value	*Access semantics*	Description
SCALE	[5:0]	0x0	*RW*	Signed scale exponent for certain instructions with *half-precision* or *quarter-precision* input or output vectors.

2.8.6 Pipeline Internal State

2.8.6.1 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 2.37: Aux pipeline internal state

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

Note: *N* is TFPU_NUM_ACCUM

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

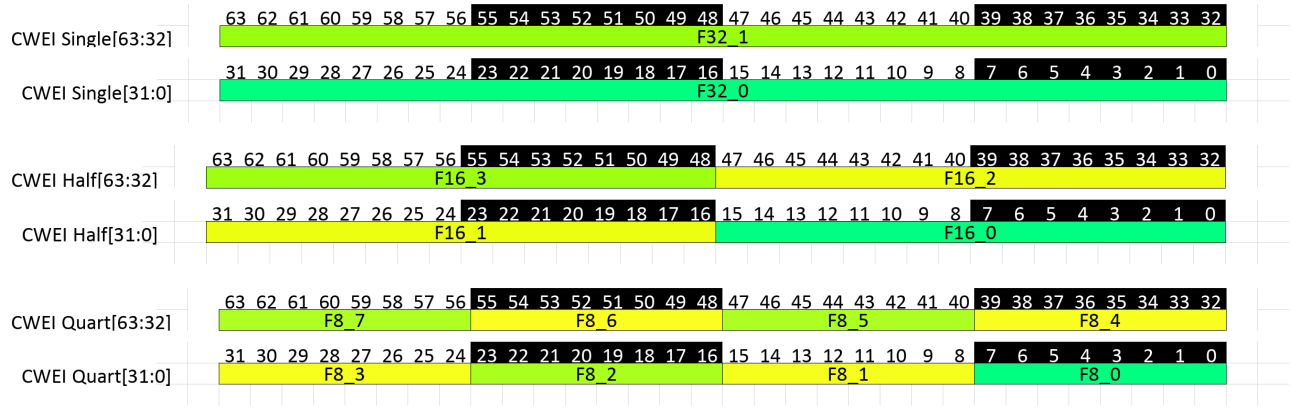
f16v2cmac, *f16v2gina*, *f16v4absacc*, *f16v4acc*, *f16v4cmac*, *f16v4gacc*, *f16v4hihoamp*, *f16v4hihoslic*, *f16v4hihov4amp*, *f16v4hihov4slic*, *f16v4istacc*, *f16v4mix*, *f16v4sisoamp*, *f16v4sisoslic*, *f16v4stacc*, *f16v8absacc*, *f16v8acc*, *f16v8sqacc*, *f32mac*, *f32sisoamp*, *f32sisoslic*, *f32sisov2amp*, *f32sisov2slic*, *f32v2aop*, *f32v2axpy*, *f32v2gina*, *f32v2mac*, *f32v4absacc*, *f32v4acc*, *f32v4sqacc*, *f8v8hihov4amp*, *f8v8hihov4slic*

2.8.7 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 the *ld64putcs* and *ld128putcs* instructions. The data format of the register contents is determined by the instruction using the CCCS state.

The register values can be *single-precision*, *half-precision* or *quarter-precision*:



The configuration space provides access to the following shared configuration state elements:

2.8.7.1 Common Compute Configuration Space

Table 2.38: Collaborative Compute Configuation Space

Index	Register name	Reset value	Description
0x00	*$CWEI_0_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x01	*$CWEI_0_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x02	*$CWEI_0_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x03	*$CWEI_0_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x04	*$CWEI_1_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x05	*$CWEI_1_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x06	*$CWEI_1_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x07	*$CWEI_1_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x08	*$CWEI_2_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x09	*$CWEI_2_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x0A	*$CWEI_2_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x0B	*$CWEI_2_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.

Continued on next page

Table 2.38 – continued from previous page

Index	Register name	Reset value	Description
0x0C	*$CWEI_3_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x0D	*$CWEI_3_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x0E	*$CWEI_3_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x0F	*$CWEI_3_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x10	*$CWEI_4_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x11	*$CWEI_4_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x12	*$CWEI_4_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x13	*$CWEI_4_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x14	*$CWEI_5_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x15	*$CWEI_5_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x16	*$CWEI_5_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x17	*$CWEI_5_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x18	*$CWEI_6_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x19	*$CWEI_6_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1A	*$CWEI_6_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1B	*$CWEI_6_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1C	*$CWEI_7_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1D	*$CWEI_7_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1E	*$CWEI_7_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x1F	*$CWEI_7_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x20	*$CWEI_8_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x21	*$CWEI_8_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x22	*$CWEI_8_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x23	*$CWEI_8_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x24	*$CWEI_9_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x25	*$CWEI_9_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x26	*$CWEI_9_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x27	*$CWEI_9_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x28	*$CWEI_10_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x29	*$CWEI_10_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2A	*$CWEI_10_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2B	*$CWEI_10_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2C	*$CWEI_11_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2D	*$CWEI_11_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2E	*$CWEI_11_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x2F	*$CWEI_11_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x30	*$CWEI_12_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x31	*$CWEI_12_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x32	*$CWEI_12_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x33	*$CWEI_12_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.

Continued on next page

Table 2.38 – continued from previous page

Index	Register name	Reset value	Description
0x34	*$CWEI_13_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x35	*$CWEI_13_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x36	*$CWEI_13_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x37	*$CWEI_13_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x38	*$CWEI_14_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x39	*$CWEI_14_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3A	*$CWEI_14_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3B	*$CWEI_14_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3C	*$CWEI_15_0*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3D	*$CWEI_15_1*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3E	*$CWEI_15_2*	0x0	Common f8v8/f16v4/f32v2 weight vector.
0x3F	*$CWEI_15_3*	0x0	Common f8v8/f16v4/f32v2 weight vector.

2.8.7.1.1 $CWEI_*n*_0

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

Common f8v8/f16v4/f32v2 weight vector. Reset to zero by the *Supervisor*. Initialised from *Tile Memory* by the *ld64putcs* and *ld128putcs* instructions.

Format of register contents can be *single-precision*, *half-precision* or *quarter-precision*.

2.8.7.1.2 $CWEI_*n*_1

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

Common f8v8/f16v4/f32v2 weight vector. Reset to zero by the *Supervisor*. Initialised from *Tile Memory* by the *ld64putcs* and *ld128putcs* instructions.

Format of register contents can be *single-precision*, *half-precision* or *quarter-precision*.

2.8.7.1.3 $CWEI_*n*_2

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

Common f8v8/f16v4/f32v2 weight vector. Reset to zero by the *Supervisor*. Initialised from *Tile Memory* by the *ld64putcs* and *ld128putcs* instructions.

Format of register contents can be *single-precision*, *half-precision* or *quarter-precision*.

2.8.7.1.4 $CWEI_*n*_3

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

Common f8v8/f16v4/f32v2 weight vector. Reset to zero by the *Supervisor*. Initialised from *Tile Memory* by the *ld64putcs* and *ld128putcs* instructions.

Format of register contents can be *single-precision*, *half-precision* or *quarter-precision*.

2.9 Memory Model

2.9.1 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 (TMEM_BASE_ADDR + (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*.

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

2.9.2 Memory Element

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

2.9.3 Memory Regions

Tile Memory may be divided into an *implementation dependent* number of *Memory Regions*. Each memory region:

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

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 TMEM_REGION0_BASE_ADDR < 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 [TMEM_REGION0_BASE_ADDR, 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*).

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. 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.

[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.

When *n*-way interleaved, *n* *memory element*s are grouped (address wise) in such a way that *n* consecutive 64-bit logical memory locations span the *n* distinct *memory element*s. 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*.

2.9.4 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.

2.9.4.1 Pointers

2.9.4.1.1 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.

2.9.4.1.2 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).

Fig. 2.30: Packed address format

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

2.9.4.2 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 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 width, 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.

2.9.4.3 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*.

2.9.5 Endianness

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

2.9.6 Memory Map

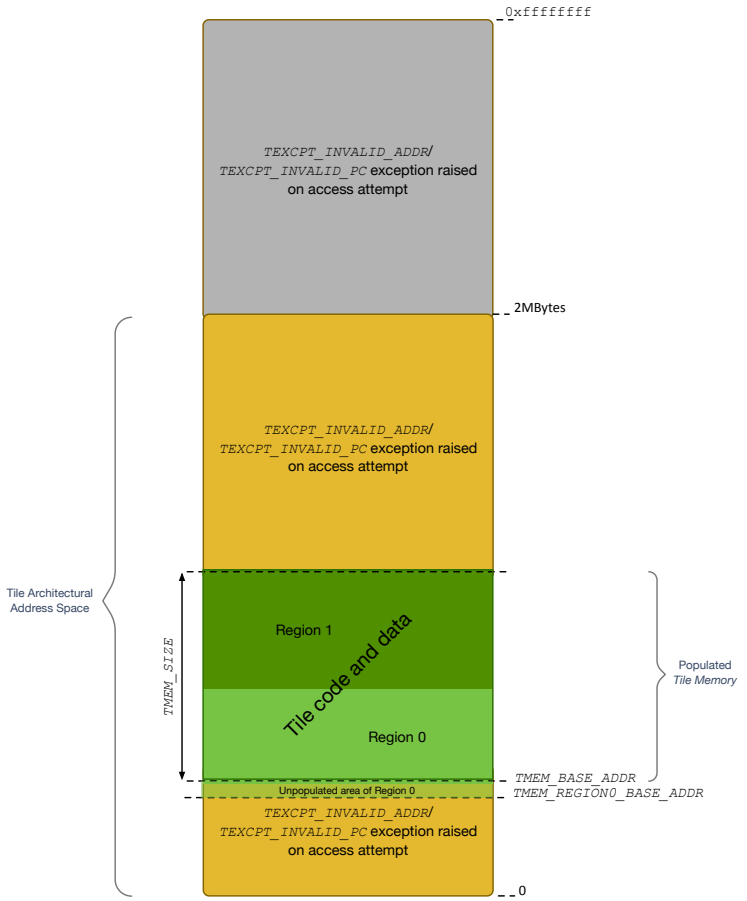


Fig. 2.31: *Tile* local memory map

See *Memory parameters* for variable definitions.

2.9.7 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.

2.9.8 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.

2.9.9 Memory Protection

Tile provides no hardware mechanisms for memory protection.

2.9.10 Memory Access Semantics

2.9.10.1 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 a *PC* that is not 4-byte aligned. 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.

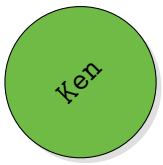
The following *pseudo-code* serves to illustrate the instruction fetch semantics:

Listing 2.5: Instruction Fetch Semantics

```
(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);
  }
}
```

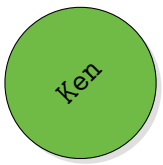
Function references: *TMem_IsValidAddress*, *TMem_AddressIsExecutable*, *TMem_ElementId*, *TMem_ElementOffset*

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*).

Implementation detail



A memory element clash between (asynchronous) *receive* data provided by the Exchange Interface and the first **instruction fetch** performed by a context will result in the fetch phase raising a TEXCPT_CONFLICT *exception event*.

See *Exchange based memory clashes* for futher details.

2.9.10.2 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 zero/sign 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 element*s. 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 Bundle*s 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

```
void DataLoad(uint32_t eAddr,         /* An access-size aligned effective address */
              int accessSizeInBytes,  /* Number of bytes to load: 1, 2, 4, 8 or 16 */
              bool signNotZeroExtend, /* True if data is to be sign extended, false otherwise */
              int zeroTailShift       /* Number of zero bits to add to data tail */
                  uint64_t *result) { /* Loaded data comprising 1 or 2 64-bit values */
  if ((eaddr & ((accessSizeInBytes)-1)) != 0) {
    // Memory accesses must be naturally aligned
    EXCEPT(TEXCPT_INVALID_ADDR);
  }

  // All memory accesses are 8-byte aligned
```

```
  unsigned elemId0 = TMem_ElementId(eAddr);
  unsigned eOffset = TMem_ElementOffset(eAddr);
  TMemElement &element0 = TileMemory[elemId0];

  if (accessSizeInBytes == 16) {
    // 128-bit access
    unsigned elemId1 = TMem_ElementId(eAddr + 8);
    TMemElement &element1 = TileMemory[elemId1];
    result[0] = element0[eOffset];
    result[1] = element1[eOffset];

  } else if (accessSizeInBytes == 8) {
    // 64-bit access
    result[0] = element0[eOffset];

  } else {
    // 32, 16 or 8-bit access - get 64 bits from the memory element
    uint64_t octaByte = element0[eOffset];

    // Pick the relevant result portion - leaving the MSBs as 0
    result[0] = (octaByte >> (8 * (eAddr & 0x7))) & ((1 << (accessSizeInBytes * 8)) - 1);

    // Sign extend if necessary
    if (signNotZeroExtend) {
      result[0] = Tile_SignExtend64((result[0] << zeroTailShift),
                                    (accessSizeInBytes * 8) + zeroTailShift);
    }
  }
}
```

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

2.9.11 Memory System Errors

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

2.9.11.1 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 event*s, the store will be committed to memory.

2.9.12 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 element*s. 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.

2.9.12.1 Instruction Stream Based

Any memory instruction which accesses multiple address locations must ensure that those effective addresses map to distinct *memory element*s, 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.

2.9.12.2 Exchange Based

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

2.10 Floating-Point Unit

2.10.1 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.

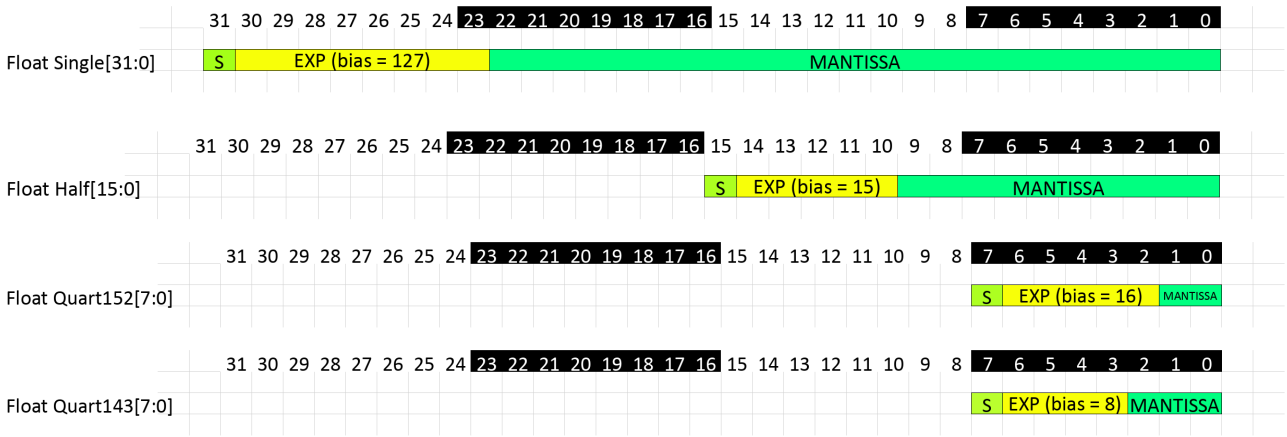
2.10.2 Number Formats

2.10.2.1 Scalars

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

Table 2.39: 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
Quarter-precision	n/a	8-bits	7	[6:2]	[1:0]	16
Quarter-precision	n/a	8-bits	7	[6:3]	[2:0]	8



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*.

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

2.10.2.1.1 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_
```

2.10.2.1.2 Half Precision Type

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/Flt16Iface.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_IS_GCNAN(v) ((v) == HALF_GCNAN)
#define HALF_INFINITY (HALF_MAX_EXP << HALF_EXP_SHIFT)
#define HALF_NEG_INFINITY ((1 << HALF_SIGN_SHIFT) | HALF_INFINITY)
#define HALF_MAX (((HALF_EXP_MASK - 1) << 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 Flt16Iface {

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);

  /**
   * Copy constructor
   */
  Half(const Half &other) = default;

  /**
   * 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 bitz32(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 &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 GCNaN
 */
bool isGCNaN() 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
 */
float maxMagnitude() const;

/**
 * What's the maximum representable finite number
 */
```

```
  static float max();

  /**
   * Returns the next representable value of x in the direction of y.
   */
  static Half nextafter(Half x, Half y);

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

} // namespace ciss

#endif // _ciss_Half_h_
```

2.10.2.1.3 Quarter Precision Type

Listing 2.9: Quarter Precision Class

```
/*************************************************************************
 * Quart.h: Quarter-precision floating-point abstraction for Colossus
 *          Instruction Set Simulator
 *
 * Copyright (c) 2020-2022 Graphcore Ltd. All rights reserved.
 *
 * Notes:
 *   Most operations are performed by:
 *     1 Converting the quarter-precision value to single-precision (float)
 *     2 Performing the operation at single-precision accuracy
 *     3 Converting the result back to quarter-precision
 *
 ************************************************************************/

#ifndef _ciss_Quart_h_
#define _ciss_Quart_h_

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

#include "colossus/arch_versions.h"

#if ARCH_VERSION_KEN_PLUS

#include <stdint.h>

#include "ciss/defines.h"
#include "colossus/tileimplconsts.h"

// The same for all quarter types
#define QUART_SIGN_SHIFT (7)

/* Fixed graphcore f8 qNaN bitcode: -0.0 */
#define QUART_ERROR ((uint8_t)(1 << QUART_SIGN_SHIFT))

namespace ciss {

class Quart {

public:
  /**
   * Only allowed to make it possible to autogenerate classes with Quart members
   * them like the PipelineState.
   */
  Quart();
```

```
/**
 * Initialise from a single-precision fp value
 */
Quart(qfmt_t qfmt, qbias_adj_t qbiasAdj, float value);

/**
 * Initialise as zero
 */
Quart(qfmt_t qfmt, qbias_adj_t qbiasAdj);

/**
 * Initialise from a raw 8-bit pattern
 */
explicit Quart(qfmt_t qfmt, qbias_adj_t qbiasAdj, uint8_t bitPattern);

/**
 * Initialise from a single-precision fp value
 */
Quart(qfmt_t qfmt, float value)
    : Quart(qfmt, 0, value) {}

/**
 * Initialise as zero
 */
Quart(qfmt_t qfmt)
    : Quart(qfmt, 0) {}

/**
 * Initialise from a raw 8-bit pattern
 */
explicit Quart(qfmt_t qfmt, uint8_t bitPattern)
    : Quart(qfmt, 0, bitPattern) {}

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

/**
 * Obtain quarter-precision bit-pattern
 * @returns raw    8-bit saturated bit-pattern.
 *
 */
uint8_t bit8(HalfSaturationMode_t smode) const;

/**
 * Obtain quarter-precision bit-pattern
 * @returns raw    zero-extended saturated 8-bit bit-pattern.
 *
 */
uint32_t bitz32(HalfSaturationMode_t smode) const {
  return (uint32_t)bit8(smode);
}

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

Quart &log(const float &other);

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

/**
 * Check if value represents an error
 */
bool isError() 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 quarter-precision?
 */
static float max(qfmt_t qfmt, qbias_adj_t qbiasAdj = 0, bool sign = false);

/**
 * Return the default bias for a given FP8 format
 * @param qfmt Quarter-precision format
 * @return Default bias for qfmt
 */
static inline qbias_adj_t defaultBias(qfmt_t qfmt) {
  return (qfmt == quart_one_five_two) ? TFPU_FP8_152_BIAS : TFPU_FP8_143_BIAS;
}

/**
 * Return the bias for the current FP8 format
 */
inline qbias_adj_t bias() const {
  return qbias;
}

/**
 * Return the maximum representable value for the current FP8 format
 */
inline float maxValue() const {
  return Quart::max(qfmt, qbias - defaultBias(qfmt), signBit());
}

/**
 * Return the mantissa size for the current FP8 format
 */
inline uint32_t mantSize() const {
  return (qfmt == quart_one_five_two) ? 2 : 3;
}

/**
 * Return the exponent size for the current FP8 format
 */
inline uint32_t expSize() const {
  return (qfmt == quart_one_five_two) ? 5 : 4;
}

/**
 * Return the bit-index of the sign bit
 */
inline uint32_t signShift() const {
  return 7;
}

/**
 * Return the bit-index of the exponent bits
 */
inline uint32_t expShift() const {
  return mantSize();
}

/**
 * Return the bit-index of the mantissa bits
 */
inline uint32_t mantShift() const {
  return 0;
}
```

```
  /**
   * Return the bit-mask of the exponent bits
   */
  inline uint32_t expMask() const {
    return (1 << expSize()) - 1;
  }

  /**
   * Return the bit-mask of the mantissa bits
   */
  inline uint32_t mantMask() const {
    return (1 << mantSize()) - 1;
  }

  /**
   * Return the sign bit
   */
  inline uint32_t signBit() const {
    return (iquart >> signShift()) & 1;
  }

  /**
   * Return the exponent bits
   */
  inline uint32_t expBits() const {
    return (iquart >> expShift()) & expMask();
  }

  /**
   * Return the mantissa bits
   */
  inline uint32_t mantBits() const {
    return (iquart >> mantShift()) & mantMask();
  }

private:
  qfmt_t qfmt;
  qbias_adj_t qbias;
  uint8_t iquart;
  float single;
  bool overflowed;
  bool hadSignBit;

  uint8_t toQuart(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 Quart

/**
 * Extract a quarter-precision value from a 32-bit input value based on given index.
 * @param value  Single precision input value.
 * @param quarter Quarter index for the extracted bits.
 * @param qfmt Quarter-precision format for the resulting value.
 * @returns Quarter-precision value extracted from input value.
 *
 */
Quart pickQuart(uint32_t value, int quarter, qfmt_t qfmt);

} // namespace ciss

#endif

#endif // _ciss_Quart_h_
```

2.10.2.2 Vectors

Tile supports the following vector floating-point formats:

Table 2.40: Floating-point vector formats

Base type	Vector size	Registers/vector	Comments	Example
Single-precision	2 elements	2		*f32v2add*
Single-precision	4 elements	4	Supported as a source operand type for certain instructions involving the *accumulator* state only	*f32v4acc*
Half-precision	2 elements	1	16-bit *Half-precision* values are packed into the upper and lower halves of each register	*f16v2add*
Half-precision	4 elements	2	16-bit *Half-precision* values are packed into the upper and lower halves of each register	*f16v4add*
Half-precision	8 elements	4	Supported as a source operand type for certain instructions involving the *accumulator* state only	*f16v8acc*
Quarter-precision	2 elements	1	8-bit *Quarter-precision* values are packed into the each byte of each register	*f8v2tof16*
Quarter-precision	4 elements	1	8-bit *Quarter-precision* values are packed into the each byte of each register	*f8v4class*
Quarter-precision	8 elements	2	8-bit *Quarter-precision* values are packed into the each byte of each register	*f8v8hihov4amp*

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

2.10.3 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 2.41: Floating-point control and status registers

Register	Description
$FP_CTL	Floating-point control register. Used to (un)mask floating-point exceptions, and set the rounding mode (including stochastic rounding).
$FP_STS	Floating-point status register. Provides floating-point exception status.
$FP_CLR	Floating-point *clear* register. Used to clear floating-point exception flags and to zero the accumulator state.
$FP_NFMT	Floating-point number format register. Used to configure the number format of *quarter-precision* ARF and CWEI values.
$FP_SCL	Floating-point operation scaling register. Used by some conversion instructions to scale output values and some AMP/SLIC instructions to scale intermediate product results.

Note: The *Worker* context floating-point control registers *$FP_CTL*, *$FP_NFMT* and *$FP_SCL* are initialised from the *Supervisor* control registers *$FP_ICTL*, *$FP_INFMT* and *$FP_ISCL* respectively during execution of the *run* and *runall* instructions. *run* and *runall* also clear all floating point exception flags in the target *Worker*.

2.10.4 General Accuracy

2.10.4.1 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*.

2.10.4.2 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 $\pm$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).

2.10.4.3 Quarter-precision

Operations on *quarter-precision* values are based on the guiding principals of *[IEEE754]*, with some significant differences:

- Two different 8-bit formats are supported, offering a trade-off between dynamic range and precision.
- *AMP and SLIC* based multiplications on *quarter-precision* values allow the 2 sets of multiplicands to be of different 8-bit formats.
- For both formats, there is a single encoding used to represent all error and infinity cases (**ERROR** == **0x80**).
- As such:
  - **-0** can not be represented in *quarter-precision* and **0** will be returned in cases where **-0** would otherwise be expected.
  - *Tile* cannot produce $\pm\infty$ in either *quarter-precision* format
  - *Tile* cannot produce a NaN code (quiet or signaling) in either *quarter-precision* format
  - *Tile* treats **ERROR** 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 the **ERROR** code, or a quiet NaN if the result format supports quiet NaNs
    - The invalid operation flag, $FP_STS.INV will be set to 0b1.
- In default mode ($FP_CTL.NANOO == 0b0), an operation that has overflowed the *quarter-precision* range will saturate to $\pm$MAX instead (and set the overflow flag $FP_STS.OFLO). **Note**: The dynamic range of the *quarter-precision* depends on the format configured by *$FP_NFMT*
- In NAN On Overflow mode ($FP_CTL.NANOO == 0b1), an operation that has overflowed the *quarter-precision* range will produce an **ERROR** and set the invalid operation flag $FP_STS.INV.

Conversions from a higher precision infinity to *quarter-precision* (*f16v8tof8* for example) will produce the **ERROR** result and set the invalid operation flag.

2.10.4.3.1 Quart formats

The format (and therefore dynamic range and precision) of *quarter-precision* numbers is configured by the *$FP_NFMT* register. 2 x 8-bit formats are supported:

- *Quart152*
- *Quart143*
- When $FP_NFMT.CWEI_FMT is 0b0 the CWEI based *quarter-precision* operands are interpreted as *Quart152* values.
- When $FP_NFMT.CWEI_FMT is 0b1 the CWEI based *quarter-precision* operands are interpreted as *Quart143* values.
- When $FP_NFMT.ARF_FMT is 0b0 the ARF based *quarter-precision* operands are interpreted as *Quart152* values.
- When $FP_NFMT.ARF_FMT is 0b1 the ARF based *quarter-precision* operands are interpreted as *Quart143* values.

2.10.4.3.2 Quart scaling

Tile supports configurable scaling of the final or intermediate results of arithmetic operations on *quarter-precision* input values. The scaling factor is a power-of-two value, configured via the signed exponenet value contained within the $FP_SCL.SCALE CSR field.

- For *quarter-precision AMP and SLIC* instructions, the scaling is applied (effectively a multiplication by a power-of-two, or equivalently the adjustment of the result exponent value) to the result of each multiplication within a dot-product, prior to the subsequent addition and accumulation.
- For conversions to and from a *quarter-precision* value, the scaling is applied to the output value prior to normalisation and formatting to the result format.

2.10.5 Rounding Modes

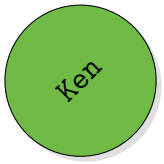
The *Tile* architecture defines the following rounding modes for floating-point operations. Not all rounding modes are supported by all implementations (see *Implementation Specifics*).

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

Table 2.42: 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 $FP_CTL.ESR. When enabled, stochastic rounding is performed (by those instructions that support it) in preference to the rounding mode specified by $FP_CTL.RND. See *Stochastic Rounding* for further details.

Implementation detail



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

2.10.6 Format Conversion and Transformations

Tile supports the following number format conversion and transformation operations:

Table 2.43: Format conversions

Source format	Result format	Instruction	Description
Single-precision	*Single-precision*	*f32int*	Round a *Single-precision* value to a single-precision integral. Rounding as per the mode specified by an instruction immediate ($FP_CTL.RND has no effect on this instruction). Zero and ∞ operands are converted to zero/∞ results of the same sign.
Single-precision	Signed 32-bit integer	*f32toi32*	Convert a *Single-precision* value to a signed 32-bit integer. Rounding as per the current mode specified by $FP_CTL.RND. NaN and ∞ operands will result in the setting of $FP_STS.INV. $FP_STS.INV will also be set if the source value is beyond the range of a signed 32-bit integer.
Single-precision	Unsigned 32-bit integer	*f32toui32*	Convert a *Single-precision* value to an unsigned 32-bit integer. Rounding as per the current mode specified by $FP_CTL.RND. NaN and ∞ operands will result in the setting of $FP_STS.INV. $FP_STS.INV will also be set if the source value is beyond the range of an unsigned 32-bit integer.
Signed 32-bit integer	*Single-precision*	*f32fromi32*	Convert a signed 32-bit integer to *Single-precision* floating-point. Values in the range [-2^{24}, 2^{24}] are converted exactly. For other values, rounding is applied as per $FP_CTL.RND.
Unsigned 32-bit integer	*Single-precision*	*f32fromui32*	Convert an unsigned 32-bit integer to *Single-precision* floating-point. Values in the range [0, 2^{24}] are converted exactly. For other values, rounding is applied as per $FP_CTL.RND.

Continued on next page

Table 2.43 – continued from previous page

Source format	Result format	Instruction	Description
Single-precision	*Half-precision*	*f32tof16*	Convert a *Single-precision* value to *Half-precision*: • Signaling NaNs are converted to the *Tile fixed-value f16 quiet NaN* and set $FP_STS.INV to 0b1 • Quiet NaNs are converted to the *Tile* fixed-value f16 quiet NaN (but don't set $FP_STS.INV) • $\pm\infty$ are treated as per signaling NaNs • ± 0 map to ± 0 • Otherwise: – If enabled, stochastic rounding is applied to the single-precision input value (see *Stochastic Rounding*) – Otherwise, rounding is applied as per $FP_CTL.RND – 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 *Half-precision* denorms (with rounded significand) – Rounded values with magnitude in the range $[2^{-14}, 65504]$ map to normalized *Half-precision* values (with rounded significand) – Rounded values with magnitude exceeding 65504 map to: * ± 65504 (the maximum representable value) and set $FP_STS.OFLO to 0b1, if $FP_CTL.NANOO is 0, * the *Tile* fixed-value f16 quiet NaN if $FP_CTL.NANOO is 1. In this case, both $FP_STS.OFLO and $FP_STS.INV are set to 0b1
2-element *Single-precision* vector	2-element *Half-precision* vector	*f32v2tof16*	Vectorized version of *f32tof16*
4-element *Single-precision* vector	4-element *Half-precision* vector	*f32v4tof16*	Vectorized version of *f32tof16*
Half-precision	*Single-precision*	*f16tof32*	Convert a *Half-precision* value to *Single-precision*: • Signaling NaNs are converted to the *Tile* fixed-value quiet NaN (*TFPU_F32_QNan*) and set $FP_STS.INV to 0b1 • Quiet NaNs are converted to the *Tile* fixed-value quiet NaN (*TFPU_F32_QNan*) • $\pm\infty$ map to $\pm\infty$ • All other values can be represented exactly in *Single-precision*, so no rounding is performed
2-element *Half-precision* vector	2-element *Single-precision* vector	*f16v2tof32*	Vectorized version of *f16tof32*.

Continued on next page

Table 2.43 – continued from previous page

Source format	Result format	Instruction	Description
Unsigned 32-bit integer	*Single-precision*	*f32sufromui*	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 *Tile* rounding mode.
2-element vector of 32-bit unsigned integers	2-element *Single-precision* vector	*f32v2sufromui*	Vectorized version of *f32sufromui*
2-element vector of 16-bit unsigned integers	2-element *Half-precision* vector	*f16v2sufromui*	Convert a pair of unsigned 16-bit integer values to a 2-element Half-precision 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 *Tile* rounding mode.
4-element vector of 16-bit unsigned integers	4-element *Half-precision* vector	*f16v4sufromui*	A 4-element variant of *f16v2sufromui*.
2-element *Half-precision* vector	2-element *Quarter-precision* vector	*f16v2tof8*	Convert a vector of *Half-precision* values to *Quarter-precision*: • Signaling NaNs as are converted to *Quarter-precision* **ERROR** and set $FP_STS.INV to 0b1 • Quiet NaNs are converted to *Quarter-precision* **ERROR** (but don't set $FP_STS.INV) • $\pm\infty$ are treated as per signaling NaNs • ± 0 map to $+0$ • Otherwise: – The input value is *scaled* – If enabled, stochastic rounding is applied to the scaled *half-precision* input value (see *Stochastic Rounding*) – Otherwise rounding is performed as per $FP_CTL.RND – Rounded values with a magnitude smaller than the *Quarter-precision* minimum denorm map to $+0$ – Rounded values within the *Quarter-precision* denorm range map to denorms (with rounded significand) – Rounded values within the *Quarter-precision* norm range map to normalized values (with rounded significand) – Rounded values with magnitude exceeding the *Quarter-precision* range map to: * $\pm$*maximum representable value* and set $FP_STS.OFLO to 0b1, if $FP_CTL.NANOO is 0, * the *Quarter-precision* **ERROR** value if $FP_CTL.NANOO is 1. In this case, both $FP_STS.OFLO and $FP_STS.INV are set to 0b1

Continued on next page

Table 2.43 – continued from previous page

Source format	Result format	Instruction	Description
8-element *Half-precision* vector	8-element *Quarter-precision* vector	*f16v8tof8*	Wider variant of *f16v2tof8*
2-element *Quarter-precision* vector	2-element *Half-precision* vector	*f8v2tof16*	Convert a vector of *Quarter-precision* values to *Half-precision*: • *Quarter-precision* **ERROR** is converted to the *Tile fixed-value f16 quiet NaN* and set $FP_STS.INV to 0b1 • +0 is converted to +0 • Otherwise: – The input value is *scaled* – Scaled values with magnitude less than 2^{-24} map to ±0 – Scaled values with magnitude in the range [2^{-24}, 2^{-14} - 2^{-24}] map to *Half-precision* denorms (with rounded significand) – Scaled values with magnitude in the range [2^{-14}, 65504] map to normalized *Half-precision* values (with rounded significand) – Scaled values with magnitude exceeding 65504 map to: * ±65504 (the maximum representable value) and set $FP_STS.OFLO to 0b1, if $FP_CTL.NANOO is 0, * the *Tile fixed-value f16 quiet NaN* if $FP_CTL.NANOO is 1. In this case, both $FP_STS.OFLO and $FP_STS.INV are set to 0b1
4-element *Quarter-precision* vector	4-element *Half-precision* vector	*f8v4tof16*	Wider variant of *f8v2tof16*

2.10.7 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*.

2.10.7.1 Exception Conditions

2.10.7.1.1 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*.

2.10.7.1.2 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*.

2.10.7.1.3 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*.

2.10.7.1.4 Underflow

Tile does not support the *[IEEE754]* underflow exception.

2.10.7.1.5 Inexact Result

Tile does not support the *[IEEE754]* inexact exception.

2.10.8 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 2.44: Floating-point comparisons

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

2.10.9 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*

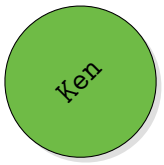
- Initialised from the *ARF* using single-precision values via *f32v2gina*
- As unformatted, arbitrary data via *f16v4istacc*

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



IPU21's floating-point accumulator format is strictly single-precision, with denorms flushed to zero.

2.10.9.1 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*

2.10.9.2 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*

2.10.10 Dot-Products

2.10.10.1 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*
- *f16v4cmac*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4mix*
- *f16v4sisoamp*
- *f16v4sisoslic*

2.10.10.2 Quarter-Precision Vector Dot-Products

Certain instructions perform a vector dot-product operation between 8-element *quarter-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 these dot-product operations, 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. The single-precision result of the dot-product is accumulated into the aux *accumulator state*.

- *f8v8hihov4amp*
- *f8v8hihov4slic*

2.10.11 AMP and SLIC Instructions

The **A**ccumulating **M**atrix **P**roduct and **Sli**m **C**onvolution instructions facilitate high performance multiply-accumulate sequences, for scenarios where weight sharing across Worker contexts is appropriate.

AMP and SLIC instructions are supported for single-precision, half-precision and quarter-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 accumulate
    - 2 x 8-element f8 *dot-product* plus accumulate

    where 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).

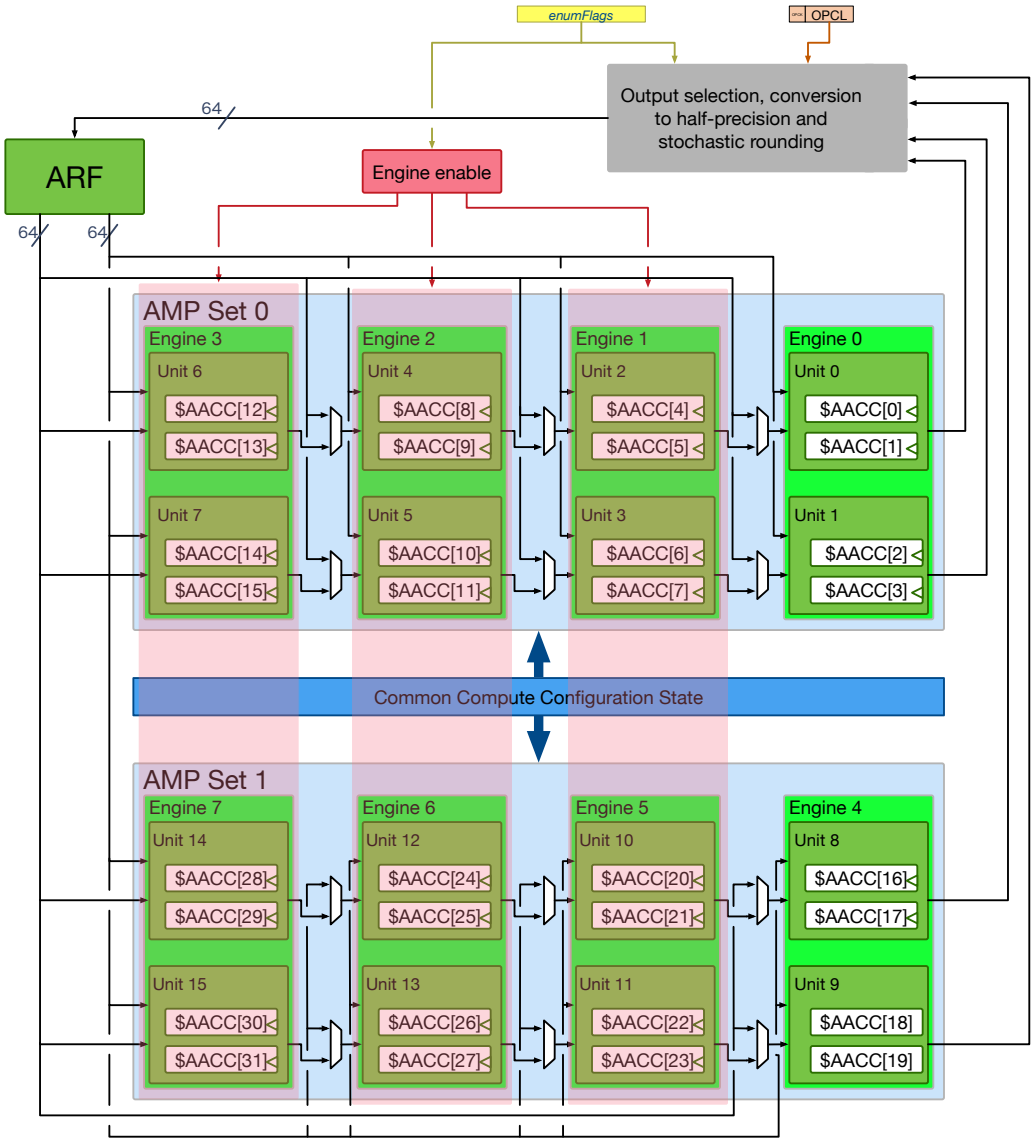


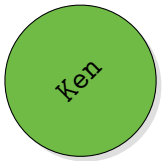
Fig. 2.32: AMP unit connectivity

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

Table 2.45: AMP/SLIC instruction variants

Instruction	Input data format	Input partial format	Output partial format	Active AMP sets
f16v4sisoamp	Half-precision (v4)	Single-precision (v2)	Single-precision (v2)	1
f16v4hihoamp	Half-precision (v4)	Half-precision (v2)	Half-precision (v2)	1
f16v4hihov4amp	Half-precision (v4)	Half-precision (v4)	Half-precision (v4)	2
f16v4sisoslic	Half-precision (v4)	Single-precision (v2)	Single-precision (v2)	1
f16v4hihoslic	Half-precision (v4)	Half-precision (v2)	Half-precision (v2)	1
f16v4hihov4slic	Half-precision (v4)	Half-precision (v4)	Half-precision (v4)	2
f32sisoamp	Single-precision (scalar)	Single-precision (v2)[5]	Single-precision (v2)[5]	1
f32sisov2amp	Single-precision (scalar)	Single-precision (v2)	Single-precision (v2)	2
f32sisoslic	Single-precision (scalar)	Single-precision (v2)[5]	Single-precision (v2)[5]	1
f32sisov2slic	Single-precision (scalar)	Single-precision (v2)	Single-precision (v2)	2
f8v8hihov4amp	Quarter-precision (v8)	Half-precision (v4)	Half-precision (v4)	2
f8v8hihov4slic	Quarter-precision (v8)	Half-precision (v4)	Half-precision (v4)	2

Implementation detail



IPU21 includes 8 of the AMP/SLIC computation engines, arranged as 2 *sets* of 4 engines, providing a peak performance of:

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

2.10.12 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 2.46: Transcendental functions

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

2.10.13 IEEE 754-2008 Clarifications

2.10.13.1 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.

2.10.14 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*
- *f8 arithmetic saturates* (maximum value depends on format and bias)
- *f8 cannot represent infinity*
- *f8 cannot represent -0.0*

2.10.14.1 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.

2.11 Exception Model

2.11.1 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*.

2.11.2 Exception Events

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

2.12 Debug

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

2.13 Exchange Interface

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

2.14 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.

2.14.1 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*.

2.14.2 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*.

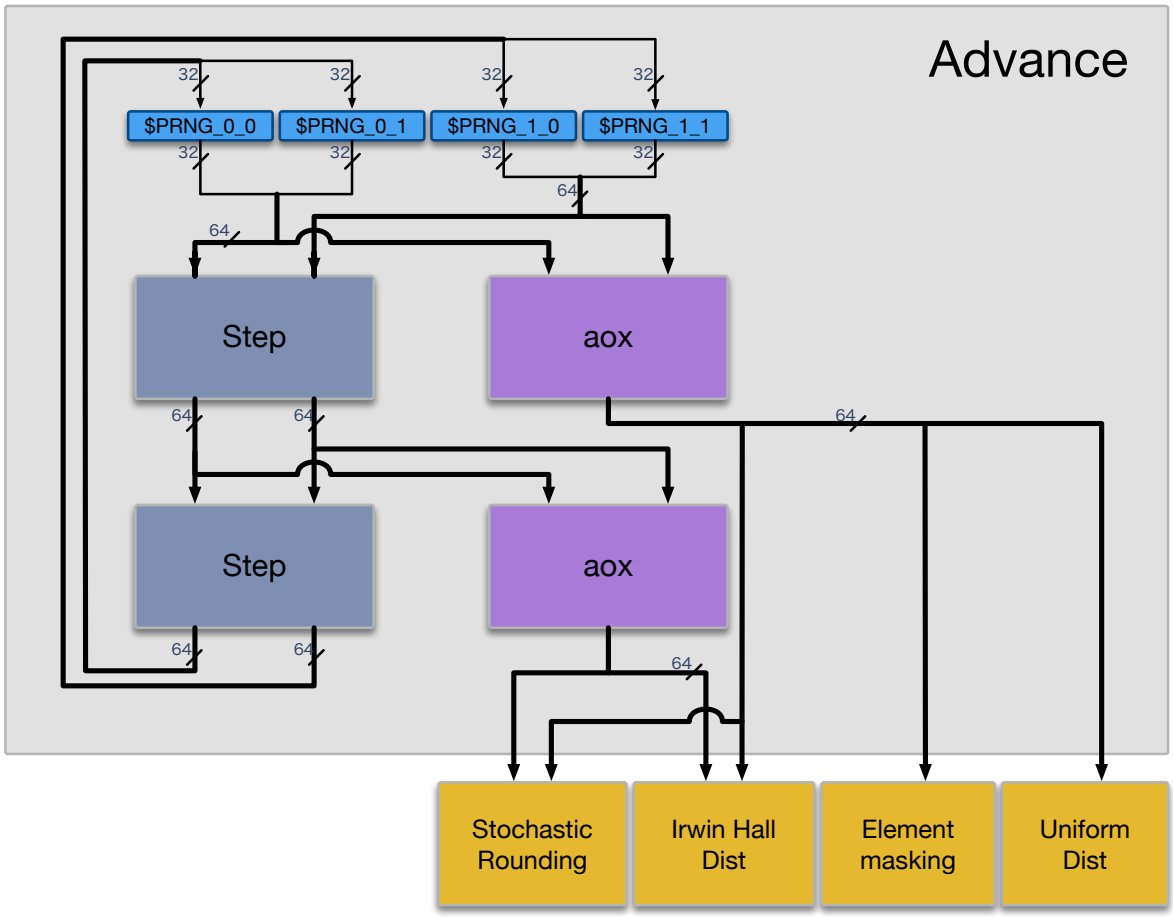


Fig. 2.33: PRNG overview

2.14.2.1 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 2.47: 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 2.10: xoroshiro128aox

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

} tprngt_Internal;
```

Listing 2.11: 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 sl = $PRNG_0_1.get() << 32 | $PRNG_0_0.get();
  uint64_t su = $PRNG_1_1.get() << 32 | $PRNG_1_0.get();

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

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

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

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

2.14.3 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:

2.14.3.1 Integer Instructions

- *urand32*
- *urand64*.

2.14.3.2 Floating-Point Conversion Instructions

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

2.14.4 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.

2.14.4.1 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 $\sqrt{\frac{1023}{1024}}$ ($\sim$0.9995)[6]
- The kurtosis of the generated distribution is $\frac{5933}{2046}$ ($\sim$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

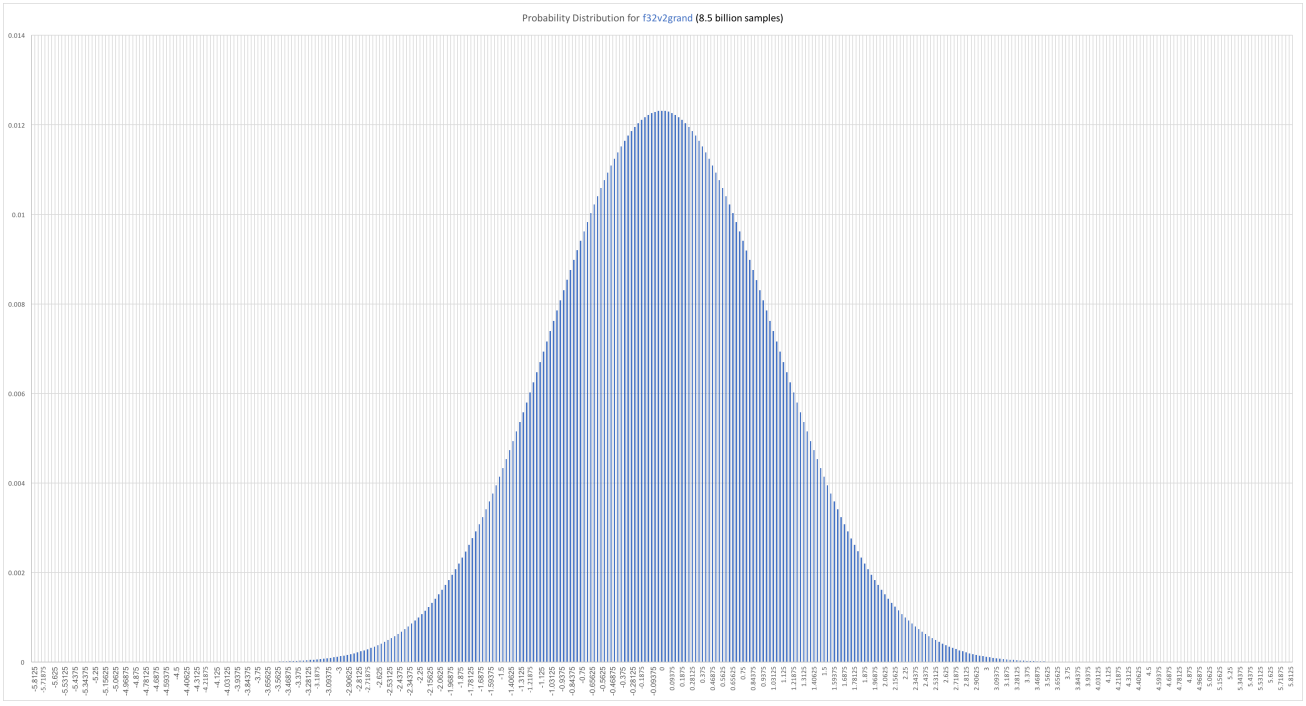


Fig. 2.34: f32v2grand probability distribution

2.14.4.2 Instructions

- *f32v2grand*
- *f16v2grand*

2.14.5 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.

2.14.5.1 Instructions

- *f32v2rmask*
- *f16v4rmask*

2.14.6 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).

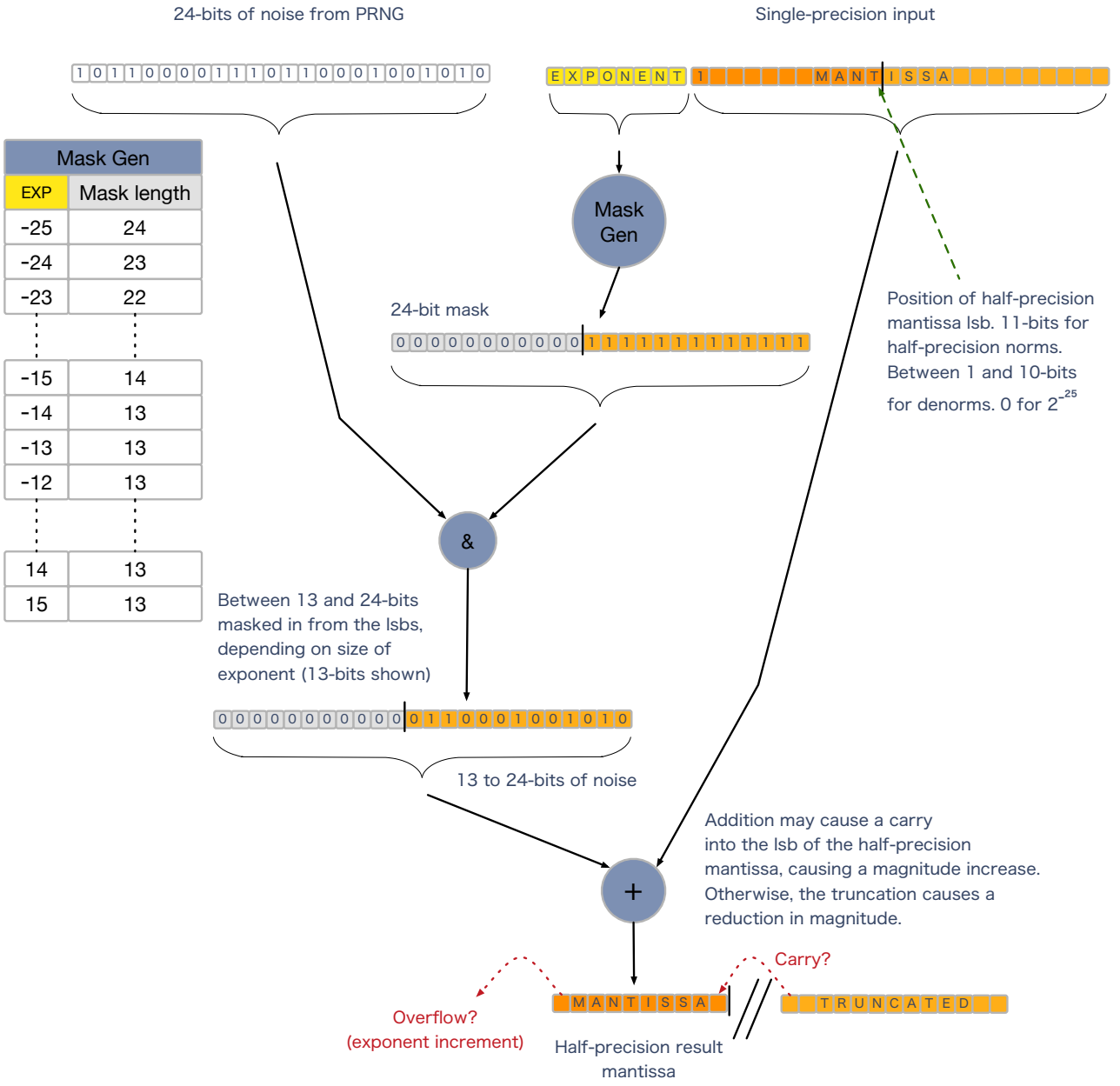


Fig. 2.35: Stochastic rounding of single to half-precision

Listing 2.12: Stochastic rounding Single to Half

```
// @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_StochasticRoundHalf(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 2.13: Stochastic rounding Single to Half

```
// @brief Apply stochastic rounding to a vector of single-precision values
void TFPU_ApplyStochasticRoundHalf(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_StochasticRoundHalf(values[i], randomBits[i]);
  }
}
```

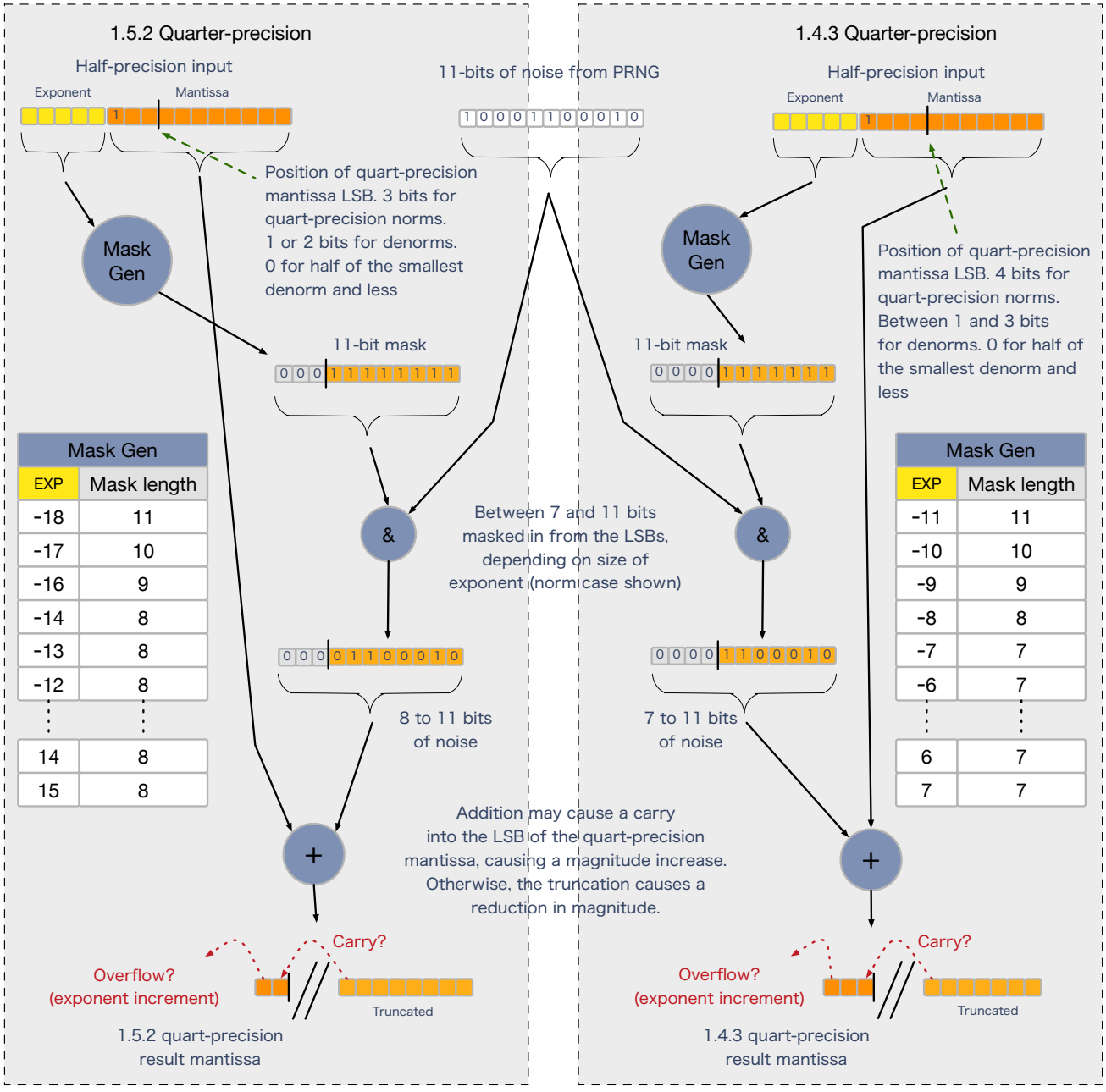


Fig. 2.36: Stochastic rounding of half-precision to quarter-precision

Listing 2.14: Stochastic rounding Half to Quart

```
// @brief Perform stochastic rounding on a half-precision value (represented using a
//        single-precision value), producing a quarter-precision result.
// @param single the half-precision value to be rounded (stored using single-precision)
// @param bitpattern the random bit-pattern of extended mantissa bits
// @param quart_exp_bits the number of exponent bits in the quarter-precision number
// @param quart_bias the bias zero offset
// @returns *single* (unmodified) if it is NaN or +/- infinity
```

```
//          +/- Zero if the rounded f32 result is less than min quart denorm
//          +/- exp2f((1 << quart_exp_bits) - quart_bias) if the rounded f32 result is
//          greater than max quarter-precision norm.
//          Otherwise, a single-precision representation of *half* stochastically rounded
//          to quarter-precision.
float TFPU_StochasticRoundF8(float single,
                             uint32_t bitpattern,
                             size_t quart_exp_bits,
                             int32_t quart_bias) {
  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);

  // Quarter-precision numbers are 8-bit
  const size_t quart_size = 8;

  // Quarter-precision numbers always have 1 sign bit
  const size_t quart_sign_size = 1;

  // Compute the number of quart mantissa bits depending on the format exponent bits
  size_t quart_mant_size = quart_size - quart_sign_size - quart_exp_bits;

  // The quart can use an exponent of all 1's - unlike f16/f32
  int32_t quart_max_exp = (1 << quart_exp_bits) - 1;

  int32_t quart_max_norm_exp = (quart_max_exp - quart_bias);
  int32_t quart_min_norm_exp = -(quart_bias - 1);
  int32_t quart_min_denorm_exp = -(quart_bias - 1 + quart_mant_size);
  if (sexponent >= quart_min_norm_exp) {
    // Quart norm - use 7 or 8 bits of random mantissa below the quart lsb.
    masklen = TFPU_F16_M_SIZE - quart_mant_size;

  } else if (sexponent >= (quart_min_denorm_exp - 1)) {
    // Quart denorms (or at least half the size of the smallest denorm)
    // - use between 8 and 11 bits of random mantissa bits below the lsb
    masklen = TFPU_F16_M_SIZE + (quart_min_denorm_exp - sexponent);

  } else {
    // Use the full 11 bits of random mantissa bits below the lsb
    masklen = TFPU_F16_M_SIZE + 1;
  }
  // Or-in the implicit 1 of the mantissa
  mantissa |= (1 << TFPU_F32_M_SIZE);

  // Mask the random bit-pattern
  size_t f16_m_size_diff = TFPU_F32_M_SIZE - TFPU_F16_M_SIZE;
  bitpattern &= ((1 << masklen) - 1);

  // Add the masked bit-pattern and add it to the mantissa
  // of the input value at the right position for f32.
  mantissa += bitpattern << f16_m_size_diff;

  // 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 quarter-precision accuracy by truncating the mantissa.
  mantissa >>= masklen + f16_m_size_diff;
  mantissa <<= masklen + f16_m_size_diff;

  // And remove the implicit 1
  mantissa <<= (32 - TFPU_F32_M_SIZE);
```

```
  mantissa >>= (32 - TFPU_F32_M_SIZE);

  // Range check result
  if (sexponent < quart_min_denorm_exp) {
    result = 0.0;

  } else if (sexponent > quart_max_norm_exp) {
    // Return a finite value above the quart range.
    // Downstream conversion to quart will then convert appropriately
    result = exp2f(quart_max_exp - quart_bias + 1);

  } else {
    result = TFPU_F32FromBits((uexponent << TFPU_F32_E_OFFSET) | (mantissa << TFPU_F32_M_OFFSET));
  }

  return copysign(result, single);
}
```

Listing 2.15: Stochastic rounding Half to Quart

```
// @brief Apply stochastic rounding to a vector of single-precision values
void TFPU_ApplyStochasticRoundF8(array<uint64_t, 2> &randm,
                                 vector<float> &values,
                                 size_t quart_exp_bits,
                                 int32_t quart_bias) {
  array<uint32_t, 8> randomBits;

  // Extract the (11-bit) random bit patterns from $PRNG_0/1 and
  // the intermediate values.
  randomBits[0] = ((randm[0] >> 0) & 0xff) | (((randm[1] >> 0) & 0x7) << 8);
  randomBits[1] = ((randm[0] >> 8) & 0xff) | (((randm[1] >> 8) & 0x7) << 8);
  randomBits[2] = ((randm[0] >> 16) & 0xff) | (((randm[1] >> 16) & 0x7) << 8);
  randomBits[3] = ((randm[0] >> 24) & 0xff) | (((randm[1] >> 24) & 0x7) << 8);
  randomBits[4] = ((randm[0] >> 32) & 0xff) | (((randm[1] >> 32) & 0x7) << 8);
  randomBits[5] = ((randm[0] >> 40) & 0xff) | (((randm[1] >> 40) & 0x7) << 8);
  randomBits[6] = ((randm[0] >> 48) & 0xff) | (((randm[1] >> 48) & 0x7) << 8);
  randomBits[7] = ((randm[0] >> 56) & 0xff) | (((randm[1] >> 56) & 0x7) << 8);

  // Apply stochastic rounding to each of the result values
  for (unsigned int i = 0; i < values.size(); i++) {
    values[i] = TFPU_StochasticRoundF8(values[i], randomBits[i], quart_exp_bits, quart_bias);
  }
}
```

The following instructions implement stochastic rounding:

- *f16v2absadd*
- *f16v2add*
- *f16v2gina*
- *f16v2mul*
- *f16v2sub*
- *f16v2tof8*
- *f16v4absadd*
- *f16v4add*
- *f16v4gacc*
- *f16v4mix*
- *f16v4mul*
- *f16v4sub*
- *f16v8tof8*

CHAPTER

THREE

INSTRUCTIONS

3.1 Instruction Signatures

Tile instructions have one of the following parameter signatures:

Table 3.1: Instruction signatures

Signature	Example	Description
instr src0	*br*	The instruction has a single register source argument (**$m4** for example, see *Register Specifiers*).
instr imm0	*bri*	The instruction has a single source argument, encoded as an immediate (**simm6** for example).
instr dst0	*f16v2grand*	The instruction has a single explicit register destination argument only (**$a2:3** for example).
instr dst0, src0	*abs*	The instruction has a single register destination argument (which is modified at instruction retirement), **$m2** for example, and a single source register argument.
instr src0, imm0	*brneg*	The instruction has one source register argument and one immediate argument.
instr dst0, imm0	*call*	The instruction has one destination register argument (which is modified at instruction retirement) and one immediate argument.
instr src0, src1	*f16v2cmac*	The instruction has two source register arguments
instr imm0, src0	*put*	The instruction has one immediate argument and one source register argument.
instr imm0, imm1	*rpt*	The instruction has two arguments encoded as immediates.
instr srcDst0, imm0	*brnzdec*	The instruction has one source and destination register argument and one immediate argument. The register argument is modified at instruction retirement.
instr dst0, src0, src1	*add*	The instruction has one destination register argument and two source register arguments.
instr dst0, src0, imm0	*add*	The instruction has one destination register argument, one source register argument and one source argument encoded as an immediate.
instr src0, src1, imm0	*f32v2aop*	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
instr src0, src1, src2	*stm32*	The instruction has three source register arguments.
instr dst0, imm0, src0	*sub*	The instruction has one destination register argument, one immediate argument and one source register argument.
instr srcDst0, src0, src1	*movz*	The instruction has one source and destination register argument and two source register arguments. The source and destination register is modified at instruction retirement.
instr src0, srcDst0+=, src1	*stm32step*	The instruction has three source register arguments, the 2nd of which is post-incremented by an amount specified by the final source register
instr dst0, src0, src1, imm0	*f16v4hihoamp*	The instruction has one destination register argument, two source register arguments and one source argument encoded as an immediate.
instr dst0, src0, src1, src2	*ld128*	The instruction has one destination register argument and three source register arguments.
instr src0, src1, src2, src3	*st32*	The instruction has four source register arguments.
instr src0, src1, src2, imm0	*st32*	The instruction has three source register arguments and one immediate.
instr dst0, src0, srcDst0+=, imm0	*ld128step*	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).
instr dst0, src0, srcDst0+=, src1	*ld128step*	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).
instr dst0, srcDst0++, src0, src1	*ld64a32*	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).
instr src0, src1, srcDst0+=, imm0	*st32step*	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).
instr src0, src1, srcDst0+=, src2	*st32step*	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).
instr src0, srcDst0+=, src1, imm0	*st64pace*	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
instr dst0, srcDst0++, src0, srcDst1@	*ldd16a32*	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.
instr dst0, src0, srcDst0++, srcDst1>>	*ldb16b16*	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 post-incremented by a value specified as a *mini delta*
instr dst0, dst1, srcDst0+=, src0, imm0	*ld2x64pace*	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.
instr dst0, src0, srcDst0+=, src1, imm0	*ld2xst64pace*	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 $a3. When specifying a register index range, the base index (2 here) must be *naturally aligned* to the size of the range. If the base index isn't naturally aligned, the least significant bits of the index are ignored and assumed to be 0. See *ARF*/*MRF* for further details.
  - a single source register index, with a broadcast modifier. Such register specifiers result in a scalar value, from the given register being *broadcast* (duplicated) to form a vector of the appropriate size for the instruction. The resulting vector is then used as the indicated source operand for the operation. For example:
    - `$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
*simm*n	An n-bit wide signed immediate. This operand is implicitly *sign extended* to *word* size.	`32'b{32-n'bsimmn[n-1],n'bsimmn>}`
*zimm*n	An n-bit wide unsigned immediate. This operand is implicitly *zero extended* (using the most-significant-bits) to *word* size.	`32'b{32-n'b0, n'bzimmn}`
*immz*n	An n-bit wide unsigned immediate. This operand is implicitly *zero tailed* (using the least-significant-bits) to *word* size.	`32'b{n'bimmzn, 32-n'b0}`
*Strimm*n*x2*	n distinct 2-bit address post-modification specifiers	
enumFlags	Flags to control the precise behaviour of an instruction	See instruction semantics

3.4 Memory Addressing Modes

3.4.1 Base Address With Scaled Unsigned Register Offset

Table 3.3: Base address with scaled unsigned register offset

Syntax	$m*Base*, $m*Offset*
Effective address(es):	EA_0 = $m*Base* + ($m*Offset* * *accessSizeInBytes*)
Register post-modification(s):	None
Example:	*stm32*

3.4.2 Base Address With Delta and Scaled Unsigned Register Offset

Table 3.4: Base address with delta and scaled unsigned register offset

Syntax	$m*Base*, $m*Delta*, $m*Off*
Effective address(es):	EA_0 = $m*Base* + $m*Delta* + ($m*Off* * *accessSizeInBytes*)
Register post-modification(s):	None
Example:	*ld32*

3.4.3 Base Address With Delta and Scaled Zero-Extended Immediate Offset

Table 3.5: Base address with delta and scaled zero-extended immediate offset

Syntax	$m*Base*, $m*Delta*, *zimm12*
Effective address(es):	EA_0 = $m*Base* + $m*Delta* + (*zimm12* * *accessSizeInBytes*)
Register post-modification(s):	None
Example:	*ld64*

3.4.4 Post-Incrementing Absolute Address

Table 3.6: Post-incrementing absolute address

Syntax	$m*Addr*++
Effective address(es):	EA_0 = $m*Addr*
Register post-modification(s):	$m*Addr* += *accessSizeInBytes*
Example:	*ld64a32*

3.4.5 Post-Incrementing Base Address With Scaled Signed Register Stride

Table 3.7: Post-incremented base address with scaled signed register stride

Syntax	$m*Base*+=, $m*Stride*
Effective address(es):	EA_0 = $m*Base*
Register post-modification(s):	$m*Base* += $m*Stride* * *accessSizeInBytes*
Example:	*stm32step*

3.4.6 Base Address With Post-Incrementing Delta and Scaled Signed Register Stride

Table 3.8: Base address with post-incremented delta and scaled signed register stride

Syntax	$m*Base*, $m*Delta*+=, $m*Stride*
Effective address(es):	EA_0 = $m*Base* + $m*Delta*
Register post-modification(s):	$m*Delta* += $m*Stride* * *accessSizeInBytes*
Example:	*lds16step*

3.4.7 Base Address With Post-Incrementing Delta and Scaled Signed Immediate Stride

Table 3.9: Base address with post-incremented delta and scaled signed immediate stride

Syntax	$m*Base*, $m*Delta*+=, *simm8*
Effective address(es):	EA_0 = $m*Base* + $m*Delta*
Register post-modification(s):	$m*Delta* += *simm8* * *accessSizeInBytes*
Example:	*lds16step*

3.4.8 Base Address With 16-Bit Delta, With Simultaneous Delta Load From Absolute, Post-Incrementing Address

Table 3.10: Base address with 16-bit delta, with simultaneous delta load from absolute, post-incrementing address

Syntax	$m*Addr*++, $m*Base*, $m*Delta*@
Effective address(es):	• EA_0 = $m*Base* + $m*Delta*[15:0] (or $m*Delta*[31:16]) • EA_1 = $m*Addr*
Register post-modification(s):	• $m*Delta* = loadHalfWord($EA_1$) • $m*Addr* += 2 (or 4)
Example(s):	*ldd16a32*, *ldd16a64*, *ldd16v2a32*

3.4.9 Base Address With Post-Incrementing Delta-Pair

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

Syntax	$m*Base*, $m*Delta*++, $m*MiniD*>>
Effective address(es):	• EA_0 = $m*Base* + $m*Delta* [15:0] • EA_1 = $m*Base* + $m*Delta* [31:16]
Register post-modification(s):	• $m*Delta* = (($m*Delta*[31:16] + 2) << 16) | ($m*Delta*[15:0] + (($m*MiniD*[3:0] + 1) << 1)) • $m*MiniD* >>= 4 (least-significant 4 bits shifted out)
Example(s):	*ldb16b16*

3.4.10 Post-Incrementing Packed Absolute Addresses (with Packed Strides)

Table 3.12: Base address with post-incrementing delta-pair

Syntax	$m*Addr0:1*+=, $m*Stride*, *Strimm*n*x2*
Effective address(es):	• EA_0 = Tile_ExtractPackedAddress($m*Addr0:1*, 0) • EA_1 = Tile_ExtractPackedAddress($m*Addr0:1*, 1) • EA_2 = Tile_ExtractPackedAddress($m*Addr0:1*, 2)[1]
Register post-modification(s):	• $stride_0$ = Tile_ExtractPackedStride($m*Stride*, *Strimm*n*x2*[1:0]) • $stride_1$ = Tile_ExtractPackedStride($m*Stride*, *Strimm*n*x2*[3:2]) • $stride_2$ = Tile_ExtractPackedStride($m*Stride*, *Strimm*n*x2*[5:4])[1] • $addrs_0$ = EA_0 + ($stride_0$ * 8) • $addrs_1$ = EA_1 + ($stride_1$ * 8) • $addrs_2$ = EA_2 + ($stride_2$ * 8)[1] • $m*Addr0* = Tile_TripleAddressPack_Lower(addrs) • $m*Addr1* = Tile_TripleAddressPack_Upper(addrs)
Example(s):	*ld2xst64pace*, *ld2x64pace*, *ldst64pace*

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

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.
  - z*n*: 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*.
  - s*n*: 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.
  - b*n*: The *n*-bit wide loaded value will be broadcast (duplicated) to fill the target register subset.

[1] If applicable

  - a*n*: 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.
  - d*n*: The *n*-bit wide loaded value is implicitly treated as a delta-offset (replacing a value previously used as a delta-offset)
  - 2x*n*: 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*
  - *8*: 8-bit, *quarter-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.
- *f8v4class* is a floating-point instruction, which operates on 4-element *quarter-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 3.13: Instruction execution sub-phases

Execution sub-phase	Description
Prepare	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.
Except In	Checks are performed on architectural state, instruction operands and on intermediate values calculated during the *Prepare* phase. The detection of a *precise* exception during this phase will result in the architectural updates defined by the *Commit* phase being squashed and a precise exception event will be raised.
Compute	The core of the operation is performed and the results computed.
Memory	Tile Memory transactions are performed using effective addresses calculated in previous phases.
Except Out	The results of the operation are checked for exceptions like result overflow. As with 'Except In', the detection of a *precise* exception during this phase will result in the architectural updates defined by the *Commit* phase being squashed and a precise exception being raised.
Commit	The architectural state changes resulting from the operations in this phase are only committed in the absence of precise exceptions detected during the *Exception In/Out* phases. Note that the architectural state changes will be committed if an *imprecise* 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:

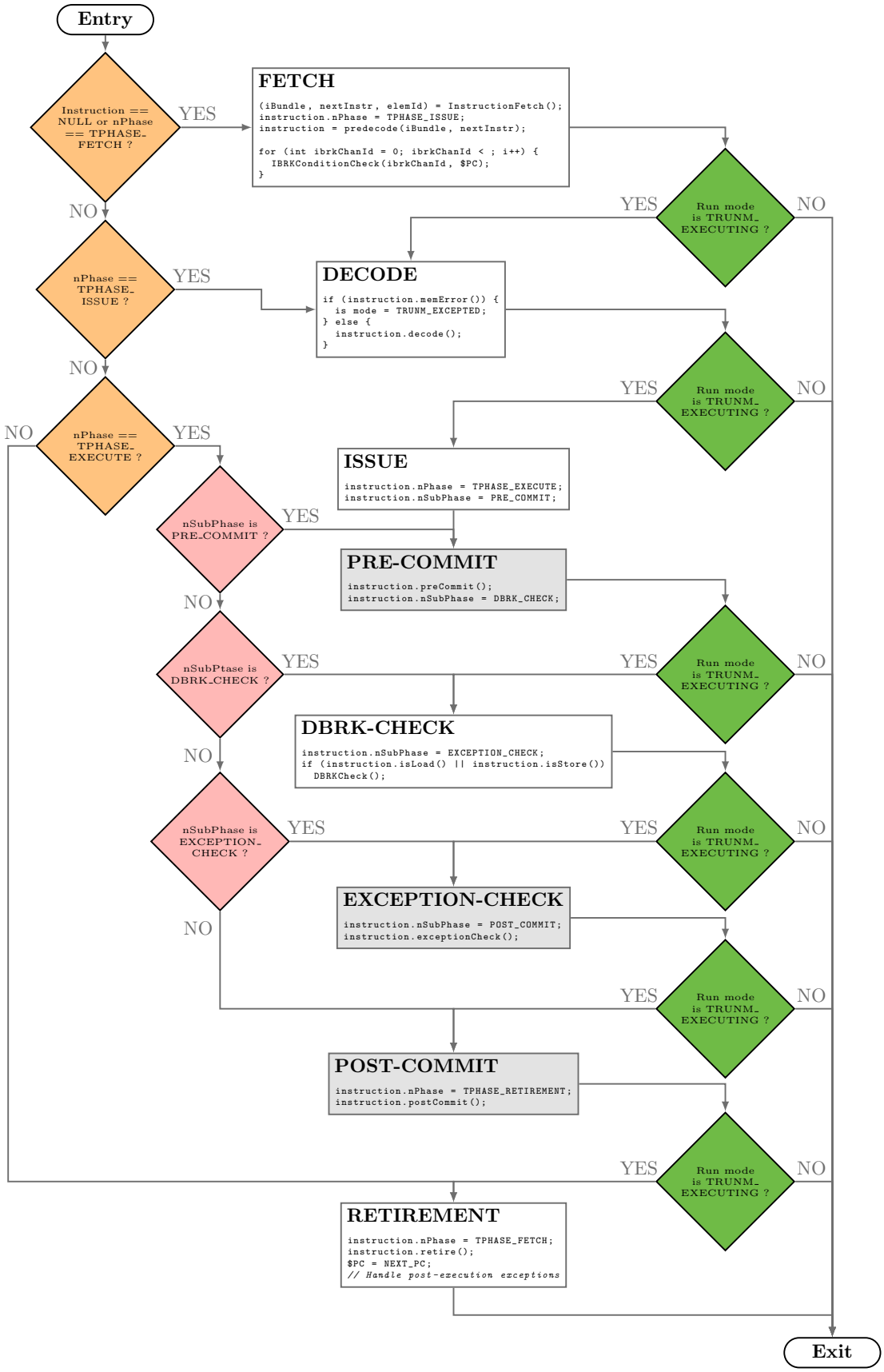


Fig. 3.1: Instruction lifespan

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 ($CTXT_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()) {
    $CTXT_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 3.14: Common functions/methods/syntax

Function/Syntax	Arguments	Description
`bit16`	*smode*	Returns the raw (16-bit) bit-pattern of a half-precision floating-point value. *smode* specifies how to treat infinity values (see bitz32())
`bitz32`	*smode*	Returns the raw bit-pattern of a half-precision value, *zero extended* to 32-bits. *smode* specifies how to treat infinity values: • *TFPU_HSATURATE_NONE*: Return unmodified infinity value • *TFPU_HSATURATE_MAX*: infinities map to +-65504 • *TFPU_HSATURATE_NAN*: infinities convert to quiet NaN
`EXCEPT`	*exceptionId*	Begins process of exception handling, using the exception ID provided. Note that any exception raised when *$REPEAT_COUNT* is non-zero will automatically set $WSR.ERPT to 0b1 and result in a *malign* exception launch. See *Exception Model*
`hadMemoryConflict`	*address*	Returns *true* if and only if there is a *memory element conflict* involving this memory access.
`hadRangedMemoryConflict`	*address*, *topAddress*	Returns *true* if and only if there is a *memory element conflict* involving memory access within the range of addresses.
`isSupervisor`		Returns *true* if being run on the *supervisor*, *false* otherwise.
`pickHalf`	*value*, *zeroOrOne*, *preserveInfMode*= *TFPU_PROP_NEITHER*	Returns one of two *half-precision* values from the 32-bit argument passed: See the *Half Precision Class* for details. • If *zeroOrOne* is 0, returns the half-precision floating-point value residing in the bottom 16-bits of *value*. • Otherwise returns the half-precision floating-point value residing in the upper 16-bits of *value* • *preserveInfMode* specifies which infinity values are left unmodified (i.e. preserved, or propagated). An unpreserved infinity value is converted to a signaling NaN. Default is for neither infinity to be propagated.
`pickQuart`	*value*, *quarter*, *qfmt*	Returns one of four *quarter-precision* values from the 32-bit argument passed. See the *Quarter Precision Class* for details.
`readState`	*csrIndex*	Returns the current value of the *CSR* at index *csrIndex*
`writeState`	*csrIndex*, *value*	Writes value to *CSR* at index *csrIndex*
`setNextRunMode`	*runMode*	Sets the *run mode* the *context* will transition to after the instruction completes.
`setExitValue`	*exitBool*	• Set *Worker* exit status to *exitBool*. • *Supervisor* *$SSR*.**LC** &= *exitBool*
`setRunMode`	*runMode*	Sets the *run mode* for the executing *context*.
`<(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 3.15: bit instructions summary

Mnemonic	Super?	Worker?	main?	aux?	Brief
and	✓	✓	✓	✓	32-bit bitwise logical AND
and64	✗	✓	✗	✓	64-bit bitwise logical AND
andc	✓	✓	✓	✓	32-bit bitwise logical AND Complement
andc64	✗	✓	✗	✓	64-bit bitwise logical AND Complement
bitrev8	✓	✓	✓	✗	Byte-wise bit order reversal
clz	✓	✓	✓	✗	Count leading zero bits
cms	✓	✓	✓	✗	Count matching sign bits
not	✗	✓	✗	✓	32-bit bitwise logical NOT
not64	✗	✓	✗	✓	64-bit bitwise logical NOT
or	✓	✓	✓	✓	Bitwise OR
or64	✗	✓	✗	✓	64-bit bitwise logical OR
popc	✓	✓	✓	✗	Population count
roll16	✓	✓	✓	✓	roll16 SIMD permutation
roll32	✗	✓	✗	✓	roll32 SIMD permutation
roll8l	✓	✓	✓	✓	roll8-left SIMD permutation
roll8r	✓	✓	✓	✓	roll8-right SIMD permutation
setzi	✓	✓	✓	✓	Register set from immediate
shuf8x8hi	✓	✓	✓	✓	8 x 8-bit SIMD permutation
shuf8x8lo	✓	✓	✓	✓	8 x 8-bit SIMD permutation
sort4x16hi	✓	✓	✓	✓	4 x 16-bit SIMD permutation
sort4x16lo	✓	✓	✓	✓	4 x 16-bit SIMD permutation
sort4x32hi	✗	✓	✗	✓	4 x 32-bit SIMD permutation
sort4x32lo	✗	✓	✗	✓	4 x 32-bit SIMD permutation
sort8	✓	✓	✓	✓	4 x 8-bit SIMD permutation
sort8x8hi	✓	✓	✓	✓	8 x 8-bit SIMD permutation
sort8x8lo	✓	✓	✓	✓	8 x 8-bit SIMD permutation
swap8	✓	✓	✓	✓	4 x 8-bit SIMD permutation
xnor	✓	✓	✓	✗	Bitwise NOT XOR
xor	✓	✓	✓	✗	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 3.16: *and* instruction definition

and			**both**	**main**
Syntax	`and $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `and $m`*Dst0*`, $m`*Src0*`, `*zimm12*			
and			**worker**	**aux**
Syntax	`and $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `and $a`*Dst0*`, $a`*Src0*`, `*immz12* `and $a`*Dst0*`, $a`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op1 = <($mSrc0, $aSrc0)>;` `DataWord op2 = <($mSrc1, zimm12, $aSrc1, (immz12 << 20))>;`		
	Compute	`DataWord result = op1 & op2;`		
	Commit	`<($mDst0, $aDst0)> = result;`		

3.7.1.2 and64

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

Table 3.17: *and64* instruction definition

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

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 3.18: *andc* instruction definition

andc			both	main
Syntax	andc $m*Dst0*, $m*Src0*, $m*Src1* andc $m*Dst0*, $m*Src0*, *zimm12*			
andc			**worker**	**aux**
Syntax	andc $a*Dst0*, $a*Src0*, $a*Src1* andc $a*Dst0*, $a*Src0*, *immz12* andc $a*Dst0*, $a*Src0*, *zimm12*			
Semantics	**Prepare**	`DataWord op1 = <($mSrc0, $aSrc0)>;` `DataWord op2 = <($mSrc1, zimm12, $aSrc1, (immz12 << 20))>;`		
	Compute	`DataWord result = op1 & ~op2;`		
	Commit	`<($mDst0, $aDst0)> = result;`		

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 3.19: *andc64* instruction definition

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

3.7.1.5 bitrev8

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

Table 3.20: *bitrev8* instruction definition

bitrev8			both	main
Syntax	`bitrev8 $m`*Dst0*`, $m`*Src0*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;`		
	Compute	`DataWord result = 0;` `for (i = 0; i < 32; i++) {` `  if (op1 & (1 << i)) {` `  result |= 1 << ((i & 24) | (7-(i & 7)));` `  }` `}`		
	Commit	`$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 3.21: *clz* instruction definition

clz			both	main
Syntax	`clz $m`*Dst0*`, $m`*Src0*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;`		
	Compute	`int i = 31;` `while (i >= 0 && ((op1 >> i) & 1) == 0) {` `  i = i - 1;` `}` `SignedDataWord result = 31 - i;`		
	Commit	`$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 3.22: *cms* instruction definition

cms			both	main
Syntax	`cms $m`*Dst0*`, $m`*Src0*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;`		
	Compute	`unsigned signBit = ((op1 >> 31) & 1);` `int i = 31;` `while ( (i > 0) && (signBit == ((op1 >> (i - 1)) & 1)) ) {` `  i = i - 1;` `}` `SignedDataWord result = 31 - i;`		
	Commit	`$mDst0 = result;`		

3.7.1.8 not

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

Table 3.23: *not* instruction definition

not			**worker**	**aux**
Syntax	`not $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`DataWord op1 = $aSrc0;`		
	Compute	`DataWord result = ~op1;`		
	Commit	`$aDst0 = result;`		

3.7.1.9 not64

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

Table 3.24: *not64* instruction definition

not64			**worker**	**aux**
Syntax	`not64 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<DataWord,2> op1 = {` `  $aSrc0:Src0+1[0],` `  $aSrc0:Src0+1[1]` `};` `array<DataWord,2> result;`		
	Compute	`result[0] = ~op1[0];` `result[1] = ~op1[1];`		
	Commit	`$aDst0:Dst0+1 = {` `  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 3.25: *or* instruction definition

or			**both**	**main**
Syntax	`or $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `or $m`*Dst0*`, $m`*Src0*`, `*immz12* `or $m`*Dst0*`, $m`*Src0*`, `*zimm12*			
or			**worker**	**aux**
Syntax	`or $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `or $a`*Dst0*`, $a`*Src0*`, `*immz12* `or $a`*Dst0*`, $a`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op1 = <($mSrc0, $aSrc0)>;` `DataWord op2 = <($mSrc1, zimm12, (immz12 << 20), $aSrc1)>;`		
	Compute	`DataWord result = op1 | op2;`		
	Commit	`<($mDst0, $aDst0)> = 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 3.26: *or64* instruction definition

or64			worker	aux
Syntax	or64 $a*Dst0*:*Dst0+1*, $a*Src0*:*Src0+1*, $a*Src1*:*Src1+1*			
Semantics	**Prepare**	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;		
	Compute	result[0] = op1[0] | op2[0]; result[1] = op1[1] | op2[1];		
	Commit	$aDst0:Dst0+1 = { result[0], result[1] };		

3.7.1.12 popc

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

Table 3.27: *popc* instruction definition

popc			both	main
Syntax	popc $m*Dst0*, $m*Src0*			
Semantics	**Prepare**	DataWord op1 = $mSrc0;		
	Compute	DataWord count = 0; for (i = 0; i < 32; i++) { count += (op1 >> i) & 1; }		
	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 3.28: *roll16* instruction definition

roll16			both	main
Syntax	`roll16 $m`*Dst0*`, $m`*Src0*`, $m`*Src1*			
roll16			**worker**	**aux**
Syntax	`roll16 $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<HalfDataWord,2> op0;` `array<HalfDataWord,2> op1 = {` `  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xffff),` `  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xffff)` `};` `array<HalfDataWord,2> op2 = {` `  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xffff),` `  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xffff)` `};`		
	Compute	`op0 = { op1[1], op2[0] };`		
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xffff) << 0) |` `  ((op0[1] & 0xffff) << 16)` `};`		

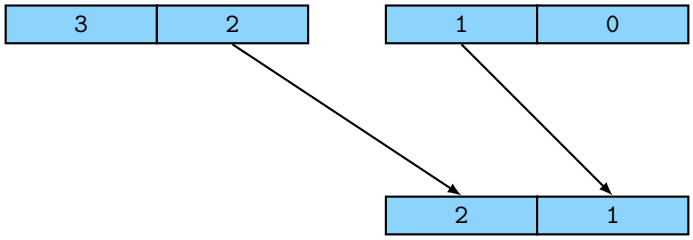


Fig. 3.2: roll16 example

3.7.1.14 roll32

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

Table 3.29: *roll32* instruction definition

roll32			worker	aux
Syntax	`roll32 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<DataWord,2> op0;` `array<DataWord,2> op1 = {` `  $aSrc0:Src0+1[0],` `  $aSrc0:Src0+1[1]` `};` `array<DataWord,2> op2 = {` `  $aSrc1:Src1+1[0],` `  $aSrc1:Src1+1[1]` `};`		
	Compute	`op0 = { op1[1], op2[0] };`		
	Commit	`$aDst0:Dst0+1 = {` `  op0[0],` `  op0[1]` `};`		

3.7.1.15 roll8l

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

Table 3.30: *roll8l* instruction definition

roll8l		**both**	**main**
Syntax	`roll8l $m`*Dst0*`, $m`*Src0*`, $m`*Src1*		
roll8l		**worker**	**aux**
Syntax	`roll8l $a`*Dst0*`, $a`*Src0*`, $a`*Src1*		
Semantics	**Prepare**		

```
array<QuarterDataWord,4> op0;
array<QuarterDataWord,4> op1 = {
  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)
};
array<QuarterDataWord,4> op2 = {
  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)
};
```

Compute

```
op0 = { op1[3], op2[0], op2[1], op2[2] };
```

Commit

```
<($mDst0, $aDst0)> = {
  ((op0[0] & 0xff) << 0) |
  ((op0[1] & 0xff) << 8) |
  ((op0[2] & 0xff) << 16) |
  ((op0[3] & 0xff) << 24)
};
```

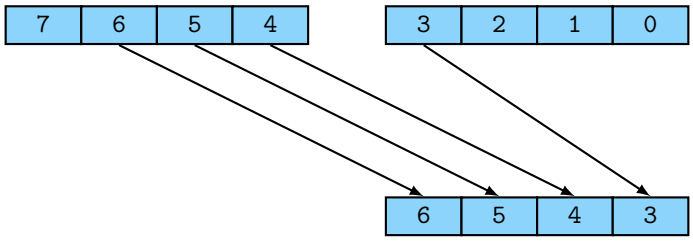


Fig. 3.3: roll8l example

3.7.1.16 roll8r

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

Table 3.31: *roll8r* instruction definition

roll8r		both	main
Syntax	`roll8r $m`*Dst0*`, $m`*Src0*`, $m`*Src1*		
roll8r		**worker**	**aux**
Syntax	`roll8r $a`*Dst0*`, $a`*Src0*`, $a`*Src1*		
Semantics	**Prepare**		

```
array<QuarterDataWord,4> op0;
array<QuarterDataWord,4> op1 = {
  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)
};
array<QuarterDataWord,4> op2 = {
  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)
};
```

Compute

```
op0 = { op1[1], op1[2], op1[3], op2[0] };
```

Commit

```
<($mDst0, $aDst0)> = {
  ((op0[0] & 0xff) << 0) |
  ((op0[1] & 0xff) << 8) |
  ((op0[2] & 0xff) << 16) |
  ((op0[3] & 0xff) << 24)
};
```

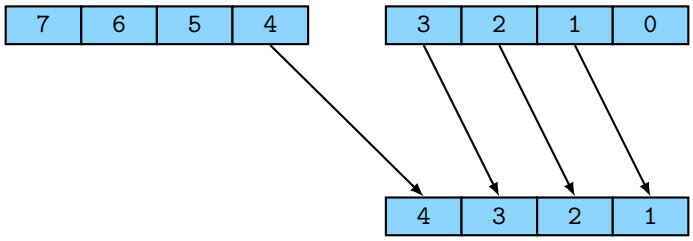


Fig. 3.4: roll8r example

3.7.1.17 setzi

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

Table 3.32: *setzi* instruction definition

setzi		both	main
Syntax	`setzi $m`*Dst0*`, `*zimm20*		
setzi		**worker**	**aux**
Syntax	`setzi $a`*Dst0*`, `*zimm20*		
Semantics	**Prepare**		

```
DataWord op1 = zimm20;
```

Commit

```
<($mDst0, $aDst0)> = op1;
```

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 3.33: *shuf8x8hi* instruction definition

shuf8x8hi		**both**	**main**
Syntax	`shuf8x8hi $mDst0, $mSrc0, $mSrc1`		
shuf8x8hi		**worker**	**aux**
Syntax	`shuf8x8hi $aDst0, $aSrc0, $aSrc1`		
Semantics	**Prepare**		

```
array<QuarterDataWord,4> op0;
array<QuarterDataWord,4> op1 = {
  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)
};
array<QuarterDataWord,4> op2 = {
  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)
};
```

Compute

```
op0 = { op1[2], op2[2], op1[3], op2[3] };
```

Commit

```
<($mDst0, $aDst0)> = {
  ((op0[0] & 0xff) << 0) |
  ((op0[1] & 0xff) << 8) |
  ((op0[2] & 0xff) << 16) |
  ((op0[3] & 0xff) << 24)
};
```

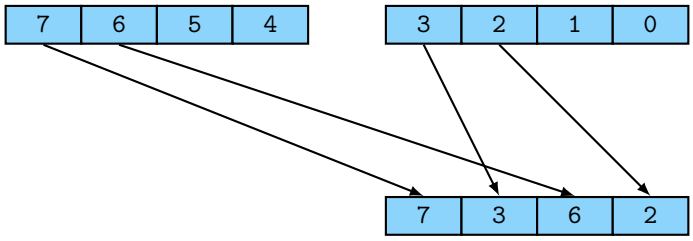


Fig. 3.5: shuf8x8hi example

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 3.34: *shuf8x8lo* instruction definition

shuf8x8lo			**both**	**main**
Syntax	`shuf8x8lo $m`*Dst0*`, $m`*Src0*`, $m`*Src1*			
shuf8x8lo			**worker**	**aux**
Syntax	`shuf8x8lo $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<QuarterDataWord,4> op1 = {` `  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)` `};` `array<QuarterDataWord,4> op2 = {` `  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),` `  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),` `  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),` `  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)` `};`		
	Compute	`op0 = { op1[0], op2[0], op1[1], op2[1] };`		
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

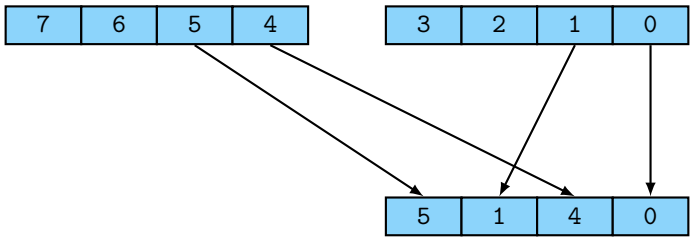


Fig. 3.6: shuf8x8lo example

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 3.35: *sort4x16hi* instruction definition

sort4x16hi		both	main
Syntax	`sort4x16hi $m`*Dst0*`, $m`*Src0*`, $m`*Src1*		
sort4x16hi		**worker**	**aux**
Syntax	`sort4x16hi $a`*Dst0*`, $a`*Src0*`:BL, $a`*Src1* `sort4x16hi $a`*Dst0*`, $a`*Src0*`, $a`*Src1*		
Semantics	**Prepare**	`array<HalfDataWord,2> op0;` `array<HalfDataWord,2> op1 = {` `  ((<($mSrc0, $aSrc0, $aSrc0:BL)>[0] >> 0) & 0xffff),` `  ((<($mSrc0, $aSrc0, $aSrc0:BL)>[0] >> 16) & 0xffff)` `};` `array<HalfDataWord,2> op2 = {` `  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xffff),` `  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xffff)` `};`		
	Compute	`op0 = { op1[1], op2[1] };`		
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xffff) << 0) |` `  ((op0[1] & 0xffff) << 16)` `};`		

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 3.36: *sort4x16lo* instruction definition

sort4x16lo		both	main
Syntax	`sort4x16lo $m`*Dst0*`, $m`*Src0*`, $m`*Src1*		
sort4x16lo		**worker**	**aux**
Syntax	`sort4x16lo $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `sort4x16lo $a`*Dst0*`, $a`*Src0*`:BU, $a`*Src1*		
Semantics	**Prepare**	`array<HalfDataWord,2> op0;` `array<HalfDataWord,2> op1 = {` `  ((<($mSrc0, $aSrc0, $aSrc0:BU)>[0] >> 0) & 0xffff),` `  ((<($mSrc0, $aSrc0, $aSrc0:BU)>[0] >> 16) & 0xffff)` `};` `array<HalfDataWord,2> op2 = {` `  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xffff),` `  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xffff)` `};`		
	Compute	`op0 = { op1[0], op2[0] };`		
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xffff) << 0) |` `  ((op0[1] & 0xffff) << 16)` `};`		

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 3.37: *sort4x32hi* instruction definition

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

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 3.38: *sort4x32lo* instruction definition

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

3.7.1.24 sort8

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

Table 3.39: *sort8* instruction definition

sort8			both	main
Syntax	`sort8 $m`*Dst0*`, $m`*Src0*			
sort8			**worker**	**aux**
Syntax	`sort8 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<QuarterDataWord,4> op1 = {` `  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)` `};`		
	Compute	`op0 = { op1[0], op1[2], op1[1], op1[3] };`		
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

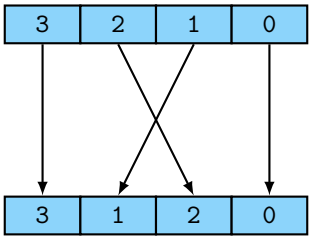


Fig. 3.7: sort8 example

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 3.40: *sort8x8hi* instruction definition

sort8x8hi		both	main
Syntax	`sort8x8hi $mDst0, $mSrc0, $mSrc1`		
sort8x8hi		**worker**	**aux**
Syntax	`sort8x8hi $aDst0, $aSrc0, $aSrc1`		
Semantics	**Prepare**		

```
array<QuarterDataWord,4> op0;
array<QuarterDataWord,4> op1 = {
  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)
};
array<QuarterDataWord,4> op2 = {
  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)
};
```

Compute

```
op0 = { op1[1], op1[3], op2[1], op2[3] };
```

Commit

```
<($mDst0, $aDst0)> = {
  ((op0[0] & 0xff) << 0) |
  ((op0[1] & 0xff) << 8) |
  ((op0[2] & 0xff) << 16) |
  ((op0[3] & 0xff) << 24)
};
```

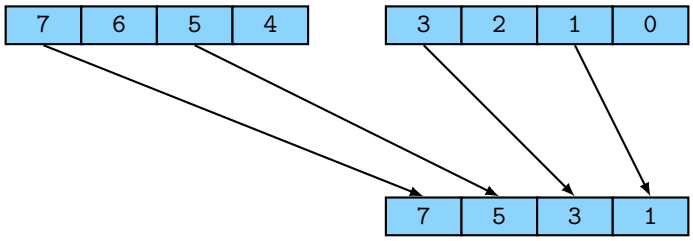


Fig. 3.8: sort8x8hi example

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 3.41: *sort8x8lo* instruction definition

sort8x8lo		**both**	**main**
Syntax	`sort8x8lo $m`*Dst0*`, $m`*Src0*`, $m`*Src1*		
sort8x8lo		**worker**	**aux**
Syntax	`sort8x8lo $a`*Dst0*`, $a`*Src0*`, $a`*Src1*		
Semantics	**Prepare**		

```
array<QuarterDataWord,4> op0;
array<QuarterDataWord,4> op1 = {
  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),
  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)
};
array<QuarterDataWord,4> op2 = {
  ((<($mSrc1, $aSrc1)>[0] >> 0) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 8) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 16) & 0xff),
  ((<($mSrc1, $aSrc1)>[0] >> 24) & 0xff)
};
```

Compute

```
op0 = { op1[0], op1[2], op2[0], op2[2] };
```

Commit

```
<($mDst0, $aDst0)> = {
  ((op0[0] & 0xff) << 0) |
  ((op0[1] & 0xff) << 8) |
  ((op0[2] & 0xff) << 16) |
  ((op0[3] & 0xff) << 24)
};
```

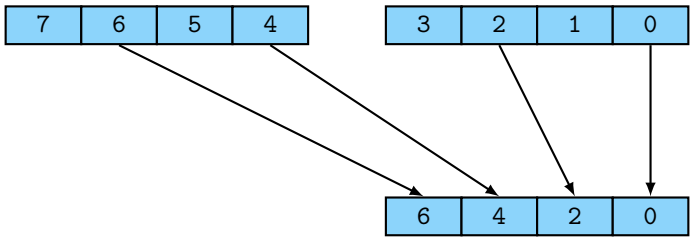


Fig. 3.9: sort8x8lo example

3.7.1.27 swap8

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

Table 3.42: *swap8* instruction definition

swap8		both	main
Syntax	`swap8 $mDst0, $mSrc0`		
swap8		**worker**	**aux**
Syntax	`swap8 $aDst0, $aSrc0`		
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<QuarterDataWord,4> op1 = {` `  ((<($mSrc0, $aSrc0)>[0] >> 0) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 8) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 16) & 0xff),` `  ((<($mSrc0, $aSrc0)>[0] >> 24) & 0xff)` `};`	
	Compute	`op0 = { op1[1], op1[0], op1[3], op1[2] };`	
	Commit	`<($mDst0, $aDst0)> = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`	

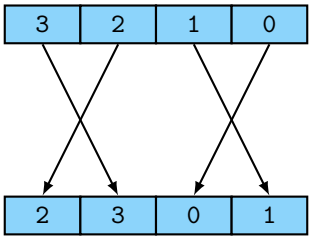


Fig. 3.10: swap8 example

3.7.1.28 xnor

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

Table 3.43: *xnor* instruction definition

xnor		both	main
Syntax	`xnor $mDst0, $mSrc0, $mSrc1`		
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 3.44: *xor* instruction definition

xor			both	main
Syntax	`xor $m`*Dst0*`, $m`*Src0*`, $m`*Src1*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;` `DataWord op2 = $mSrc1;`		
	Compute	`DataWord result = op1 ^ op2;`		
	Commit	`$mDst0 = result;`		

3.7.2 Control

Table 3.45: control instructions summary

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

3.7.2.1 br

Unconditional absolute branch to register target address.

Table 3.46: *br* instruction definition

br			both	main
Syntax	`br $mSrc0`			
Semantics	**Prepare**	`DataWord op0 = $mSrc0;` `DataWord target = op0;`		
	Except In	`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);` `}`		
	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 3.47: *bri* instruction definition

bri			both	main
Syntax	`bri` *zimm19*			
Semantics	**Prepare**	`DataWord op0 = (zimm19 << 2);` `DataWord target = op0;`		
	Except In	`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);` `}`		
	Commit	`TARGET_PC = target;`		

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 3.48: *brneg* instruction definition

brneg			both	main
Syntax	`brneg $m`*Src0*`,` *zimm19*			
Semantics	**Prepare**	`DataWord op0 = $mSrc0;` `DataWord op1 = (zimm19 << 2);` `DataWord target = op1;`		
	Except In	`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);` `}`		
	Commit	`if ((op0 >> 31) & 1) {` `  TARGET_PC = target;` `}`		

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 3.49: *brnz* instruction definition

brnz			both	main
Syntax	brnz $m*Src0*, *zimm19*			
Semantics	**Prepare**	DataWord op0 = $mSrc0; DataWord op1 = (zimm19 << 2); DataWord target = op1;		
	Except In	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); }		
	Commit	if (op0 != 0) { TARGET_PC = target; }		

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 3.50: *brnzdec* instruction definition

brnzdec			both	main
Syntax	brnzdec $m*SrcDst0*, *zimm19*			
Semantics	**Prepare**	DataWord op0 = $mSrcDst0; DataWord op1 = (zimm19 << 2); DataWord target = op1;		
	Except In	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); }		
	Commit	if (op0 != 0) { TARGET_PC = target; } $mSrcDst0 = op0 - 1;		

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 3.51: *brpos* instruction definition

brpos			both	main
Syntax	brpos $m*Src0*, *zimm19*			
Semantics	**Prepare**	DataWord op0 = $mSrc0; DataWord op1 = (zimm19 << 2); DataWord target = op1;		
	Except In	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); }		
	Commit	if (((op0 >> 31) & 1) == 0) { TARGET_PC = target; }		

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 3.52: *brz* instruction definition

brz			both	main
Syntax	brz $m*Src0*, *zimm19*			
Semantics	**Prepare**	DataWord op0 = $mSrc0; DataWord op1 = (zimm19 << 2); DataWord target = op1;		
	Except In	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); }		
	Commit	if (op0 == 0) { TARGET_PC = target; }		

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 3.53: *call* instruction definition

call			both	main
Syntax	`call $m`*Dst0*`, `*zimm20*			
Semantics	**Prepare**	`DataWord op1 = (zimm20 << 2);` `DataWord target = (op1 & TMEM_FULL_ADDRESS_MASK);`		
	Except In	`if (!isSupervisor() && (0 != $REPEAT_COUNT) && !isExecutingTDI()) {` `  // Cannot execute this instruction in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `} else if (!TMem_IsValidAddress(target)) {` `  // Target isn't within valid memory range` `  EXCEPT(TEXCPT_INVALID_PC);` `} else if (!TMem_AddressIsExecutable(target)) {` `  // Target isn't within an executable memory region` `  EXCEPT(TEXCPT_INVALID_PC);` `}`		
	Commit	`// Store return address in op0` `TARGET_PC = target;` `$mDst0 = COISSUE ? $PC + 8 : $PC + 4;`		

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 3.54: *exitneg* instruction definition

exitneg			worker	main
Syntax	exitneg $m*Src0*			
Semantics	**Prepare**	`SignedDataWord op0 = $mSrc0;`		
	Except In	`if (0 != $REPEAT_COUNT) {` `  // Cannot execute this instruction in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `}`		
	Commit	`setExitValue(op0 < 0);` `// run mode will transition to INACTIVE at instruction retirement` `setNextRunMode(TRUNM_INACTIVE);`		

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 3.55: *exitnz* instruction definition

exitnz			worker	main
Syntax	exitnz $m*Src0*			
Semantics	**Prepare**	`DataWord op0 = $mSrc0;`		
	Except In	`if (0 != $REPEAT_COUNT) {` `  // Cannot execute this instruction in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `}`		
	Commit	`setExitValue(op0 != 0);` `// run mode will transition to INACTIVE at instruction retirement` `setNextRunMode(TRUNM_INACTIVE);`		

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 3.56: *exitpos* instruction definition

exitpos			worker	main
Syntax	exitpos $m*Src0*			
Semantics	**Prepare**	`SignedDataWord op0 = $mSrc0;`		
	Except In	`if (0 != $REPEAT_COUNT) {` `  // Cannot execute this instruction in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `}`		
	Commit	`setExitValue(op0 >= 0);` `// run mode will transition to INACTIVE at instruction retirement` `setNextRunMode(TRUNM_INACTIVE);`		

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 3.57: *exitz* instruction definition

exitz			worker	main
Syntax	exitz $m*Src0*			
Semantics	**Prepare**	`DataWord op0 = $mSrc0;`		
	Except In	`if (0 != $REPEAT_COUNT) {` `  // Cannot execute this instruction in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `}`		
	Commit	`setExitValue(op0 == 0);` `// run mode will transition to INACTIVE at instruction retirement` `setNextRunMode(TRUNM_INACTIVE);`		

Architectural state references: *$REPEAT_COUNT*

3.7.2.13 rpt

rpt provides a zero-overhead loop facility, causing the subsequent sequence of *Execution Bundle*s (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 Bundle*s). 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 Bundle*s 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 3.58: *rpt* instruction definition

rpt			worker	main
Syntax	`rpt $m`*Src0*`, `*zimm8* `rpt `*zimm12*`, `*zimm8*			
Semantics	**Prepare**	`DataWord op0 = <($mSrc0, zimm12)>;` `DataWord op1 = (zimm8 << 3);` `DataWord nextPC = COISSUE ? $PC + 8 : $PC + 4;` `DataWord pcAfterRptBody = nextPC + (op1 + 8);`		
	Except In	`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);` `}`		
	Commit	`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;` `}`		

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 *f16* value to *single-precision*.

Table 3.59: *f16tof32* instruction definition

f16tof32			worker	aux
Syntax	`f16tof32 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Half op1 = pickHalf($aSrc0, 0, PROP_INF);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `if (op1.issNaN()) {` `  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 3.60: *f16v2sufromui* instruction definition

f16v2sufromui			worker	aux
Syntax	`f16v2sufromui $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<HalfDataWord,2> op1 = {` `  (($aSrc0[0] >> 0) & 0xffff),` `  (($aSrc0[0] >> 16) & 0xffff)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<Single,2> fval;`		
	Compute	`for (i = 0; i < 2; i++) {` `  // Double the magnitude of the input value` `  // and shift the result to produce a symmetric` `  // distribution centred on 0 (resulting range here is [-65535, 65535])` `  int32_t valp = (2 * op1[i]) - ((1 << 16) - 1);` `  // Scale the result so that the resulting output range is [-0.5, 0.5]` `  fval[i] = valp / exp2(17);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(fval[0]).bitz32(smode) |` `  (Half(fval[1]).bitz32(smode) << 16)` `};`		

Architectural state references: *$FP_CTL*

Function references: *TFPU_GetNanooMode*

3.7.3.1.3 f16v2tof32

f16 floating-point pair to *single-precision* conversion

Table 3.61: *f16v2tof32* instruction definition

f16v2tof32			worker	aux
Syntax	`f16v2tof32 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `if (op1[0].issNaN() || op1[1].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (op1[0].isNaN()) {` `  result[0] = TFPU_F32_QuietenNan((Single)op1[0]);` `} else {` `  result[0] = (Single)op1[0];` `}` `if (op1[1].isNaN()) {` `  result[1] = TFPU_F32_QuietenNan((Single)op1[1]);` `} else {` `  result[1] = (Single)op1[1];` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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

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

3.7.3.1.4 f16v2tof8

Half-precision floating-point 2-element vector to *quarter-precision* 2-element vector conversion.

Table 3.62: *f16v2tof8* instruction definition

f16v2tof8			worker	aux
Syntax	`f16v2tof8 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `array<Quart,2> resultQ;` `vector<Single> result = { op1[0], op1[1] };`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `}` `for (i = 0; i < 2; i++) {` `  if (op1[i].issNaN()) {` `  fpExcpt |= TFPEXCPT_INV;` `  } else if (op1[i].isqNaN()) {` `  // No exception` `  } else if (op1[i].isInf()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}`		
	Compute	`Quart q = Quart(qArfFmt);` `int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `for (i = 0; i < 2; i++) {` `  if (!op1[i].isNaN() && !op1[i].isInf()) {` `  result[i] = result[i] * exp2f(scale);` `  }` `}` `if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundF8(randomBits, result, q.expSize(), q.bias());` `}` `for (i = 0; i < 2; i++) {` `  fpExcpt |= TFPU_GenOFLOCheckF8(` `  result[i], q.mantSize(), q.maxValue(), nanoo);` `}` `for (i = 0; i < 2; i++) {` `  if (op1[i].isNaN() || op1[i].isInf()) {` `  resultQ[i] = Quart(qArfFmt, QUART_ERROR);` `  } else {` `  resultQ[i] = Quart(qArfFmt, result[i]);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		

Continued on next page

Table 3.62: *f16v2tof8* instruction definition (continued)

f16v2tof8			worker	aux
Syntax	`f16v2tof8 $a`*Dst0*`, $a`*Src0*			
Semantics	**Commit**	`// Values returned in both halves of the result` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  resultQ[0].bitz32() |` `  (resultQ[1].bitz32() << 8) |` `  (resultQ[0].bitz32() << 16) |` `  (resultQ[1].bitz32() << 24)` `};`		

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

Function references: *TFPU_ApplyStochasticRoundF8* , *TFPU_GenOFLOCheckF8* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

3.7.3.1.5 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 3.63: *f16v4sufromui* instruction definition

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

Architectural state references: *$FP_CTL*

Function references: *TFPU_GetNanooMode*

3.7.3.1.6 f16v8tof8

Half-precision 8-element vector to *quarter-precision* 8-element vector conversion.

Table 3.64: *f16v8tof8* instruction definition

f16v8tof8			worker	aux
Syntax	`f16v8tof8 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<Half,8> op1 = {` `  pickHalf($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)` `};` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `array<Quart,8> resultQ;` `vector<Single> result = {` `  op1[0], op1[1], op1[2], op1[3],` `  op1[4], op1[5], op1[6], op1[7]` `};`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `}` `for (i = 0; i < 8; i++) {` `  if (op1[i].issNaN()) {` `  fpExcpt |= TFPEXCPT_INV;` `  } else if (op1[i].isqNaN()) {` `  // No exception` `  } else if (op1[i].isInf()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}`		
	Compute	`Quart q = Quart(qArfFmt);` `int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `for (i = 0; i < 8; i++) {` `  if (!op1[i].isNaN() && !op1[i].isInf()) {` `  result[i] = result[i] * exp2f(scale);` `  }` `}` `if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundF8(randomBits, result, q.expSize(), q.bias());` `}` `for (i = 0; i < 8; i++) {` `  fpExcpt |= TFPU_GenOFLOCheckF8(` `  result[i], q.mantSize(), q.maxValue(), nanoo);` `}` `for (i = 0; i < 8; i++) {` `  if (op1[i].isNaN() || op1[i].isInf()) {` `  resultQ[i] = Quart(qArfFmt, QUART_ERROR);`		

Continued on next page

Table 3.64: *f16v8tof8* instruction definition (continued)

f16v8tof8			worker	aux
Syntax	`f16v8tof8 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+3*			
Semantics	**Compute cont'd**	`  } else {` `    resultQ[i] = Quart(qArfFmt, result[i]);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0:Dst0+1 = {` `  resultQ[0].bitz32() |` `  (resultQ[1].bitz32() << 8) |` `  (resultQ[2].bitz32() << 16) |` `  (resultQ[3].bitz32() << 24),` `  resultQ[4].bitz32() |` `  (resultQ[5].bitz32() << 8) |` `  (resultQ[6].bitz32() << 16) |` `  (resultQ[7].bitz32() << 24)` `};`		

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

Function references: *TFPU_ApplyStochasticRoundF8* , *TFPU_GenOFLOCheckF8* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

3.7.3.1.7 f32fromi32

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

Table 3.65: *f32fromi32* instruction definition

f32fromi32			worker	aux
Syntax	`f32fromi32 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`SignedDataWord op1 = $aSrc0;`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `Double accurate = op1;` `Single result = TFPU_RoundFP64ToFmt(accurate, TFPU_FP32, rmode);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

Architectural state references: *$FP_CTL*

Function references: *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

3.7.3.1.8 f32fromui32

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

Table 3.66: *f32fromui32* instruction definition

f32fromui32			worker	aux
Syntax	`f32fromui32 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`DataWord op1 = $aSrc0;`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `Double accurate = op1;` `Single result = TFPU_RoundFP64ToFmt(accurate, TFPU_FP32, rmode);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

Architectural state references: *$FP_CTL*

Function references: *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

f32fromui32 occurs in the following code examples:

- *ldb16b16 example*

3.7.3.1.9 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 3.67: *f32sufromui* instruction definition

f32sufromui			worker	aux
Syntax	`f32sufromui $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`DataWord op1 = $aSrc0;`		
	Compute	`// Obtain unsigned 32-bit input value` `uint64_t uival = op1;` `// Double the magnitude of the input value` `// and shift the result to produce a symmetric` `// distribution centred on 0 (resulting range here is [-2^32-1, 2^32-1])` `int64_t valp = (2 * uival) - ((1ULL << 32) - 1);` `// Scale the result so that the resulting output range is [-0.5, 0.5]` `Double res = valp / exp2(33);` `// Round to nearest single-precision value, ties to even` `Single result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

Function references: *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

3.7.3.1.10 f32tof16

Convert a *single-precision* value to *f16*, using the rounding mode as specified by $FP_CTL.RND/$FP_CTL.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 *f16* values.

Table 3.68: *f32tof16* instruction definition

f32tof16			worker	aux
Syntax	`f32tof16 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `vector<Single> result = { op1 };`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `// Floating-point exception check` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `  TFPU_ApplyF16StochasticRound(randomBits, result, fp16Fmt);` `}` `if (TFPU_F32_IsSNan(op1)) {` `  fpExcpt = TFPEXCPT_INV;` `} else if (TFPU_F32_IsQNan(op1)) {` `  // No exception` `} else if (isinf(op1)) {` `  fpExcpt = TFPEXCPT_INV;` `} else {` `  fpExcpt = TFPU_GenOFLOCheck(result[0], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (TFPU_F32_IsSNan(op1) || TFPU_F32_IsQNan(op1) || isinf(op1)) {` `  result[0] = TFPU_F32_QNan();` `} else {` `  result[0] = result[0];` `}`		
	Commit	`// Value returned in both halves of the result` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[0]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_ApplyF16StochasticRound* , *TFPU_F32_IsSNan* , *TFPU_F32_IsQNan* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

3.7.3.1.11 f32toi32

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

Table 3.69: *f32toi32* instruction definition

f32toi32			worker	aux
Syntax	`f32toi32 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `SignedDataWord result;`		
	Except In	`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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (isnan(op1) || isinf(op1)) {` `  // IEEE 754-2008: 5.8 (result is implementation defined)` `  result = INT32_MIN;` `} else if (fabs(op1) == 0.0) {` `  // -0.0 -> 0` `  result = 0;` `} else {` `  // Source operand is not a NaN, infinity, or zero.` `  if ((static_cast<int64_t>(rounded) > INT32_MAX) ||` `  (static_cast<int64_t>(rounded) < INT32_MIN) ) {` `  // IEEE 754-2008: 5.8 (result is implementation defined)` `  result = INT32_MIN;` `  } else {` `  // Source operand is not a NaN, infinity or zero and is within` `  // uint32 range.` `  result = static_cast<int32_t>(rounded);` `  }` `}`		
	Commit	`$aDst0 = result;`		

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

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

3.7.3.1.12 f32toui32

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

Table 3.70: *f32toui32* instruction definition

f32toui32			worker	aux
Syntax	f32toui32 $a*Dst0*, $a*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `DataWord result;`		
	Except In	`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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (isnan(op1) || isinf(op1)) {` `  // IEEE 754-2008: 5.8 (result is implementation defined)` `  result = 0;` `} else if (fabs(op1) == 0.0) {` `  // -0.0 -> 0` `  result = 0;` `} else {` `  // Source operand is not a NaN, infinity, or zero.` `  if ((static_cast<int64_t>(rounded) > UINT32_MAX) ||` `  (static_cast<int64_t>(rounded) < 0) ) {` `  // IEEE 754-2008: 5.8 (result is implementation defined)` `  result = 0;` `  } else {` `  // Source operand is not a NaN, infinity or zero and is within` `  // uint32 range.` `  result = static_cast<uint32_t>(rounded);` `  }` `}`		
	Commit	`$aDst0 = result;`		

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

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

3.7.3.1.13 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 3.71: *f32v2sufromui* instruction definition

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

Function references: *TFPU_RoundFP64ToFmt* , *TFPU_BitsFromF32*

3.7.3.1.14 f32v2tof16

Single-precision floating-point pair to *f16* conversion

Table 3.72: *f32v2tof16* instruction definition

f32v2tof16			worker	aux
Syntax	`f32v2tof16 $a`*Dst0*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($aSrc0:Src0+1[0]),` `  TFPU_F32FromBits($aSrc0:Src0+1[1])` `};` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `vector<Single> result = { op1[0], op1[1] };`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `// Floating-point exception check` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `  TFPU_ApplyF16StochasticRound(randomBits, result, fp16Fmt);` `}` `for (i = 0; i < 2; i++) {` `  if (TFPU_F32_IsSNan(op1[i])) {` `  fpExcpt \|= TFPEXCPT_INV;` `  } else if (TFPU_F32_IsQNan(op1[i])) {` `  // No exception` `  } else if (isinf(op1[i])) {` `  fpExcpt \|= TFPEXCPT_INV;` `  } else {` `  fpExcpt \|= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  if (TFPU_F32_IsSNan(op1[i]) || TFPU_F32_IsQNan(op1[i]) || isinf(op1[i])) {` `  result[i] = TFPU_F32_QNan();` `  } else {` `  result[i] = result[i];` `  }` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_ApplyF16StochasticRound* , *TFPU_F32_IsSNan* , *TFPU_F32_IsQNan* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

3.7.3.1.15 f32v4tof16

Single-precision floating-point 4-element vector to *f16* 4-element vector conversion.

Table 3.73: *f32v4tof16* instruction definition

f32v4tof16			worker	aux
Syntax	`f32v4tof16 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Single,4> op1 = {` `  TFPU_F32FromBits($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;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `vector<Single> result = { op1[0], op1[1], op1[2], op1[3] };`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `// Floating-point exception check` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `  TFPU_ApplyF16StochasticRound(randomBits, result, fp16Fmt);` `}` `for (i = 0; i < 4; i++) {` `  if (TFPU_F32_IsSNan(op1[i])) {` `  fpExcpt \|= TFPEXCPT_INV;` `  } else if (TFPU_F32_IsQNan(op1[i])) {` `  // No exception` `  } else if (isinf(op1[i])) {` `  fpExcpt \|= TFPEXCPT_INV;` `  } else {` `  fpExcpt \|= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 4; i++) {` `  if (TFPU_F32_IsSNan(op1[i]) || TFPU_F32_IsQNan(op1[i]) || isinf(op1[i])) {` `  result[i] = TFPU_F32_QNan();` `  } else {` `  result[i] = result[i];` `  }` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_ApplyF16StochasticRound* , *TFPU_F32_IsSNan* , *TFPU_F32_IsQNan* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

3.7.3.1.16 f8v2tof16

Quarter-precision floating-point vector to *half-precision* conversion

Table 3.74: *f8v2tof16* instruction definition

f8v2tof16			worker	aux
Syntax	`f8v2tof16 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<Quart,2> op1 = {` `  pickQuart($aSrc0[0], 0, qArfFmt),` `  pickQuart($aSrc0[0], 1, qArfFmt)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<Half,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (op1[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}`		
	Compute	`int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `for (i = 0; i < 2; i++) {` `  fpExcpt |= TFPU_GenOFLOCheck(op1[i] * exp2f(scale), TFPU_FP16, nanoo);` `}` `for (i = 0; i < 2; i++) {` `  if (op1[i].isError()) {` `  result[i] = Half(HALF_GCNAN);` `  } else {` `  result[i] = Half(op1[i] * exp2f(scale));` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

Architectural state references: *$FP_NFMT* , *$FP_CTL* , *$FP_SCL* , *$FP_STS*

Function references: *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

3.7.3.1.17 f8v4tof16

Quarter-precision floating-point vector to *half-precision* conversion

Table 3.75: *f8v4tof16* instruction definition

f8v4tof16			worker	aux
Syntax	`f8v4tof16 $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<Quart,4> op1 = {` `  pickQuart($aSrc0[0], 0, qArfFmt),` `  pickQuart($aSrc0[0], 1, qArfFmt),` `  pickQuart($aSrc0[0], 2, qArfFmt),` `  pickQuart($aSrc0[0], 3, qArfFmt)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<Half,4> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; i++) {` `  if (op1[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}`		
	Compute	`int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `for (i = 0; i < 4; i++) {` `  fpExcpt |= TFPU_GenOFLOCheck(op1[i] * exp2f(scale), TFPU_FP16, nanoo);` `}` `for (i = 0; i < 4; i++) {` `  if (op1[i].isError()) {` `  result[i] = Half(HALF_GCNAN);` `  } else {` `  result[i] = Half(op1[i] * exp2f(scale));` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

Architectural state references: *$FP_NFMT* , *$FP_CTL* , *$FP_SCL* , *$FP_STS*

Function references: *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

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 3.76: *f16v2absadd* instruction definition

f16v2absadd			worker	aux
Syntax	`f16v2absadd $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($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 = $FP_CTL.ESR;` `vector<Single> result;` `array<bool,2> specialCase;` `array<uint64_t,2> randomBits;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `result.resize(2);` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1,` `  randomBits);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  op1[i] = fabs(op1[i]);` `  op2[i] = fabs(op2[i]);` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16,` `  TFPU_ROUND_EVEN);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 2; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.2.2 f16v2absmax

Half-precision floating-point vector max of absolute values

Table 3.77: *f16v2absmax* instruction definition

f16v2absmax			worker	aux
Syntax	`f16v2absmax $a`*Dst0*`, $a`*Src1*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<Half,2> z;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`z[0] = TFPU_Max(fabs(op1[0]), fabs(op2[0]));` `z[1] = TFPU_Max(fabs(op1[1]), fabs(op2[1]));`		
	Commit	`$aDst0 = {` `  Half(z[0]).bitz32(PROP_INF) |` `  (Half(z[1]).bitz32(PROP_INF) << 16)` `};`		

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 3.78: *f16v2add* instruction definition

f16v2add			worker	aux
Syntax	`f16v2add $a`*Dst0*`, $a`*Src0*`:BL, $a`*Src1* `f16v2add $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `f16v2add $a`*Dst0*`, $a`*Src0*`:BU, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($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 = $FP_CTL.ESR;` `vector<Single> result;` `array<bool,2> specialCase;` `array<uint64_t,2> randomBits;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `result.resize(2);` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1,` `  randomBits);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16,` `  TFPU_ROUND_EVEN);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 2; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.2.4 f16v2clamp

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

Table 3.79: *f16v2clamp* instruction definition

f16v2clamp			worker	aux
Syntax	`f16v2clamp $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<Half,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Half lower = (float)(op2[0]);` `Half upper = (float)(op2[1]);` `// Unlike min/max, clamp explicitly propagates (regenerates)` `// NaN inputs` `if (lower.isNaN() || upper.isNaN()) {` `  for (i = 0; i < 2; i++) {` `  result[i] = TFPU_F32_QNan();` `  }` `} else {` `  for (i = 0; i < 2; i++) {` `  if (op1[i].isNaN()) {` `  result[i] = TFPU_F32_QNan();` `  } else if (op1[i] > upper) {` `  result[i] = upper;` `  } else {` `  result[i] = TFPU_Max(op1[i], lower);` `  }` `  }` `}`		
	Commit	`$aDst0 = {` `  Half(result[0]).bitz32(PROP_INF) |` `  (Half(result[1]).bitz32(PROP_INF) << 16)` `};`		

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 3.80: *f16v2class* instruction definition

f16v2class			worker	aux
Syntax	`f16v2class $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};`		
	Compute	`op0 = { 0, 0, 0, 0 };` `for (i = 0; i < 2; i++) {` `  DataWord clss;` `  bool sign = op1[i].sign();` `  if (op1[i].issNaN()) {` `  clss = TFPU_CLASS_SNAN;` `  } else if (op1[i].isqNaN()) {` `  clss = TFPU_CLASS_QNAN;` `  } else if (op1[i].isInf()) {` `  clss = ( sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF );` `  } else if (op1[i].isZero()) {` `  clss = ( sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO );` `  } else if (op1[i].isDenorm()) {` `  clss = ( sign ? TFPU_CLASS_NEG_DENORM : TFPU_CLASS_POS_DENORM );` `  } else {` `  clss = ( sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM );` `  }` `  op0[i] = clss;` `}`		
	Commit	`$aDst0 = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

3.7.3.2.6 f16v2cmac

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

Table 3.81: *f16v2cmac* instruction definition

f16v2cmac			worker	aux
Syntax	`f16v2cmac $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op0 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `array<Half,2> op1 = {` `  pickHalf($aSrc1[0], 0),` `  pickHalf($aSrc1[0], 1)` `};` `Single result;`		
	Except In	`bool specialCase = false;` `uint32_t fpExcpt = TFPEXCPT_NONE;` `if (op0[0].issNaN() || op1[0].issNaN() ||` `  op0[1].issNaN() || op1[1].issNaN()) {` `  specialCase = true;` `  result = TFPU_F32_QNan();` `  fpExcpt = TFPEXCPT_INV;` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (!specialCase) {` `  vector<Single> xv = { op0[0], op0[1] };` `  vector<Single> yv = { op1[0], op1[1] };` `  result = TFPU_F16DotProduct(xv, yv, 0);` `  result = TFPU_Add($AACC[0], result, TFPU_FP32);` `}`		
	Commit	`$AACC[0] = result;`		

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 3.82: *f16v2cmpeq* instruction definition

f16v2cmpeq			worker	aux
Syntax	f16v2cmpeq $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2cmpeq $a*Dst0*, $a*Src0*, $a*Src1* f16v2cmpeq $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	array<Half,2> op1 = { pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF), pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF) }; array<Half,2> op2 = { pickHalf($aSrc1[0], 0, PROP_INF), pickHalf($aSrc1[0], 1, PROP_INF) }; array<TileFPRelation_t,2> rl; array<bool,2> result; for (i = 0; i < 2; ++i) { rl[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 (rl[i] == TFPU_RELATION_UN) { fpExcpt |= TFPEXCPT_INV; } } $FP_STS = $FP_STS | fpExcpt; if (TFPU_IsMalign(fpExcpt, $FP_CTL)) { EXCEPT(TEXCPT_FP); }		
	Compute	result[0] = (rl[0] == TFPU_RELATION_EQ); result[1] = (rl[1] == TFPU_RELATION_EQ);		
	Commit	$aDst0 = { (result[0] ? 0xffff : 0x0000) | ((result[1] ? 0xffff : 0x0000) << 16) };		

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 3.83: *f16v2cmpge* instruction definition

f16v2cmpge			worker	aux
Syntax	f16v2cmpge $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2cmpge $a*Dst0*, $a*Src0*, $a*Src1* f16v2cmpge $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF),` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; ++i) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`result[0] = ((rl[0] == TFPU_RELATION_EQ) || (rl[0] == TFPU_RELATION_GT));` `result[1] = ((rl[1] == TFPU_RELATION_EQ) || (rl[1] == TFPU_RELATION_GT));`		
	Commit	`$aDst0 = {` `  (result[0] ? 0xffff : 0x0000) |` `  ((result[1] ? 0xffff : 0x0000) << 16)` `};`		

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 3.84: *f16v2cmpgt* instruction definition

f16v2cmpgt			worker	aux
Syntax	f16v2cmpgt $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2cmpgt $a*Dst0*, $a*Src0*, $a*Src1* f16v2cmpgt $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	array<Half,2> op1 = { pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF), pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF) }; array<Half,2> op2 = { pickHalf($aSrc1[0], 0, PROP_INF), pickHalf($aSrc1[0], 1, PROP_INF) }; array<TileFPRelation_t,2> rl; array<bool,2> result; for (i = 0; i < 2; ++i) { rl[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 (rl[i] == TFPU_RELATION_UN) { fpExcpt |= TFPEXCPT_INV; } } $FP_STS = $FP_STS | fpExcpt; if (TFPU_IsMalign(fpExcpt, $FP_CTL)) { EXCEPT(TEXCPT_FP); }		
	Compute	result[0] = (rl[0] == TFPU_RELATION_GT); result[1] = (rl[1] == TFPU_RELATION_GT);		
	Commit	$aDst0 = { (result[0] ? 0xffff : 0x0000) | ((result[1] ? 0xffff : 0x0000) << 16) };		

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 3.85: *f16v2cmple* instruction definition

f16v2cmple			worker	aux
Syntax	f16v2cmple $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2cmple $a*Dst0*, $a*Src0*, $a*Src1* f16v2cmple $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	array<Half,2> op1 = { pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF), pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF) }; array<Half,2> op2 = { pickHalf($aSrc1[0], 0, PROP_INF), pickHalf($aSrc1[0], 1, PROP_INF) }; array<TileFPRelation_t,2> rl; array<bool,2> result; for (i = 0; i < 2; ++i) { rl[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 (rl[i] == TFPU_RELATION_UN) { fpExcpt |= TFPEXCPT_INV; } } $FP_STS = $FP_STS | fpExcpt; if (TFPU_IsMalign(fpExcpt, $FP_CTL)) { EXCEPT(TEXCPT_FP); }		
	Compute	result[0] = ((rl[0] == TFPU_RELATION_EQ) || (rl[0] == TFPU_RELATION_LT)); result[1] = ((rl[1] == TFPU_RELATION_EQ) || (rl[1] == TFPU_RELATION_LT));		
	Commit	$aDst0 = { (result[0] ? 0xffff : 0x0000) | ((result[1] ? 0xffff : 0x0000) << 16) };		

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 3.86: *f16v2cmplt* instruction definition

f16v2cmplt			worker	aux
Syntax	f16v2cmplt $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2cmplt $a*Dst0*, $a*Src0*, $a*Src1* f16v2cmplt $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF),` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; ++i) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`result[0] = (rl[0] == TFPU_RELATION_LT);` `result[1] = (rl[1] == TFPU_RELATION_LT);`		
	Commit	`$aDst0 = {` `  (result[0] ? 0xffff : 0x0000) |` `  ((result[1] ? 0xffff : 0x0000) << 16)` `};`		

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 3.87: *f16v2cmpne* instruction definition

f16v2cmpne			worker	aux
Syntax	`f16v2cmpne $a`*Dst0*`, $a`*Src0*`:BL, $a`*Src1* `f16v2cmpne $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `f16v2cmpne $a`*Dst0*`, $a`*Src0*`:BU, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF),` `  pickHalf(<($aSrc0, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; ++i) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`result[0] = (rl[0] != TFPU_RELATION_EQ);` `result[1] = (rl[1] != TFPU_RELATION_EQ);`		
	Commit	`$aDst0 = {` `  (result[0] ? 0xffff : 0x0000) |` `  ((result[1] ? 0xffff : 0x0000) << 16)` `};`		

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

Function references: *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.2.13 f16v2exp

Table 3.88: *f16v2exp* instruction definition

f16v2exp			worker	aux
Syntax	`f16v2exp $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoExpPreExecute(op1[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  result[i] = TFPU_F16Exp(op1[i], TFPU_BASE_E, rmode);` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

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

3.7.3.2.14 f16v2exp2

Table 3.89: *f16v2exp2* instruction definition

f16v2exp2			worker	aux
Syntax	`f16v2exp2 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoExpPreExecute(op1[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  result[i] = TFPU_F16Exp(op1[i], TFPU_BASE_2, rmode);` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

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

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:

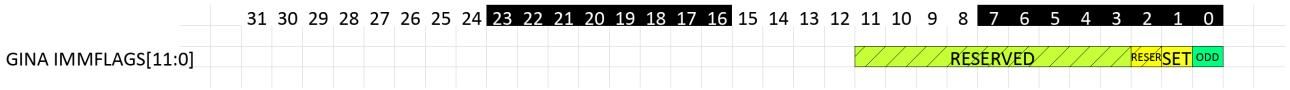


Fig. 3.11: f16v2gina immediate format

Table 3.90: *f16v2gina* instruction definition

f16v2gina			worker	aux
Syntax	`f16v2gina $a`*Dst0*`, $a`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `DataWord op2 = zimm12;` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `array<uint64_t,2> randomBits;` `vector<Single> result;` `unsigned set = GINA_IMMFLAGS__SET__GET(op2);` `// base accumulator ID for set` `unsigned b = set * TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `// Input exception check` `if (op1[0].issNaN() || op1[1].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `}`		
	Compute	`if (GINA_IMMFLAGS__ODD__GET(op2) == 0) {` `  result = { $AACC[b+0], $AACC[b+2] };` `} else {` `  result = { $AACC[b+1], $AACC[b+3] };` `}` `if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(result, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`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];`		

Continued on next page

Table 3.90: *f16v2gina* instruction definition (continued)

f16v2gina			worker	aux
Syntax	`f16v2gina $a`*Dst0*`, $a`*Src0*`,` *zimm12*			
Semantics	**Commit cont'd**	

```
    // 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];

    // 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 = TFPU_GetNanooMode(nanoo);
  $aDst0 = {
    Half(result[0]).bitz32(smode) |
    (Half(result[1]).bitz32(smode) << 16)
  };
```

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

Function references: *TFPU_ApplyStochasticRoundHalf* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_F32_QuietenNan* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

3.7.3.2.16 f16v2grand

Gaussian distribution, 2-element *half-precision* random vector

Table 3.91: *f16v2grand* instruction definition

f16v2grand			worker	aux
Syntax	`f16v2grand $a`*Dst0*			
Semantics	**Prepare**	`bool nanoo = $FP_CTL.NANOO;` `array<Single,2> result;`		
	Compute	`array<uint64_t,2> randomBits;` `TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `// Calculate overall summation based on outputs from successive applications` `// of xoroshiro128aox` `int16_t rsum[2] = { 0, 0 };` `for (i = 0; i < 12; i++) {` `  rsum[0] += ((randomBits[0] >> (i * 5)) & 0x1f);` `  rsum[1] += ((randomBits[1] >> (i * 5)) & 0x1f);` `}` `result[0] = (rsum[0] - 186) / 32.0;` `result[1] = (rsum[1] - 186) / 32.0;`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_GetNanooMode*

3.7.3.2.17 f16v2ln

Table 3.92: *f16v2ln* instruction definition

f16v2ln			worker	aux
Syntax	`f16v2ln $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoLogPreExecute(op1[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  result[i] = TFPU_F16Log(op1[i], TFPU_BASE_E, rmode);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

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

3.7.3.2.18 f16v2log2

Table 3.93: *f16v2log2* instruction definition

f16v2log2			worker	aux
Syntax	`f16v2log2 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoLogPreExecute(op1[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  result[i] = TFPU_F16Log(op1[i], TFPU_BASE_2, rmode);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

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

3.7.3.2.19 f16v2max

Half-precision floating-point vector max

Table 3.94: *f16v2max* instruction definition

f16v2max			worker	aux
Syntax	`f16v2max $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<Half,2> z;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`z[0] = TFPU_Max(op1[0], op2[0]);` `z[1] = TFPU_Max(op1[1], op2[1]);`		
	Commit	`$aDst0 = {` `  Half(z[0]).bitz32(PROP_INF) |` `  (Half(z[1]).bitz32(PROP_INF) << 16)` `};`		

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 3.95: *f16v2maxc* instruction definition

f16v2maxc			worker	aux
Syntax	`f16v2maxc $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (op1[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Half max = TFPU_Max(op1[0], op1[1]);`		
	Commit	`$aDst0 = Half(max).bitz32(PROP_INF);`		

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 3.96: *f16v2min* instruction definition

f16v2min			worker	aux
Syntax	`f16v2min $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0, PROP_INF),` `  pickHalf($aSrc0[0], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<Half,2> z;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`z[0] = TFPU_Min(op1[0], op2[0]);` `z[1] = TFPU_Min(op1[1], op2[1]);`		
	Commit	`$aDst0 = {` `  Half(z[0]).bitz32(PROP_INF) |` `  (Half(z[1]).bitz32(PROP_INF) << 16)` `};`		

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 3.97: *f16v2mul* instruction definition

f16v2mul			worker	aux
Syntax	f16v2mul $a*Dst0*, $a*Src0*:BL, $a*Src1* f16v2mul $a*Dst0*, $a*Src0*, $a*Src1* f16v2mul $a*Dst0*, $a*Src0*:BU, $a*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($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 = $FP_CTL.ESR;` `TileRoundMode_t rmode;` `vector<Single> result;` `array<bool,2> specialCase;` `array<uint64_t,2> randomBits;` `if (enableStochasticRounding) {` `  rmode = TFPU_ROUND_EVEN;` `} else {` `  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, $PRNG_1_1, randomBits);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) *` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, rmode);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 2; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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_ApplyStochasticRoundHalf* ,

TFPU_GenOFLOCheck , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.2.23 f16v2sigm

Table 3.98: *f16v2sigm* instruction definition

f16v2sigm			worker	aux
Syntax	`f16v2sigm $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoSigmoidPreExecute(op1[i], &fpExcpt, &result[i]);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  result[i] = specialCase[i] ? result[i] : TFPU_F16Sigmoid(op1[i], rmode);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_DoSigmoidPreExecute* , *TFPU_IsMalign* , *TFPU_F16Sigmoid* , *TFPU_GetNanooMode*

3.7.3.2.24 f16v2sub

Half-precision floating-point vector subtract

Table 3.99: *f16v2sub* instruction definition

f16v2sub			worker	aux
Syntax	`f16v2sub $a`*Dst0*`, $a`*Src0*`:BL, $a`*Src1* `f16v2sub $a`*Dst0*`, $a`*Src0*`, $a`*Src1* `f16v2sub $a`*Dst0*`, $a`*Src0*`:BU, $a`*Src1*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf(<($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 = $FP_CTL.ESR;` `vector<Single> result;` `array<bool,2> specialCase;` `array<uint64_t,2> randomBits;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `result.resize(2);` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1,` `  randomBits);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) -` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16,` `  TFPU_ROUND_EVEN);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 2; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.2.25 f16v2sum

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

Table 3.100: *f16v2sum* instruction definition

f16v2sum			worker	aux
Syntax	`f16v2sum $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoAddPreExecute(op1[0], op1[1], &fpExcpt, &result);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`if (!specialCase) {` `  Double res = op1[0] + op1[1];` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `}`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

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

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

3.7.3.2.26 f16v2tanh

Table 3.101: *f16v2tanh* instruction definition

f16v2tanh			worker	aux
Syntax	`f16v2tanh $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`array<Half,2> op1 = {` `  pickHalf($aSrc0[0], 0),` `  pickHalf($aSrc0[0], 1)` `};` `bool nanoo = $FP_CTL.NANOO;` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoTanhPreExecute(op1[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  result[i] = TFPU_F16Tanh(op1[i], rmode);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(result[0]).bitz32(smode) |` `  (Half(result[1]).bitz32(smode) << 16)` `};`		

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

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

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 3.102: *f16v4absacc* instruction definition

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

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 3.103: *f16v4absadd* instruction definition

f16v4absadd			worker	aux
Syntax	`f16v4absadd $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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)` `};` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `vector<Single> result;` `array<bool,4> specialCase;` `array<uint64_t,2> randomBits;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `result.resize(4);` `for (i = 0; i < 4; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1,` `  randomBits);` `}`		
	Compute	`for (i = 0; i < 4; ++i) {` `  if (!specialCase[i]) {` `  op1[i] = fabs(op1[i]);` `  op2[i] = fabs(op2[i]);` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16,` `  TFPU_ROUND_EVEN);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 4; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.3.3 f16v4absmax

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

Table 3.104: *f16v4absmax* instruction definition

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

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 3.105: *f16v4acc* instruction definition

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

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 3.106: *f16v4add* instruction definition

f16v4add			worker	aux
Syntax	f16v4add $a*Dst0*:*Dst0+1*, $a*Src0*:BL, $a*Src1*:*Src1+1* f16v4add $a*Dst0*:*Dst0+1*, $a*Src0*:*Src0+1*, $a*Src1*:*Src1+1* f16v4add $a*Dst0*:*Dst0+1*, $a*Src0*:BU, $a*Src1*:*Src1+1*			
Semantics	**Prepare**	``` array<Half,4> op1 = { pickHalf(<($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 = $FP_CTL.ESR; vector<Single> result; array<bool,4> specialCase; array<uint64_t,2> randomBits; ```		
	Except In	``` uint32_t fpExcpt = TFPEXCPT_NONE; result.resize(4); for (i = 0; i < 4; ++i) { specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt, &result[i]); } if (enableStochasticRounding) { TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits); } ```		
	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, TFPU_ROUND_EVEN); } } ```		
	Except Out	``` if (enableStochasticRounding) { TFPU_ApplyStochasticRoundHalf(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); } ```		
	Commit	``` HalfSaturationMode smode = TFPU_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) }; ```		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.3.6 f16v4clamp

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

Table 3.107: *f16v4clamp* instruction definition

f16v4clamp			worker	aux
Syntax	`f16v4clamp $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[0], 1, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 1, PROP_INF)` `};` `array<Half,2> op2 = {` `  pickHalf($aSrc1[0], 0, PROP_INF),` `  pickHalf($aSrc1[0], 1, PROP_INF)` `};` `array<Half,4> result;`		
	Except In	`Single ops[6] = { op1[0], op1[1], op1[2], op1[3], op2[0], op2[1] };` `uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 6);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Half lower = (float)(op2[0]);` `Half upper = (float)(op2[1]);` `// Unlike min/max, clamp explicitly propagates (regenerates)` `// NaN inputs` `if (lower.isNaN() || upper.isNaN()) {` `  for (i = 0; i < 4; i++) {` `  result[i] = TFPU_F32_QNan();` `  }` `} else {` `  for (i = 0; i < 4; i++) {` `  if (op1[i].isNaN()) {` `  result[i] = TFPU_F32_QNan();` `  } else if (op1[i] > upper) {` `  result[i] = upper;` `  } else {` `  result[i] = TFPU_Max(op1[i], lower);` `  }` `  }` `}`		
	Commit	`$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)` `};`		

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 3.108: *f16v4class* instruction definition

f16v4class			worker	aux
Syntax	`f16v4class $a`*Dst0*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[0], 1, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 1, PROP_INF)` `};`		
	Compute	`for (i = 0; i < 4; i++) {` `  DataWord clss;` `  bool sign = op1[i].sign();` `  if (op1[i].issNaN()) {` `  clss = TFPU_CLASS_SNAN;` `  } else if (op1[i].isqNaN()) {` `  clss = TFPU_CLASS_QNAN;` `  } else if (op1[i].isInf()) {` `  clss = ( sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF );` `  } else if (op1[i].isZero()) {` `  clss = ( sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO );` `  } else if (op1[i].isDenorm()) {` `  clss = ( sign ? TFPU_CLASS_NEG_DENORM : TFPU_CLASS_POS_DENORM );` `  } else {` `  clss = ( sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM );` `  }` `  op0[i] = clss;` `}`		
	Commit	`$aDst0 = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

3.7.3.3.8 f16v4cmac

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

Table 3.109: *f16v4cmac* instruction definition

f16v4cmac			worker	aux
Syntax	f16v4cmac $a*Src0*:*Src0+1*, $a*Src1*:*Src1+1*			
Semantics	**Prepare**	array<Half,4> op0 = { pickHalf($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;		
	Except In	array<bool,2> specialCase = { false, false }; uint32_t fpExcpt = TFPEXCPT_NONE; for (i = 0; i < 4; i+=2) { if (op0[i].issNaN() || op1[i].issNaN() || op0[i+1].issNaN() || op1[i+1].issNaN()) { 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); }		
	Compute	TileRoundMode_t rmode = $FP_CTL.RND; for (int i = 0; i < 4; i+=2) { if (!specialCase[i/2]) { vector<Single> xv = { op0[i], op0[i+1] }; vector<Single> yv = { op1[i], op1[i+1] }; result[i/2] = TFPU_F16DotProduct(xv, yv, 0); result[i/2] = TFPU_Add($AACC[i], result[i/2], TFPU_FP32) ; } }		
	Commit	$AACC[0] = result[0]; $AACC[2] = result[1];		

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        $triPtr, $inPtr, $weightPtr , $mzero

  ld2x64pace    $inputs, $weights, $triPtr+=, $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, $triPtr+=, $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, $azeros, 0
  f32add        $a0, $a0, $a1
```

3.7.3.3.9 f16v4cmpeq

Half-precision 4-element vector equality test

Table 3.110: *f16v4cmpeq* instruction definition

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

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 3.111: *f16v4cmpge* instruction definition

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

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 3.112: *f16v4cmpgt* instruction definition

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

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 3.113: *f16v4cmple* instruction definition

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

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 3.114: *f16v4cmplt* instruction definition

f16v4cmplt			worker	aux
Syntax	`f16v4cmplt $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BL, $a`*Src1*`:`*Src1+1* `f16v4cmplt $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f16v4cmplt $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BU, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[1], 0, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[1], 1, PROP_INF)` `};` `array<Half,4> op2 = {` `  pickHalf($aSrc1:Src1+1[0], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[0], 1, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 1, PROP_INF)` `};` `array<TileFPRelation_t,4> rl;` `array<bool,4> result;` `for (i = 0; i < 4; ++i) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`result[0] = (rl[0] == TFPU_RELATION_LT);` `result[1] = (rl[1] == TFPU_RELATION_LT);` `result[2] = (rl[2] == TFPU_RELATION_LT);` `result[3] = (rl[3] == TFPU_RELATION_LT);`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffff : 0x0000) |` `  ((result[1] ? 0xffff : 0x0000) << 16),` `  (result[2] ? 0xffff : 0x0000) |` `  ((result[3] ? 0xffff : 0x0000) << 16)` `};`		

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 3.115: *f16v4cmpne* instruction definition

f16v4cmpne			worker	aux
Syntax	`f16v4cmpne $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BL, $a`*Src1*`:`*Src1+1* `f16v4cmpne $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f16v4cmpne $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BU, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[0], 0, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[0], 1, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[1], 0, PROP_INF),` `  pickHalf(<($aSrc0:Src0+1, $aSrc0:BL, $aSrc0:BU)>[1], 1, PROP_INF)` `};` `array<Half,4> op2 = {` `  pickHalf($aSrc1:Src1+1[0], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[0], 1, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 1, PROP_INF)` `};` `array<TileFPRelation_t,4> rl;` `array<bool,4> result;` `for (i = 0; i < 4; ++i) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`result[0] = (rl[0] != TFPU_RELATION_EQ);` `result[1] = (rl[1] != TFPU_RELATION_EQ);` `result[2] = (rl[2] != TFPU_RELATION_EQ);` `result[3] = (rl[3] != TFPU_RELATION_EQ);`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffff : 0x0000) |` `  ((result[1] ? 0xffff : 0x0000) << 16),` `  (result[2] ? 0xffff : 0x0000) |` `  ((result[3] ? 0xffff : 0x0000) << 16)` `};`		

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 3.116: *f16v4gacc* instruction definition

f16v4gacc			worker	aux
Syntax	`f16v4gacc $a`*Dst0*`:`*Dst0+1*			
Semantics	**Prepare**	`bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `vector<Single> result = { $AACC[0],` `$AACC[2],` `$AACC[4],` `$AACC[6] };`		
	Compute	`// Output overflow check` `if (enableStochasticRounding) {` `array<uint64_t,2> randomBits;` `TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}`		
	Except **Out**	`// Floating-point result exception check` `uint32_t fpExcpt = TFPU_AACCReadFlags(result, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `EXCEPT(TEXCPT_FP);` `}`		
	Commit	`for (i = 0; i < 4; i++) {` `if (isinf(result[i]) || isnan(result[i])) {` `result[i] = TFPU_F32_QNan();` `}` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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

Function references: *TFPU_ApplyStochasticRoundHalf* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

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 3.117: f16v4hihoamp 8x1x1x16 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[12]	$AACC[10]	$AACC[8]	$AACC[6]	$AACC[4]	$AACC[2]	$AACC[0]	$a*Dst0*
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	$R_7 = x_0.\mathbf{CW}_{7,0} + P_7$	$R_6 = x_0.\mathbf{CW}_{6,0} + P_6$	$R_5 = x_0.\mathbf{CW}_{5,0} + P_5$	$R_4 = x_0.\mathbf{CW}_{4,0} + P_4$	$R_3 = x_0.\mathbf{CW}_{3,0} + P_3$	$R_2 = x_0.\mathbf{CW}_{2,0} + P_2$	$R_1 = x_0.\mathbf{CW}_{1,0} + P_1$	$R_0 = x_0.\mathbf{CW}_{0,0} + P_0$	-
x_1 | P_{10},P_{11}	$R_7 += x_1.\mathbf{CW}_{7,1}$	$R_6 += x_1.\mathbf{CW}_{6,1}$	$R_5 += x_1.\mathbf{CW}_{5,1}$	$R_4 += x_1.\mathbf{CW}_{4,1}$	$R_3 += x_1.\mathbf{CW}_{3,1}$	$R_2 += x_1.\mathbf{CW}_{2,1}$	$R_1 += x_1.\mathbf{CW}_{1,1}$	$R_0 += x_1.\mathbf{CW}_{0,1}$	-
x_2 | P_{12},P_{13}	$R_7 += x_2.\mathbf{CW}_{7,2}$	$R_6 += x_2.\mathbf{CW}_{6,2}$	$R_5 += x_2.\mathbf{CW}_{5,2}$	$R_4 += x_2.\mathbf{CW}_{4,2}$	$R_3 += x_2.\mathbf{CW}_{3,2}$	$R_2 += x_2.\mathbf{CW}_{2,2}$	$R_1 += x_2.\mathbf{CW}_{1,2}$	$R_0 += x_2.\mathbf{CW}_{0,2}$	-
x_3 | P_{14},P_{15}	$R_7 += x_3.\mathbf{CW}_{7,3}$	$R_6 += x_3.\mathbf{CW}_{6,3}$	$R_5 += x_3.\mathbf{CW}_{5,3}$	$R_4 += x_3.\mathbf{CW}_{4,3}$	$R_3 += x_3.\mathbf{CW}_{3,3}$	$R_2 += x_3.\mathbf{CW}_{2,3}$	$R_1 += x_3.\mathbf{CW}_{1,3}$	$R_0 += x_3.\mathbf{CW}_{0,3}$	-
x_4 | P_{16},P_{17}	$R_{15} = x_4.\mathbf{CW}_{7,0} + P_{15}$	$R_{14} = x_4.\mathbf{CW}_{6,0} + P_{14}$	$R_{13} = x_4.\mathbf{CW}_{5,0} + P_{13}$	$R_{12} = x_4.\mathbf{CW}_{4,0} + P_{12}$	$R_{11} = x_4.\mathbf{CW}_{3,0} + P_{11}$	$R_{10} = x_4.\mathbf{CW}_{2,0} + P_{10}$	$R_9 = x_4.\mathbf{CW}_{1,0} + P_9$	$R_8 = x_4.\mathbf{CW}_{0,0} + P_8$	R_0,R_1
x_5 | P_{18},P_{19}	$R_{15} += x_5.\mathbf{CW}_{7,1}$	$R_{14} += x_5.\mathbf{CW}_{6,1}$	$R_{13} += x_5.\mathbf{CW}_{5,1}$	$R_{12} += x_5.\mathbf{CW}_{4,1}$	$R_{11} += x_5.\mathbf{CW}_{3,1}$	$R_{10} += x_5.\mathbf{CW}_{2,1}$	$R_9 += x_5.\mathbf{CW}_{1,1}$	$R_8 += x_5.\mathbf{CW}_{0,1}$	R_2,R_3
x_6 | P_{20},P_{21}	$R_{15} += x_6.\mathbf{CW}_{7,2}$	$R_{14} += x_6.\mathbf{CW}_{6,2}$	$R_{13} += x_6.\mathbf{CW}_{5,2}$	$R_{12} += x_6.\mathbf{CW}_{4,2}$	$R_{11} += x_6.\mathbf{CW}_{3,2}$	$R_{10} += x_6.\mathbf{CW}_{2,2}$	$R_9 += x_6.\mathbf{CW}_{1,2}$	$R_8 += x_6.\mathbf{CW}_{0,2}$	R_4,R_5
x_7 | P_{22},P_{23}	$R_{15} += x_7.\mathbf{CW}_{7,3}$	$R_{14} += x_7.\mathbf{CW}_{6,3}$	$R_{13} += x_7.\mathbf{CW}_{5,3}$	$R_{12} += x_7.\mathbf{CW}_{4,3}$	$R_{11} += x_7.\mathbf{CW}_{3,3}$	$R_{10} += x_7.\mathbf{CW}_{2,3}$	$R_9 += x_7.\mathbf{CW}_{1,3}$	$R_8 += x_7.\mathbf{CW}_{0,3}$	R_6,R_7
x_8 | P_{24},P_{25}	$R_{23} = x_8.\mathbf{CW}_{7,0} + P_{23}$	$R_{22} = x_8.\mathbf{CW}_{6,0} + P_{22}$	$R_{21} = x_8.\mathbf{CW}_{5,0} + P_{21}$	$R_{20} = x_8.\mathbf{CW}_{4,0} + P_{20}$	$R_{19} = x_8.\mathbf{CW}_{3,0} + P_{19}$	$R_{18} = x_8.\mathbf{CW}_{2,0} + P_{18}$	$R_{17} = x_8.\mathbf{CW}_{1,0} + P_{17}$	$R_{16} = x_8.\mathbf{CW}_{0,0} + P_{16}$	R_8,R_9
x_9 | P_{26},P_{27}	$R_{23} += x_9.\mathbf{CW}_{7,1}$	$R_{22} += x_9.\mathbf{CW}_{6,1}$	$R_{21} += x_9.\mathbf{CW}_{5,1}$	$R_{20} += x_9.\mathbf{CW}_{4,1}$	$R_{19} += x_9.\mathbf{CW}_{3,1}$	$R_{18} += x_9.\mathbf{CW}_{2,1}$	$R_{17} += x_9.\mathbf{CW}_{1,1}$	$R_{16} += x_9.\mathbf{CW}_{0,1}$	R_{10},R_{11}
x_{10} | P_{28},P_{29}	$R_{23} += x_{10}.\mathbf{CW}_{7,2}$	$R_{22} += x_{10}.\mathbf{CW}_{6,2}$	$R_{21} += x_{10}.\mathbf{CW}_{5,2}$	$R_{20} += x_{10}.\mathbf{CW}_{4,2}$	$R_{19} += x_{10}.\mathbf{CW}_{3,2}$	$R_{18} += x_{10}.\mathbf{CW}_{2,2}$	$R_{17} += x_{10}.\mathbf{CW}_{1,2}$	$R_{16} += x_{10}.\mathbf{CW}_{0,2}$	R_{12},R_{13}
x_{11} | P_{30},P_{31}	$R_{23} += x_{11}.\mathbf{CW}_{7,3}$	$R_{22} += x_{11}.\mathbf{CW}_{6,3}$	$R_{21} += x_{11}.\mathbf{CW}_{5,3}$	$R_{20} += x_{11}.\mathbf{CW}_{4,3}$	$R_{19} += x_{11}.\mathbf{CW}_{3,3}$	$R_{18} += x_{11}.\mathbf{CW}_{2,3}$	$R_{17} += x_{11}.\mathbf{CW}_{1,3}$	$R_{16} += x_{11}.\mathbf{CW}_{0,3}$	R_{14},R_{15}
x_{12} | P_{32},P_{33}	$R_{31} = x_{12}.\mathbf{CW}_{7,0} + P_{31}$	$R_{30} = x_{12}.\mathbf{CW}_{6,0} + P_{30}$	$R_{29} = x_{12}.\mathbf{CW}_{5,0} + P_{29}$	$R_{28} = x_{12}.\mathbf{CW}_{4,0} + P_{28}$	$R_{27} = x_{12}.\mathbf{CW}_{3,0} + P_{27}$	$R_{26} = x_{12}.\mathbf{CW}_{2,0} + P_{26}$	$R_{25} = x_{12}.\mathbf{CW}_{1,0} + P_{25}$	$R_{24} = x_{12}.\mathbf{CW}_{0,0} + P_{24}$	R_{16},R_{17}

P_n is *half-precision* input partial-sum *n*
x_n is an *f16v4* input vector
$\mathbf{CW}_{m,n}$ 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 3.118: *f16v4hihoamp* instruction definition

f16v4hihoamp			worker	aux
Syntax	`f16v4hihoamp $a`*Dst0*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`, `*enumFlags*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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)` `};` `DataWord op3 = enumFlags;` `const unsigned ampUnits = 8;` `bool nanoo = $FP_CTL.NANOO;` `array<uint64_t,2> randomBits;` `vector<Single> out;` `array<vector<Single>,ampUnits> weights;` `array<Single,ampUnits> resultEven;` `array<Single,ampUnits> 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] };`		
	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 (int k = 0; k < ampUnits; k++) {` `  if (engineEnable[k >> 1]) {` `  // Weight selection - all fp16` `  uint64_t fullCwei =` `  context.getCCCSState((k * TREG_CCCS_WEIGHT_GROUP_SIZE) + 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());` `  }` `}`		

Continued on next page

Table 3.118: *f16v4hihoamp* instruction definition (continued)

f16v4hihoamp			worker	aux
Syntax	f16v4hihoamp $a*Dst0*, $a*Src0*:*Src0+1*, $a*Src1*, *enumFlags*			
Semantics	**Compute**	`int scale = 0;` `for (int k = 0; k < ampUnits; k++) {` `  unsigned engine = k >> 1;` `  if (engineEnable[engine]) {` `  // 4-element dot-product` `  resultEven[k] = TFPU_F16DotProduct(weights[k], inputs, scale);` `  Single incomingp;` `  if (0 == phase) {` `  // Even accumulators -` `  //  combine incoming partial-sum (currently stored in our` `  //  odd-accumulator) with dot-product result` `  resultEven[k] =` `  TFPU_Add($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 = $AACC[(k + 2) * 2];` `  } else {` `  // Engines behind me are disabled (or I am the final` `  // engine in the chain). Use new partial-sum inputs` `  Half partialIn = (float)(op2[k & 1]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  } else { // phase != 0` `  // Even accumulators -` `  //  accumulate dot-product result` `  resultEven[k] = TFPU_Add($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 = (float)(op2[k & 1]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `}`		

Continued on next page

Table 3.118: *f16v4hihoamp* instruction definition (continued)

f16v4hihoamp			worker	aux
Syntax	`f16v4hihoamp $a`*Dst0*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`, `*enumFlags*			
Semantics	**Compute cont'd**	`    resultOdd[k] = incomingp;` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS	fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` ` EXCEPT(TEXCPT_FP);` `}`		
	Commit	`for (int k = 0; k < ampUnits; k++) {` `  unsigned engine = k >> 1;` `  if (engineEnable[engine]) {` `  $AACC[(k * 2)]     = resultEven[k];` `  $AACC[(k * 2) + 1] = resultOdd[k];` `  }` `}` `for (i = 0; i < 2; i++) {` `  if (isinf(out[i]) || isnan(out[i])) {` `  out[i] = TFPU_F32_QNan();` `  }` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(out[0]).bitz32(smode) |` `  (Half(out[1]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_GenSNanCheck* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

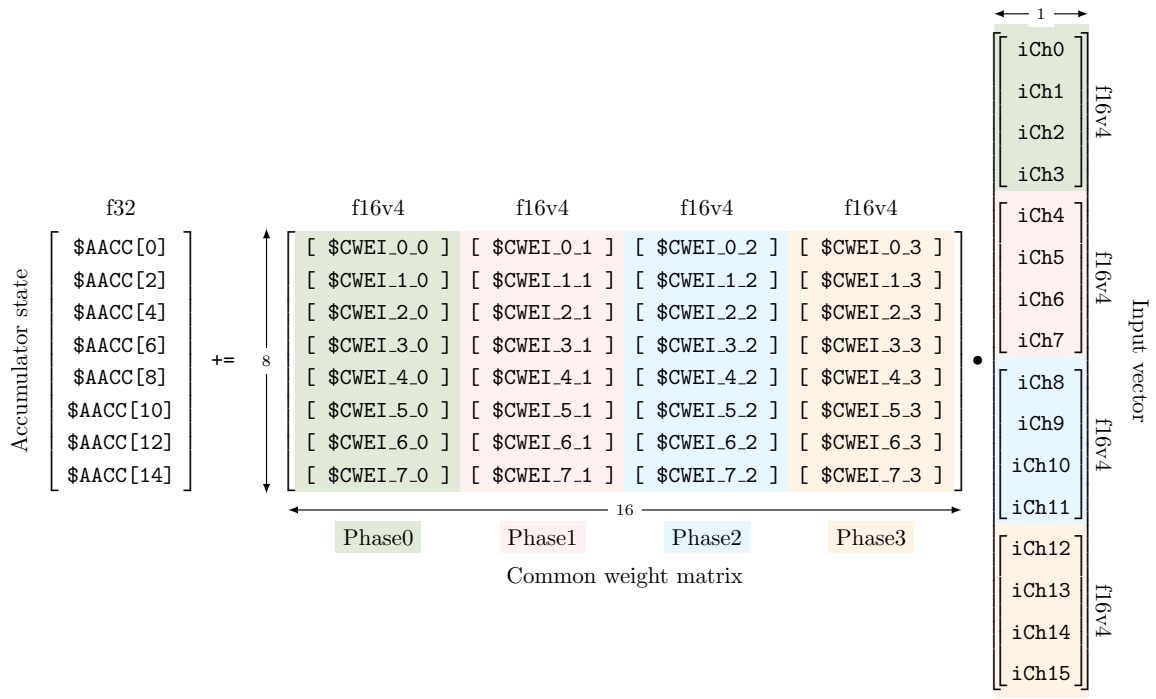


Fig. 3.12: f16v4hihoamp

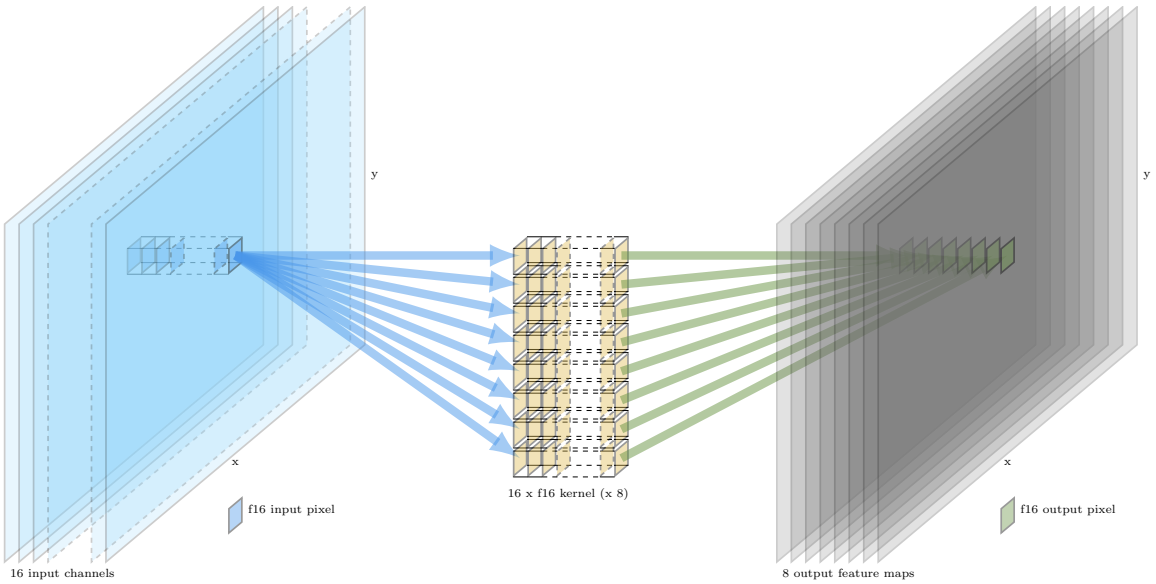


Fig. 3.13: f16v4hihoamp

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, $triPtr+=, $mzero, 0
    f16v4hihoamp  $outPartial0, $inData, $inPartial0, TAMP_F16V4_E4_P0
```

```
  }
  {
    ld2x64pace    $inData, $inPartials, $triPtr+=, $mzero, 0
    f16v4hihoamp  $outPartial1, $inData, $inPartial1, TAMP_F16V4_E4_P1
  }
  {
    ldst64pace    $inData, $outPartials, $triPtr+=, $mzero, 0
    f16v4hihoamp  $outPartial0, $inData, $inPartial0, TAMP_F16V4_E4_P2
  }
  {
    ld2x64pace    $inData, $inPartials, $triPtr+=, $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 3.119: f16v4hihoslic, 2x1x3x4 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[10]	$AACC[6]	$AACC[2]	$AACC[12]	$AACC[8]	$AACC[4]	$AACC[0]	$a*Dst0*
x_0|P_0,P_1	-	$R_1 = x_0.\mathbf{CW}_{5,0} + P_1$	-	-	-	$R_0 = x_0.\mathbf{CW}_{4,0} + P_0$	-	-	-
x_1|P_2,P_3	-	$R_3 = x_1.\mathbf{CW}_{5,0} + P_3$	$R_1 += x_1.\mathbf{CW}_{3,0}$	-	-	$R_2 = x_1.\mathbf{CW}_{4,0} + P_2$	$R_0 += x_1.\mathbf{CW}_{2,0}$	-	-
x_2|P_4,P_5	-	$R_5 = x_2.\mathbf{CW}_{5,0} + P_5$	$R_3 += x_2.\mathbf{CW}_{3,0}$	$R_1 += x_2.\mathbf{CW}_{1,0}$	-	$R_4 = x_2.\mathbf{CW}_{4,0} + P_4$	$R_2 += x_2.\mathbf{CW}_{2,0}$	$R_0 += x_2.\mathbf{CW}_{0,0}$	-
x_3|P_6,P_7	-	$R_7 = x_3.\mathbf{CW}_{5,0} + P_7$	$R_5 += x_3.\mathbf{CW}_{3,0}$	$R_3 += x_3.\mathbf{CW}_{1,0}$	-	$R_6 = x_3.\mathbf{CW}_{4,0} + P_6$	$R_4 += x_3.\mathbf{CW}_{2,0}$	$R_2 += x_3.\mathbf{CW}_{0,0}$	R_0,R_1
x_4|P_8,P_9	-	$R_9 = x_4.\mathbf{CW}_{5,0} + P_9$	$R_7 += x_4.\mathbf{CW}_{3,0}$	$R_5 += x_4.\mathbf{CW}_{1,0}$	-	$R_8 = x_4.\mathbf{CW}_{4,0} + P_8$	$R_6 += x_4.\mathbf{CW}_{2,0}$	$R_4 += x_4.\mathbf{CW}_{0,0}$	R_2,R_3
x_5|P_{10},P_{11}	-	$R_{11} = x_5.\mathbf{CW}_{5,0} + P_{11}$	$R_9 += x_5.\mathbf{CW}_{3,0}$	$R_7 += x_5.\mathbf{CW}_{1,0}$	-	$R_{10} = x_5.\mathbf{CW}_{4,0} + P_{10}$	$R_8 += x_5.\mathbf{CW}_{2,0}$	$R_6 += x_5.\mathbf{CW}_{0,0}$	R_4,R_5
x_6|P_{12},P_{13}	-	$R_{13} = x_6.\mathbf{CW}_{5,0} + P_{13}$	$R_{11} += x_6.\mathbf{CW}_{3,0}$	$R_9 += x_6.\mathbf{CW}_{1,0}$	-	$R_{12} = x_6.\mathbf{CW}_{4,0} + P_{12}$	$R_{10} += x_6.\mathbf{CW}_{2,0}$	$R_8 += x_6.\mathbf{CW}_{0,0}$	R_6,R_7

Table 3.120: f16v4hihoslic, 1x4 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[10]	$AACC[6]	$AACC[2]	$AACC[12]	$AACC[8]	$AACC[4]	$AACC[0]	$a*Dst0*
x_0|P_0,P_1	$R_1 = x_0.\mathbf{CW}_{7,0} + P_1$	-	-	-	$R_0 = x_0.\mathbf{CW}_{6,0} + P_0$	-	-	-	-
x_1|P_2,P_3	$R_3 = x_1.\mathbf{CW}_{7,0} + P_3$	$R_1 += x_1.\mathbf{CW}_{5,0}$	-	-	$R_2 = x_1.\mathbf{CW}_{6,0} + P_2$	$R_0 += x_1.\mathbf{CW}_{4,0}$	-	-	-
x_2|P_4,P_5	$R_5 = x_2.\mathbf{CW}_{7,0} + P_5$	$R_3 += x_2.\mathbf{CW}_{5,0}$	$R_1 += x_2.\mathbf{CW}_{3,0}$	-	$R_4 = x_2.\mathbf{CW}_{6,0} + P_4$	$R_2 += x_2.\mathbf{CW}_{4,0}$	$R_0 += x_2.\mathbf{CW}_{2,0}$	-	-
x_3|P_6,P_7	$R_7 = x_3.\mathbf{CW}_{7,0} + P_7$	$R_5 += x_3.\mathbf{CW}_{5,0}$	$R_3 += x_3.\mathbf{CW}_{3,0}$	$R_1 += x_3.\mathbf{CW}_{1,0}$	$R_6 = x_3.\mathbf{CW}_{6,0} + P_6$	$R_4 += x_3.\mathbf{CW}_{4,0}$	$R_2 += x_3.\mathbf{CW}_{2,0}$	$R_0 += x_3.\mathbf{CW}_{0,0}$	-
x_4|P_8,P_9	$R_9 = x_4.\mathbf{CW}_{7,0} + P_9$	$R_7 += x_4.\mathbf{CW}_{5,0}$	$R_5 += x_4.\mathbf{CW}_{3,0}$	$R_3 += x_4.\mathbf{CW}_{1,0}$	$R_8 = x_4.\mathbf{CW}_{6,0} + P_8$	$R_6 += x_4.\mathbf{CW}_{4,0}$	$R_4 += x_4.\mathbf{CW}_{2,0}$	$R_2 += x_4.\mathbf{CW}_{0,0}$	R_0,R_1
x_5|P_{10},P_{11}	$R_{11} = x_5.\mathbf{CW}_{7,0} + P_{11}$	$R_9 += x_5.\mathbf{CW}_{5,0}$	$R_7 += x_5.\mathbf{CW}_{3,0}$	$R_5 += x_5.\mathbf{CW}_{1,0}$	$R_{10} = x_5.\mathbf{CW}_{6,0} + P_{10}$	$R_8 += x_5.\mathbf{CW}_{4,0}$	$R_6 += x_5.\mathbf{CW}_{2,0}$	$R_4 += x_5.\mathbf{CW}_{0,0}$	R_2,R_3
x_6|P_{12},P_{13}	$R_{13} = x_6.\mathbf{CW}_{7,0} + P_{13}$	$R_{11} += x_6.\mathbf{CW}_{5,0}$	$R_9 += x_6.\mathbf{CW}_{3,0}$	$R_7 += x_6.\mathbf{CW}_{1,0}$	$R_{12} = x_6.\mathbf{CW}_{6,0} + P_{12}$	$R_{10} += x_6.\mathbf{CW}_{4,0}$	$R_8 += x_6.\mathbf{CW}_{2,0}$	$R_6 += x_6.\mathbf{CW}_{0,0}$	R_4,R_5

P_n is *half-precision* input partial-sum *n*
x_n is an *f16v4* input vector
$\mathbf{CW}_{m,n}$ 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 3.121: *f16v4hihoslic* instruction definition

f16v4hihoslic			worker	aux
Syntax	f16v4hihoslic $a*Dst0*, $a*Src0*:*Src0+1*, $a*Src1*, *enumFlags*			
Semantics	**Prepare**	array<Half,4> op1 = { pickHalf($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) }; DataWord op3 = enumFlags; const unsigned ampUnits = 8; bool nanoo = $FP_CTL.NANOO; array<uint64_t,2> randomBits; array<vector<Single>,ampUnits> weights; array<Single,ampUnits> 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] };		
	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 selection unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3); for (int u = 0; u < ampUnits; u++) { if (engineEnable[u >> 1]) { uint64_t fullCwei = context.getCCCSState((u * TREG_CCCS_WEIGHT_GROUP_SIZE) + 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	int scale = 0; for (int u = 0; u < ampUnits; 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];		

Continued on next page

Table 3.121: *f16v4hihoslic* instruction definition (continued)

f16v4hihoslic			worker	aux
Syntax	`f16v4hihoslic $a`*Dst0*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`,` *enumFlags*			
Semantics	**Compute cont'd**	`    } else {` `      // Engines behind me are disabled (or I am the final engine` `      // in the chain). Use the new partial-sum inputs` `      Half partialIn = (float)(op2[u & 1]);` `      if (partialIn.isNaN()) {` `        incomingp = TFPU_F32_QNan();` `      } else {` `        incomingp = partialIn;` `      }` `    }` `    // 4-element dot-product` `    result[u] = TFPU_F16DotProduct(weights[u], inputs, scale);` `    // Combine internal, incoming partial-result` `    // with result of local dot-product` `    result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32);` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`for (int u = 0; u < ampUnits; u++) {` `  if (engineEnable[u >> 1]) {` `    $AACC[u * 2] = result[u];` `  }` `}` `for (i = 0; i < 2; i++) {` `  if (isinf(out[i]) \|\| isnan(out[i])) {` `    out[i] = TFPU_F32_QNan();` `  }` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$aDst0 = {` `  Half(out[0]).bitz32(smode) |` `  (Half(out[1]).bitz32(smode) << 16)` `};`		

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

Function references: *TFPU_GenSNanCheck* , *TFPU_F32_QNan* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

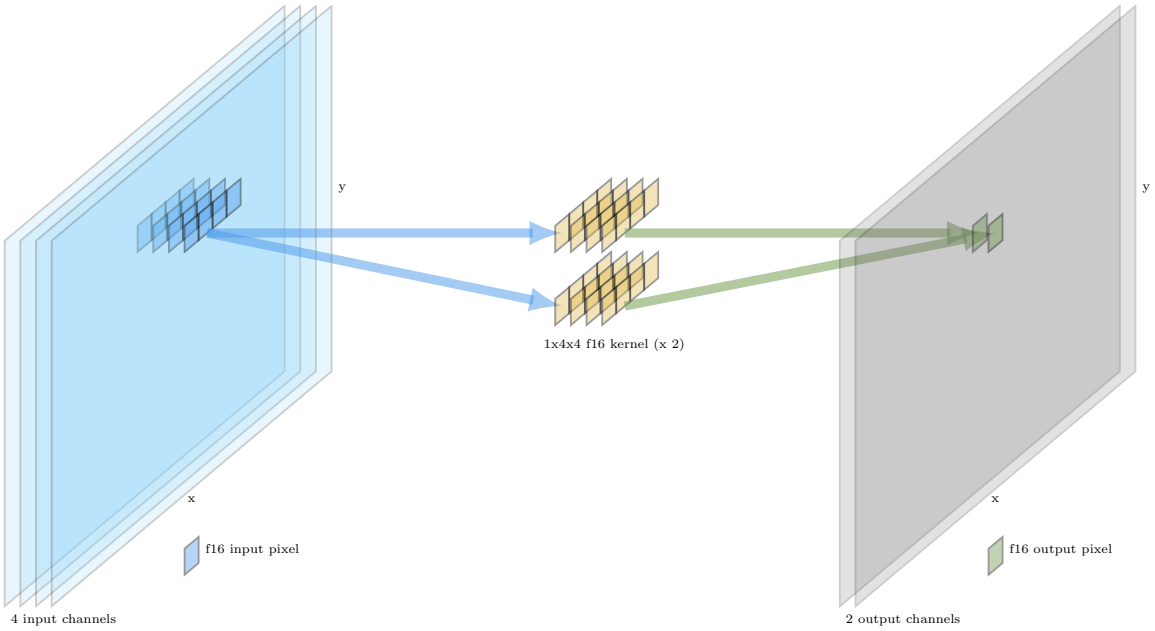


Fig. 3.14: f16v4hihoslic

3.7.3.3.18 f16v4hihov4amp

Half-precision floating-point accumulating matrix-vector product. Input and result partial-sums are 4-element *half-precision* vectors.

Table 3.122: *f16v4hihov4amp* instruction definition

f16v4hihov4amp			worker	aux
Syntax	`f16v4hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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;` `const unsigned ampUnits = 16;` `bool nanoo = $FP_CTL.NANOO;` `array<uint64_t,2> randomBits;` `vector<Single> out;` `array<vector<Single>,ampUnits> weights;` `array<Single,ampUnits> resultEven;` `array<Single,ampUnits> resultOdd;` `// Extract immediate config` `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` `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 * TREG_CCCS_WEIGHT_GROUP_SIZE) + phase);` `  uint32_t *cwei = & fullCwei;` `  weights[k] = { pickHalf(cwei[0], 0), pickHalf(cwei[0], 1),` `  pickHalf(cwei[1], 0), pickHalf(cwei[1], 1) };`		

Continued on next page

Table 3.122: *f16v4hihov4amp* instruction definition (continued)

f16v4hihov4amp			worker	aux
Syntax	`f16v4hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Except In cont'd**	`    fpExcpt |= TFPU_GenSNanCheck(weights[k].data(), weights[k].size());` `  }` `}`		
	Compute	`int scale = 0;` `// Input partial sum consumption and internal state updates` `for (unsigned k = 0; k < ampUnits; k++) {` `  unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  unsigned set = k / TFPU_AMP_UNITS_PER_SET;` `  unsigned partialIndex = (set * 2) + (k & 1);` `  if (engineEnable[engine]) {` `  // 4-element dot-product` `  resultEven[k] = TFPU_F16DotProduct(weights[k], inputs, scale);` `  Single incomingp;` `  if (phase == 0) {` `  // Even accumulators -` `  //  combine incoming partial-sum (currently stored in our` `  //  odd-accumulator) with dot-product result` `  resultEven[k] =` `  TFPU_Add($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 = $AACC[(k + 2) * 2];` `  } else {` `  // Engines behind me are disabled (or I am the final` `  // engine in the chain). Use new partial-sum inputs` `  Half partialIn = (float)(op2[partialIndex]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  } else { // Phase != 0` `  // Even accumulators -` `  //  accumulate dot-product result` `  resultEven[k] = TFPU_Add($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];` `  } else {` `  // Engines behind me are disabled (or I am the final` `  // engine in the chain). Use new partial-sum inputs` `  Half partialIn = (float)(op2[partialIndex]);`		

Continued on next page

Table 3.122: *f16v4hihov4amp* instruction definition (continued)

f16v4hihov4amp			worker	aux
Syntax	`f16v4hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Compute cont'd**	`if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `} else {` `  incomingp = (Single)partialIn;` `}` `}` `}` `resultOdd[k] = incomingp;` `}` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// 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];` `  $AACC[(k * 2) + 1] = resultOdd[k];` `  }` `}` `for (unsigned i = 0; i < 4; i++) {` `  if (isinf(out[i]) \|\| isnan(out[i])) {` `  out[i] = TFPU_F32_QNan();` `  }` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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

Function references: *TFPU_GenSNanCheck* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

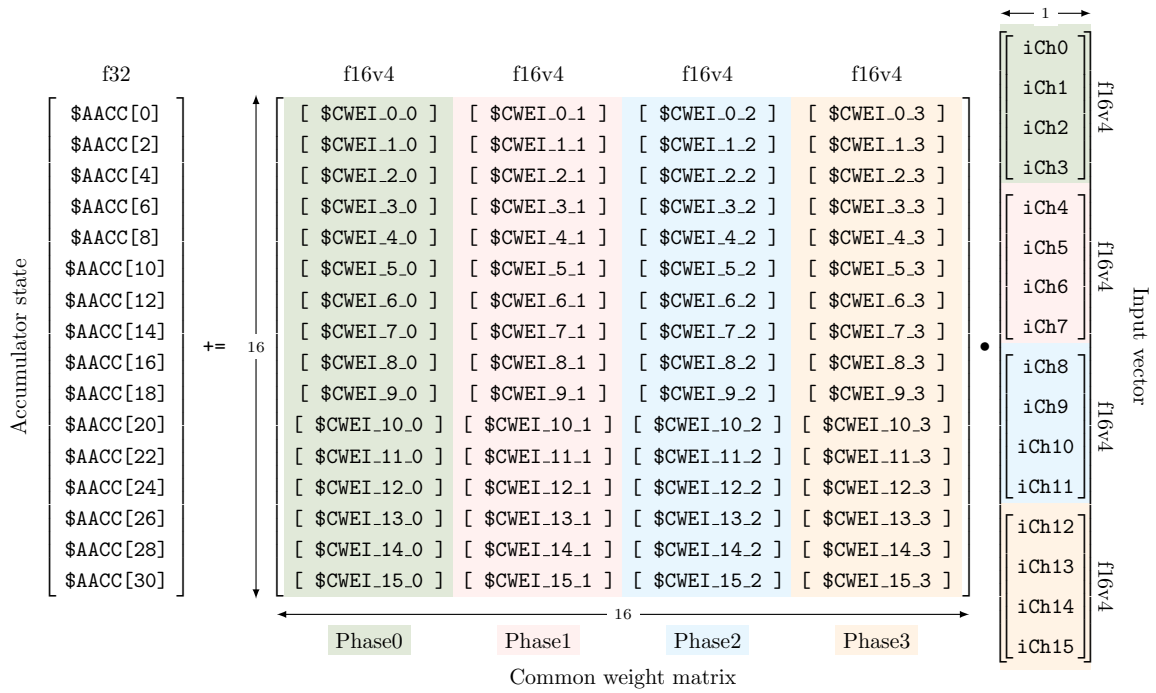


Fig. 3.15: f16v4hihov4amp

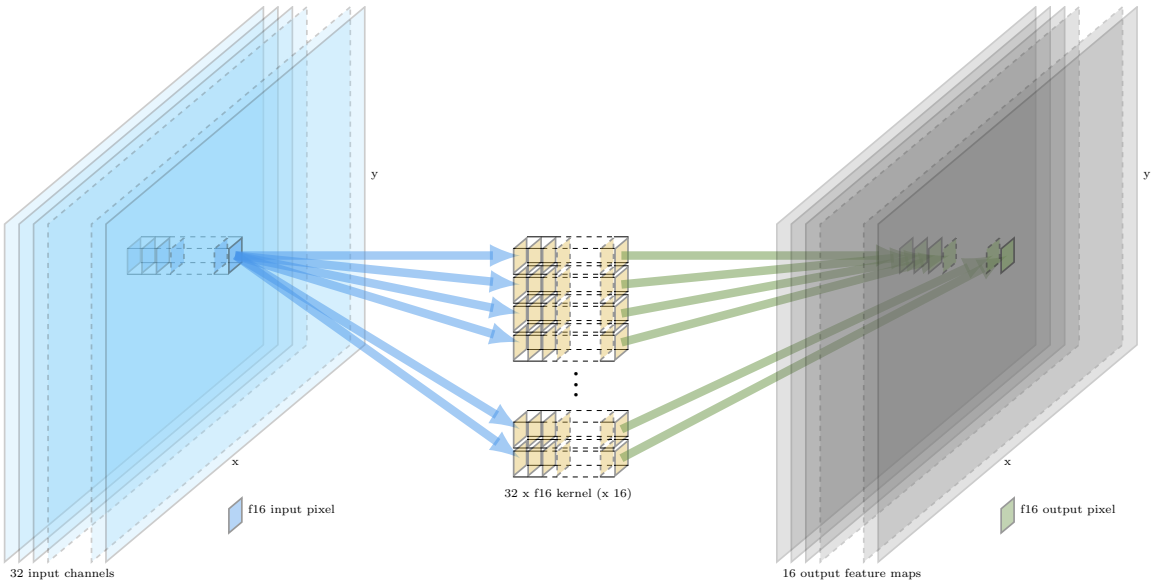


Fig. 3.16: f16v4hihov4amp

3.7.3.3.19 f16v4hihov4slic

Half-precision floating-point slim convolution. Input and result partial-sums are 4 x *half-precision* values.

Table 3.123: *f16v4hihov4slic* instruction definition

f16v4hihov4slic			worker	aux
Syntax	`f16v4hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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;` `const unsigned ampUnits = 16;` `bool nanoo = $FP_CTL.NANOO;` `array<uint64_t,2> randomBits;` `array<vector<Single>,ampUnits> weights;` `array<Double,ampUnits> result;` `// Extract immediate config` `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.getCCCSState((u * TREG_CCCS_WEIGHT_GROUP_SIZE) + 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());` `  }` `}`		

Continued on next page

Table 3.123: *f16v4hihov4slic* instruction definition (continued)

f16v4hihov4slic			worker	aux
Syntax	`f16v4hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Compute**	`int scale = 0;` `// 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 = $AACC[(u + 2) * 2];` `  } else {` `  // Engines behind me are disabled (or I am the final engine` `  // in the chain). Use the new partial-sum inputs` `  unsigned set = u / TFPU_AMP_UNITS_PER_SET;` `  unsigned partialIndex = (set * 2) + (u & 1);` `  Half partialIn = (float)(op2[partialIndex]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  // 4-element dot-product` `  result[u] = TFPU_F16DotProduct(weights[u], inputs, scale);` `  // Combine internal, incoming partial-result` `  // with result of local dot-product` `  result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32);` `  }` `}`		
	Except **Out**	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// 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();` `  }` `}`		

Continued on next page

Table 3.123: *f16v4hihov4slic* instruction definition (continued)

f16v4hihov4slic			**worker**	**aux**
Syntax	`f16v4hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Commit cont'd**	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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

Function references: *TFPU_GenSNanCheck* , *TFPU_F32_QNan* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

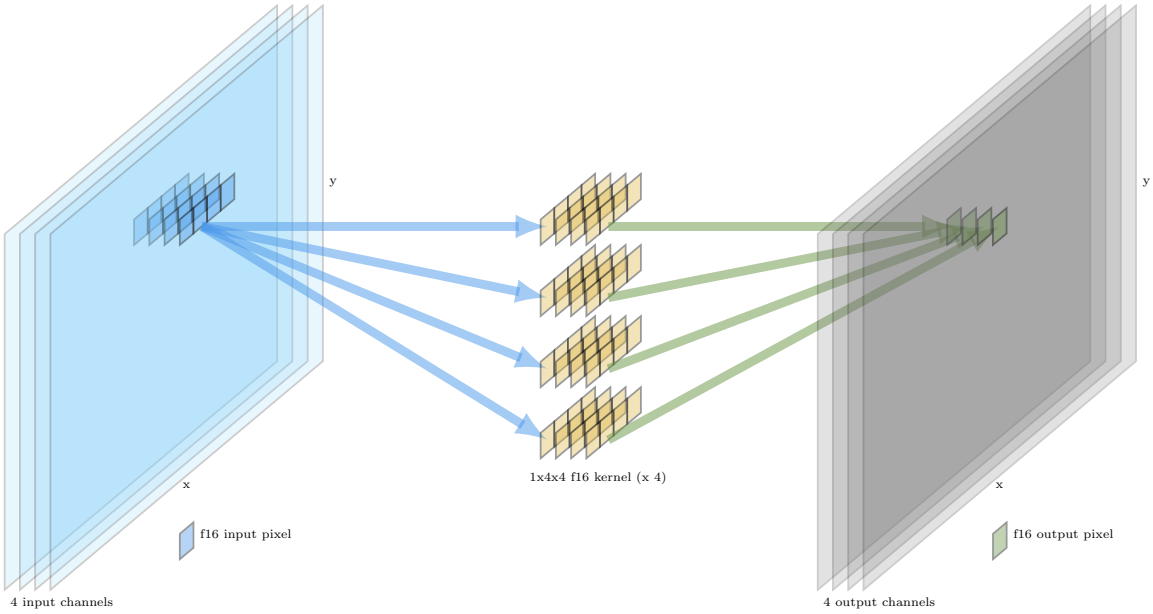


Fig. 3.17: f16v4hihov4slic

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 3.124: *f16v4istacc* instruction definition

f16v4istacc			worker	aux
Syntax	`f16v4istacc $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	`array<HalfDataWord,4> op1 = {` `  (($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]` `};`		

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 3.125: *f16v4max* instruction definition

f16v4max			worker	aux
Syntax	`f16v4max $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[0], 1, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 1, PROP_INF)` `};` `array<Half,4> op2 = {` `  pickHalf($aSrc1:Src1+1[0], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[0], 1, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 1, PROP_INF)` `};` `array<Half,4> z;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$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)` `};`		

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 3.126: *f16v4maxc* instruction definition

f16v4maxc			worker	aux
Syntax	`f16v4maxc $a`*Dst0*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[0], 1, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 1, PROP_INF)` `};` `array<Half,2> max;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (op1[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`max[0] = TFPU_Max(op1[0], op1[1]);` `max[1] = TFPU_Max(op1[2], op1[3]);`		
	Commit	`$aDst0 = {` `  Half(max[0]).bitz32(PROP_INF) |` `  (Half(max[1]).bitz32(PROP_INF) << 16)` `};`		

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 3.127: *f16v4min* instruction definition

f16v4min			worker	aux
Syntax	`f16v4min $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[0], 1, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 0, PROP_INF),` `  pickHalf($aSrc0:Src0+1[1], 1, PROP_INF)` `};` `array<Half,4> op2 = {` `  pickHalf($aSrc1:Src1+1[0], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[0], 1, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 0, PROP_INF),` `  pickHalf($aSrc1:Src1+1[1], 1, PROP_INF)` `};` `array<Half,4> z;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (op1[i].issNaN() || op2[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$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)` `};`		

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

Function references: *TFPU_IsMalign* , *TFPU_Min*

3.7.3.3.24 f16v4mix

Half-precision 4-element vector $\mathbf{z} = a\mathbf{x} + b\mathbf{y}$. 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 3.128: *f16v4mix* instruction definition

f16v4mix			worker	aux
Syntax	`f16v4mix $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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)` `};` `bool nanoo = $FP_CTL.NANOO;` `bool enableStochasticRounding = $FP_CTL.ESR;` `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($TAS, 1);`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `// Input exception check` `for (i = 0; i < 4; ++i) {` `  specialCase[i] = TFPU_DoAxpbyPreExecute(a, op1[i], b, op2[i],` `  &fpExcpt, &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `}`		
	Compute	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, out);` `}` `// z = ax + by` `for (i = 0; i < 4; ++i) {` `  if (!specialCase[i]) {` `  vector<Single> xv = { a, b };` `  vector<Single> yv = { op1[i], op2[i] };` `  result[i] = TFPU_F16DotProduct(xv, yv, 0);` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`$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();` `  }` `}`		

Continued on next page

Table 3.128: *f16v4mix* instruction definition (continued)

f16v4mix			**worker**	**aux**
Syntax	`f16v4mix $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Commit cont'd**	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_F16DotProduct* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_GetNanooMode*

3.7.3.3.25 f16v4mul

Half-precision 4-element vector, Hadamard product

Table 3.129: *f16v4mul* instruction definition

f16v4mul			worker	aux
Syntax	`f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BL, $a`*Src1*`:`*Src1+1* `f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BU, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf(<($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 = $FP_CTL.ESR;` `TileRoundMode_t rmode;` `vector<Single> result;` `array<bool,4> specialCase;` `array<uint64_t,2> randomBits;` `if (enableStochasticRounding) {` `  rmode = TFPU_ROUND_EVEN;` `} else {` `  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, $PRNG_1_1, randomBits);` `}`		
	Compute	`for (i = 0; i < 4; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) *` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16, rmode);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 4; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		

Continued on next page

Table 3.129: *f16v4mul* instruction definition (continued)

f16v4mul			worker	aux
Syntax	`f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BL, $a`*Src1*`:`*Src1+1* `f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f16v4mul $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:BU, $a`*Src1*`:`*Src1+1*			
Semantics	**Commit**	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

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 $a*Src1*[16] == 1, no masking is applied (the result is a copy of the input vector)
- else if $a*Src1*[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 3.130: *f16v4rmask* instruction definition

f16v4rmask			worker	aux
Syntax	`f16v4rmask $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*			
Semantics	**Prepare**	`array<HalfDataWord,4> op0;` `array<HalfDataWord,4> op1 = {` `  (($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 = $aSrc1;`		
	Compute	`bool always = ((op2 >> 16) & 1) == 1;` `bool never = (op2 & 0x1ffff) == 0;` `array<uint64_t,2> randomBits;` `TPRNG_Advance($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)` `};`		

Architectural state references: *$PRNG_0_0* , *$PRNG_0_1* , *$PRNG_1_0* , *$PRNG_1_1*

3.7.3.3.27 f16v4sisoamp

f16 floating-point accumulating matrix-vector product. Input and result partial-sums are 2 x *single-precision* values.

Table 3.131: f16v4sisoamp 8x1x1x16 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[12]	$AACC[10]	$AACC[8]	$AACC[6]	$AACC[4]	$AACC[2]	$AACC[0]	$a*Dst0*
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	$R_7 = x_0.CW_{7,0} + P_7$	$R_6 = x_0.CW_{6,0} + P_6$	$R_5 = x_0.CW_{5,0} + P_5$	$R_4 = x_0.CW_{4,0} + P_4$	$R_3 = x_0.CW_{3,0} + P_3$	$R_2 = x_0.CW_{2,0} + P_2$	$R_1 = x_0.CW_{1,0} + P_1$	$R_0 = x_0.CW_{0,0} + P_0$	-
x_1 | P_{10},P_{11}	$R_7 += x_1.CW_{7,1}$	$R_6 += x_1.CW_{6,1}$	$R_5 += x_1.CW_{5,1}$	$R_4 += x_1.CW_{4,1}$	$R_3 += x_1.CW_{3,1}$	$R_2 += x_1.CW_{2,1}$	$R_1 += x_1.CW_{1,1}$	$R_0 += x_1.CW_{0,1}$	-
x_2 | P_{12},P_{13}	$R_7 += x_2.CW_{7,2}$	$R_6 += x_2.CW_{6,2}$	$R_5 += x_2.CW_{5,2}$	$R_4 += x_2.CW_{4,2}$	$R_3 += x_2.CW_{3,2}$	$R_2 += x_2.CW_{2,2}$	$R_1 += x_2.CW_{1,2}$	$R_0 += x_2.CW_{0,2}$	-
x_3 | P_{14},P_{15}	$R_7 += x_3.CW_{7,3}$	$R_6 += x_3.CW_{6,3}$	$R_5 += x_3.CW_{5,3}$	$R_4 += x_3.CW_{4,3}$	$R_3 += x_3.CW_{3,3}$	$R_2 += x_3.CW_{2,3}$	$R_1 += x_3.CW_{1,3}$	$R_0 += x_3.CW_{0,3}$	-
x_4 | P_{16},P_{17}	$R_{15} = x_4.CW_{7,0} + P_{15}$	$R_{14} = x_4.CW_{6,0} + P_{14}$	$R_{13} = x_4.CW_{5,0} + P_{13}$	$R_{12} = x_4.CW_{4,0} + P_{12}$	$R_{11} = x_4.CW_{3,0} + P_{11}$	$R_{10} = x_4.CW_{2,0} + P_{10}$	$R_9 = x_4.CW_{1,0} + P_9$	$R_8 = x_4.CW_{0,0} + P_8$	R_0,R_1
x_5 | P_{18},P_{19}	$R_{15} += x_5.CW_{7,1}$	$R_{14} += x_5.CW_{6,1}$	$R_{13} += x_5.CW_{5,1}$	$R_{12} += x_5.CW_{4,1}$	$R_{11} += x_5.CW_{3,1}$	$R_{10} += x_5.CW_{2,1}$	$R_9 += x_5.CW_{1,1}$	$R_8 += x_5.CW_{0,1}$	R_2,R_3
x_6 | P_{20},P_{21}	$R_{15} += x_6.CW_{7,2}$	$R_{14} += x_6.CW_{6,2}$	$R_{13} += x_6.CW_{5,2}$	$R_{12} += x_6.CW_{4,2}$	$R_{11} += x_6.CW_{3,2}$	$R_{10} += x_6.CW_{2,2}$	$R_9 += x_6.CW_{1,2}$	$R_8 += x_6.CW_{0,2}$	R_4,R_5
x_7 | P_{22},P_{23}	$R_{15} += x_7.CW_{7,3}$	$R_{14} += x_7.CW_{6,3}$	$R_{13} += x_7.CW_{5,3}$	$R_{12} += x_7.CW_{4,3}$	$R_{11} += x_7.CW_{3,3}$	$R_{10} += x_7.CW_{2,3}$	$R_9 += x_7.CW_{1,3}$	$R_8 += x_7.CW_{0,3}$	R_6,R_7
x_8 | P_{24},P_{25}	$R_{23} = x_8.CW_{7,0} + P_{23}$	$R_{22} = x_8.CW_{6,0} + P_{22}$	$R_{21} = x_8.CW_{5,0} + P_{21}$	$R_{20} = x_8.CW_{4,0} + P_{20}$	$R_{19} = x_8.CW_{3,0} + P_{19}$	$R_{18} = x_8.CW_{2,0} + P_{18}$	$R_{17} = x_8.CW_{1,0} + P_{17}$	$R_{16} = x_8.CW_{0,0} + P_{16}$	R_8,R_9
x_9 | P_{26},P_{27}	$R_{23} += x_9.CW_{7,1}$	$R_{22} += x_9.CW_{6,1}$	$R_{21} += x_9.CW_{5,1}$	$R_{20} += x_9.CW_{4,1}$	$R_{19} += x_9.CW_{3,1}$	$R_{18} += x_9.CW_{2,1}$	$R_{17} += x_9.CW_{1,1}$	$R_{16} += x_9.CW_{0,1}$	R_{10},R_{11}
x_{10} | P_{28},P_{29}	$R_{23} += x_{10}.CW_{7,2}$	$R_{22} += x_{10}.CW_{6,2}$	$R_{21} += x_{10}.CW_{5,2}$	$R_{20} += x_{10}.CW_{4,2}$	$R_{19} += x_{10}.CW_{3,2}$	$R_{18} += x_{10}.CW_{2,2}$	$R_{17} += x_{10}.CW_{1,2}$	$R_{16} += x_{10}.CW_{0,2}$	R_{12},R_{13}
x_{11} | P_{30},P_{31}	$R_{23} += x_{11}.CW_{7,3}$	$R_{22} += x_{11}.CW_{6,3}$	$R_{21} += x_{11}.CW_{5,3}$	$R_{20} += x_{11}.CW_{4,3}$	$R_{19} += x_{11}.CW_{3,3}$	$R_{18} += x_{11}.CW_{2,3}$	$R_{17} += x_{11}.CW_{1,3}$	$R_{16} += x_{11}.CW_{0,3}$	R_{14},R_{15}
x_{12} | P_{32},P_{33}	$R_{31} = x_{12}.CW_{7,0} + P_{31}$	$R_{30} = x_{12}.CW_{6,0} + P_{30}$	$R_{29} = x_{12}.CW_{5,0} + P_{29}$	$R_{28} = x_{12}.CW_{4,0} + P_{28}$	$R_{27} = x_{12}.CW_{3,0} + P_{27}$	$R_{26} = x_{12}.CW_{2,0} + P_{26}$	$R_{25} = x_{12}.CW_{1,0} + P_{25}$	$R_{24} = x_{12}.CW_{0,0} + P_{24}$	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:

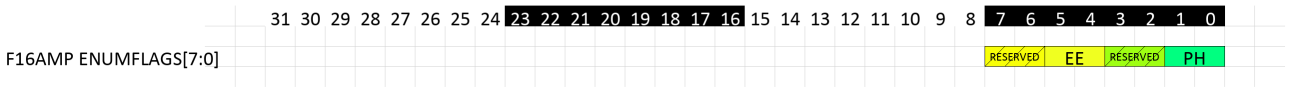


Fig. 3.18: f16v4sisoamp immediate format

8 output channels are processed/produced.

Table 3.132: *f16v4sisoamp* instruction definition

f16v4sisoamp			worker	aux
Syntax	`f16v4sisoamp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($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;` `const unsigned ampUnits = 8;` `array<uint64_t,2> randomBits;` `vector<Single> out;` `array<vector<Single>,ampUnits> weights;` `array<Single,ampUnits> resultEven;` `array<Single,ampUnits> resultOdd;` `// Extract immediate config` `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 (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 * TREG_CCCS_WEIGHT_GROUP_SIZE) + phase);` `  uint32_t *cwei = & fullCwei;` `  weights[k] = {` `  pickFlt16(fp16Fmt, cwei[0], 0), pickFlt16(fp16Fmt, cwei[0], 1),` `  pickFlt16(fp16Fmt, cwei[1], 0), pickFlt16(fp16Fmt, cwei[1], 1)` `  };` `  fpExcpt |= TFPU_GenSNanCheck(weights[k].data(), weights[k].size());` `  }` `}`		

Continued on next page

Table 3.132: *f16v4sisoamp* instruction definition (continued)

f16v4sisoamp			worker	aux
Syntax	f16v4sisoamp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags			
Semantics	**Compute**	`int scale = 0;` `// Input partial sum consumption and internal state updates` `for (unsigned k = 0; k < ampUnits; k++) {` `  unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  // 4-element dot-product` `  resultEven[k] = TFPU_F16DotProduct(weights[k], inputs, scale, fp16Fmt);` `  Single incomingp;` `  if (phase == 0) {` `  // Even accumulators -` `  //  combine incoming partial-sum (currently stored in our` `  //  odd-accumulator) with dot-product result` `  resultEven[k] =` `  TFPU_Add($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;` `  }` `}`		

Continued on next page

Table 3.132: *f16v4sisoamp* instruction definition (continued)

f16v4sisoamp			worker	aux
Syntax	`f16v4sisoamp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Except Out**	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// 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])` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_GenSNanCheck* , *TFPU_F16DotProduct* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

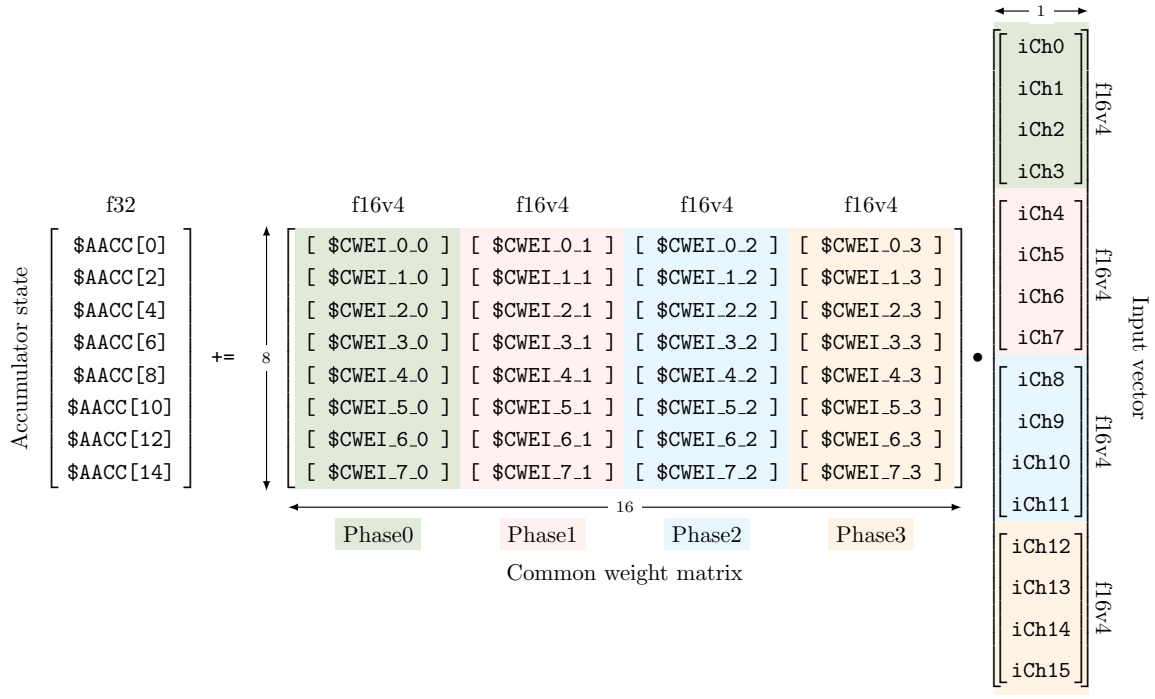


Fig. 3.19: f16v4sisoamp

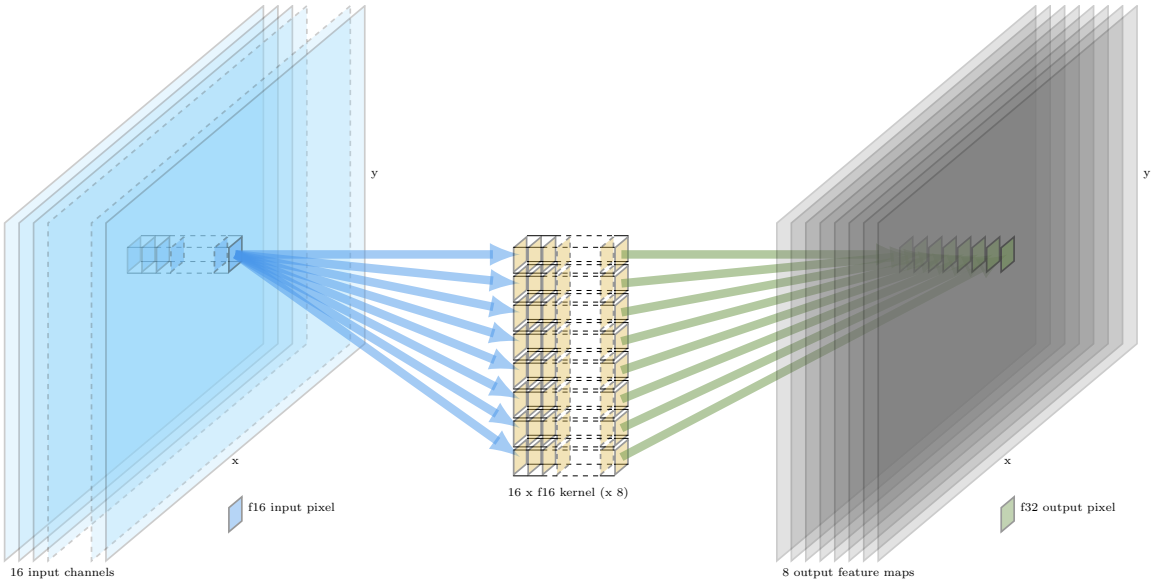


Fig. 3.20: f16v4sisoamp

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:
```

3.7.3.3.28 f16v4sisoslic

f16 floating-point slim convolution. Input and result partial-sums are 2 x *single-precision* values.

Table 3.133: f16v4sisoslic, 2x1x3x4 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[10]	$AACC[6]	$AACC[2]	$AACC[12]	$AACC[8]	$AACC[4]	$AACC[0]	$a*Dst0*
x_0|P_0,P_1	-	$R_1 = x_0.\mathbf{CW}_{5,0} + P_1$	-	-	-	$R_0 = x_0.\mathbf{CW}_{4,0} + P_0$	-	-	-
x_1|P_2,P_3	-	$R_3 = x_1.\mathbf{CW}_{5,0} + P_3$	$R_1 += x_1.\mathbf{CW}_{3,0}$	-	-	$R_2 = x_1.\mathbf{CW}_{4,0} + P_2$	$R_0 += x_1.\mathbf{CW}_{2,0}$	-	-
x_2|P_4,P_5	-	$R_5 = x_2.\mathbf{CW}_{5,0} + P_5$	$R_3 += x_2.\mathbf{CW}_{3,0}$	$R_1 += x_2.\mathbf{CW}_{1,0}$	-	$R_4 = x_2.\mathbf{CW}_{4,0} + P_4$	$R_2 += x_2.\mathbf{CW}_{2,0}$	$R_0 += x_2.\mathbf{CW}_{0,0}$	-
x_3|P_6,P_7	-	$R_7 = x_3.\mathbf{CW}_{5,0} + P_7$	$R_5 += x_3.\mathbf{CW}_{3,0}$	$R_3 += x_3.\mathbf{CW}_{1,0}$	-	$R_6 = x_3.\mathbf{CW}_{4,0} + P_6$	$R_4 += x_3.\mathbf{CW}_{2,0}$	$R_2 += x_3.\mathbf{CW}_{0,0}$	R_0,R_1
x_4|P_8,P_9	-	$R_9 = x_4.\mathbf{CW}_{5,0} + P_9$	$R_7 += x_4.\mathbf{CW}_{3,0}$	$R_5 += x_4.\mathbf{CW}_{1,0}$	-	$R_8 = x_4.\mathbf{CW}_{4,0} + P_8$	$R_6 += x_4.\mathbf{CW}_{2,0}$	$R_4 += x_4.\mathbf{CW}_{0,0}$	R_2,R_3
x_5|P_{10},P_{11}	-	$R_{11} = x_5.\mathbf{CW}_{5,0} + P_{11}$	$R_9 += x_5.\mathbf{CW}_{3,0}$	$R_7 += x_5.\mathbf{CW}_{1,0}$	-	$R_{10} = x_5.\mathbf{CW}_{4,0} + P_{10}$	$R_8 += x_5.\mathbf{CW}_{2,0}$	$R_6 += x_5.\mathbf{CW}_{0,0}$	R_4,R_5
x_6|P_{12},P_{13}	-	$R_{13} = x_6.\mathbf{CW}_{5,0} + P_{13}$	$R_{11} += x_6.\mathbf{CW}_{3,0}$	$R_9 += x_6.\mathbf{CW}_{1,0}$	-	$R_{12} = x_6.\mathbf{CW}_{4,0} + P_{12}$	$R_{10} += x_6.\mathbf{CW}_{2,0}$	$R_8 += x_6.\mathbf{CW}_{0,0}$	R_6,R_7

Table 3.134: f16v4sisoslic, 2x1x4x4 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[10]	$AACC[6]	$AACC[2]	$AACC[12]	$AACC[8]	$AACC[4]	$AACC[0]	$a*Dst0*
x_0|P_0,P_1	$R_1 = x_0.\mathbf{CW}_{7,0} + P_1$	-	-	-	$R_0 = x_0.\mathbf{CW}_{6,0} + P_0$	-	-	-	-
x_1|P_2,P_3	$R_3 = x_1.\mathbf{CW}_{7,0} + P_3$	$R_1 += x_1.\mathbf{CW}_{5,0}$	-	-	$R_2 = x_1.\mathbf{CW}_{6,0} + P_2$	$R_0 += x_1.\mathbf{CW}_{4,0}$	-	-	-
x_2|P_4,P_5	$R_5 = x_2.\mathbf{CW}_{7,0} + P_5$	$R_3 += x_2.\mathbf{CW}_{5,0}$	$R_1 += x_2.\mathbf{CW}_{3,0}$	-	$R_4 = x_2.\mathbf{CW}_{6,0} + P_4$	$R_2 += x_2.\mathbf{CW}_{4,0}$	$R_0 += x_2.\mathbf{CW}_{2,0}$	-	-
x_3|P_6,P_7	$R_7 = x_3.\mathbf{CW}_{7,0} + P_7$	$R_5 += x_3.\mathbf{CW}_{5,0}$	$R_3 += x_3.\mathbf{CW}_{3,0}$	$R_1 += x_3.\mathbf{CW}_{1,0}$	$R_6 = x_3.\mathbf{CW}_{6,0} + P_6$	$R_4 += x_3.\mathbf{CW}_{4,0}$	$R_2 += x_3.\mathbf{CW}_{2,0}$	$R_0 += x_3.\mathbf{CW}_{0,0}$	-
x_4|P_8,P_9	$R_9 = x_4.\mathbf{CW}_{7,0} + P_9$	$R_7 += x_4.\mathbf{CW}_{5,0}$	$R_5 += x_4.\mathbf{CW}_{3,0}$	$R_3 += x_4.\mathbf{CW}_{1,0}$	$R_8 = x_4.\mathbf{CW}_{6,0} + P_8$	$R_6 += x_4.\mathbf{CW}_{4,0}$	$R_4 += x_4.\mathbf{CW}_{2,0}$	$R_2 += x_4.\mathbf{CW}_{0,0}$	R_0,R_1
x_5|P_{10},P_{11}	$R_{11} = x_5.\mathbf{CW}_{7,0} + P_{11}$	$R_9 += x_5.\mathbf{CW}_{5,0}$	$R_7 += x_5.\mathbf{CW}_{3,0}$	$R_5 += x_5.\mathbf{CW}_{1,0}$	$R_{10} = x_5.\mathbf{CW}_{6,0} + P_{10}$	$R_8 += x_5.\mathbf{CW}_{4,0}$	$R_6 += x_5.\mathbf{CW}_{2,0}$	$R_4 += x_5.\mathbf{CW}_{0,0}$	R_2,R_3
x_6|P_{12},P_{13}	$R_{13} = x_6.\mathbf{CW}_{7,0} + P_{13}$	$R_{11} += x_6.\mathbf{CW}_{5,0}$	$R_9 += x_6.\mathbf{CW}_{3,0}$	$R_7 += x_6.\mathbf{CW}_{1,0}$	$R_{12} = x_6.\mathbf{CW}_{6,0} + P_{12}$	$R_{10} += x_6.\mathbf{CW}_{4,0}$	$R_8 += x_6.\mathbf{CW}_{2,0}$	$R_6 += x_6.\mathbf{CW}_{0,0}$	R_4,R_5

P_n is *single-precision* input partial-sum *n*
x_n is an *f16v4* input vector
$\mathbf{CW}_{m,n}$ 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:

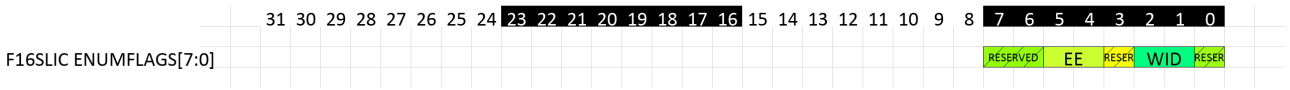


Fig. 3.21: f16v4sisoslic immediate format

Table 3.135: *f16v4sisoslic* instruction definition

f16v4sisoslic			worker	aux
Syntax	f16v4sisoslic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags			
Semantics	**Prepare**	array<Half,4> op1 = { pickHalf($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; const unsigned ampUnits = 8; array<uint64_t,2> randomBits; array<vector<Single>,ampUnits> weights; array<Double,ampUnits> 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 // 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 * TREG_CCCS_WEIGHT_GROUP_SIZE) + wid); uint32_t *cwei = & fullCwei; weights[u] = { pickFlt16(fp16Fmt, cwei[0], 0), pickFlt16(fp16Fmt, cwei[0], 1), pickFlt16(fp16Fmt, cwei[1], 0), pickFlt16(fp16Fmt, cwei[1], 1) }; fpExcpt |= TFPU_GenSNanCheck(weights[u].data(), weights[u].size()); } }		
	Compute	int scale = 0; // Input partial-sum consumption for (unsigned u = 0; u < ampUnits; u++) { unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;		

Continued on next page

Table 3.135: *f16v4sisoslic* instruction definition (continued)

f16v4sisoslic			worker	aux
Syntax	`f16v4sisoslic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Compute cont'd**	`if (engineEnable[engine]) {` `  Single incomingp;` `  if (engineEnable[engine + 1]) {` `  // Engine behind me is enabled` `  // use its current accumulator value` `  incomingp = $AACC[(u + 2) * 2];` `  } else {` `  // Engines behind me are disabled (or I am the final engine` `  // in the chain). Use the new partial-sum inputs` `  incomingp = op2[u & 1];` `  if (isnan(incomingp)) {` `  incomingp = TFPU_F32_QNan();` `  }` `  }` `  // 4-element dot-product` `  result[u] = TFPU_F16DotProduct(weights[u], inputs, scale, fp16Fmt);` `  // Combine internal, incoming partial-result` `  // with result of local dot-product` `  result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32);` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// 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])` `};`		

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*

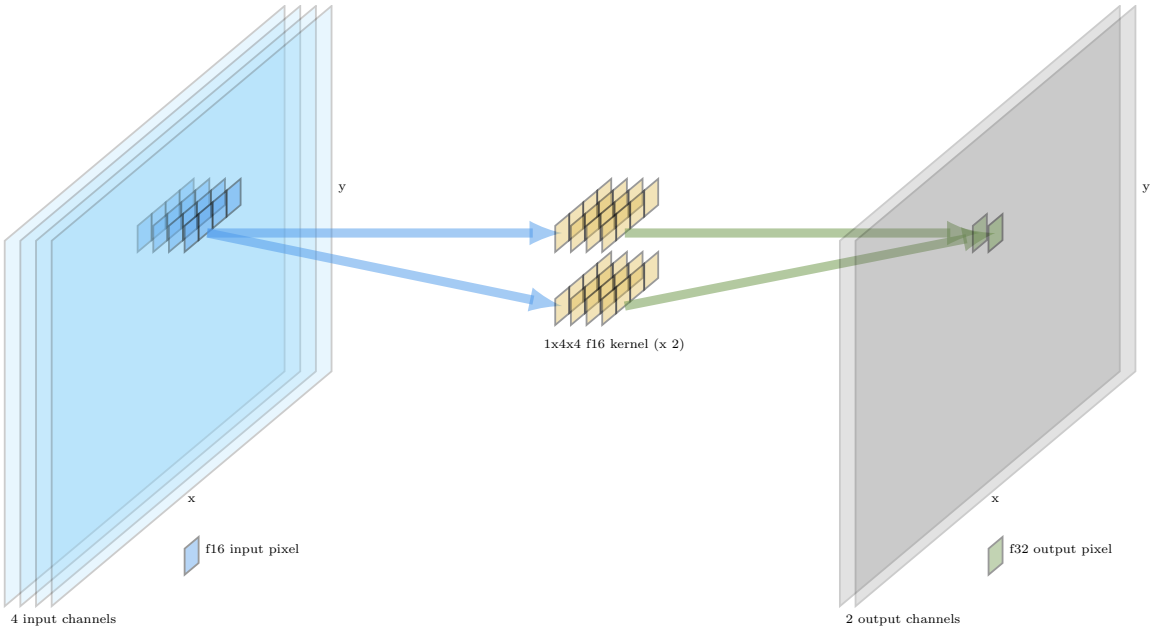


Fig. 3.22: f16v4sisoslic

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+=, $mzero, 0
  {
    ld2x64pace    $inData, $inPartials, $triPtr+=, $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+=, $mzero, 0
    f16v4sisoslic $outPartials, $inData, $inPartials, TSLIC_F16V4_1x3_W0
  }
_loop_end:

  // Store final outputs
  st64pace      $outPartials, $triPtr+=, $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+=, $mzero, 0
  {
```

```
    ld2x64pace    $inData, $inPartials, $triPtr+=, $mzero, 0
    f16v4sisoslic $outPartialsA, $inData, $inPartials, TSLIC_F16V4_1x4_W0
  }
  {
    ld2x64pace    $inData, $inPartials, $triPtr+=, $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, $triPtr+=, $mzero, 0
    f16v4sisoslic $outPartialsA, $inData, $inPartials, TSLIC_F16V4_1x4_W0
  }
  {
    ld2xst64pace  $inDataInPartials, $outPartialsB, $triPtr+=, $mzero, 0
    f16v4sisoslic $outPartialsB, $inData, $inPartials, TSLIC_F16V4_1x4_W0
  }
_loop_end:

  // Store final outputs
  st64pace      $outPartialsA, $triPtr+=, $mzero, 0
```

3.7.3.3.29 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 3.136: *f16v4stacc* instruction definition

f16v4stacc			worker	aux
Syntax	`f16v4stacc $a`*Dst0*`:`*Dst0+1*`, `*enumFlags*			
Semantics	**Prepare**	`array<DataWord,2> op0;` `DataWord op1 = enumFlags;`		
	Commit	`int phase = op1 & 1;` `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]` `};`		

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 $azeros, $a0:1, $a2:3, TISTACC_P0
}
{
  ld64step $a0:1, $mzero, $inputPtr+=, 4
  f16v4istacc $azeros, $a4:5, $a6:7, TISTACC_P1
}
{
  ld64step $a2:3, $mzero, $inputPtr+=, 4
  f16v4stacc $a6:7, TSTACC_P0
```

```
  }

  tapack $triPtr, $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, $triPtr+=, $strides, INC_ST4_LD4
    f16v4stacc    $a6:7, TSTACC_P1
  }
  {
    ldst64pace    $a6:7, $a6:7, $triPtr+=, $strides, INC_ST4_LDM11
    f16v4istacc   $a2:3, $a0:1, $a2:3, TISTACC_P0
  }
  {
    ldst64pace    $a0:1, $a2:3, $triPtr+=, $strides, INC_ST4_LD4
    f16v4istacc   $a2:3, $a4:5, $a6:7, TISTACC_P1
  }
  {
    ldst64pace    $a2:3, $a2:3, $triPtr+=, $strides, INC_ST4_LD4
    f16v4stacc    $a6:7, TSTACC_P0
  }
_inner_loop_end:

  {
    ldst64pace    $a4:5, $a6:7, $triPtr+=, $strides, INC_ST4_LD4
    f16v4stacc    $a6:7, TSTACC_P1
  }
  {
    ldst64pace    $a6:7, $a6:7, $triPtr+=, $strides, INC_ST4_LD1
    f16v4istacc   $a2:3, $a0:1, $a2:3, TISTACC_P0
  }
  {
    ldst64pace    $a0:1, $a2:3, $triPtr+=, $strides, INC_ST4_LD4
    f16v4istacc   $a2:3, $a4:5, $a6:7, TISTACC_P1
  }
  {
    ldst64pace    $a2:3, $a2:3, $triPtr+=, $strides, INC_STM59_LD4
    f16v4stacc    $a6:7, TSTACC_P0
  }

  brnzdec       $counter, _outer_loop_start

_outer_loop_end:
```

3.7.3.3.30 f16v4sub

Half-precision floating-point 4-element vector subtraction

Table 3.137: *f16v4sub* instruction definition

f16v4sub			worker	aux
Syntax	f16v4sub $a*Dst0*:*Dst0+1*, $a*Src0*:BL, $a*Src1*:*Src1+1* f16v4sub $a*Dst0*:*Dst0+1*, $a*Src0*:*Src0+1*, $a*Src1*:*Src1+1* f16v4sub $a*Dst0*:*Dst0+1*, $a*Src0*:BU, $a*Src1*:*Src1+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf(<($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 = $FP_CTL.ESR;` `vector<Single> result;` `array<bool,4> specialCase;` `array<uint64_t,2> randomBits;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `result.resize(4);` `for (i = 0; i < 4; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}` `if (enableStochasticRounding) {` `  TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1,` `  randomBits);` `}`		
	Compute	`for (i = 0; i < 4; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i])  -` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res,` `  enableStochasticRounding ? TFPU_FP32 : TFPU_FP16,` `  TFPU_ROUND_EVEN);` `  }` `}`		
	Except Out	`if (enableStochasticRounding) {` `  TFPU_ApplyStochasticRoundHalf(randomBits, result);` `}` `for (i = 0; i < 4; ++i) {` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP16, nanoo);` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

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_ApplyStochasticRoundHalf* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_GetNanooMode*

3.7.3.3.31 f16v4sum

Half-precision 4-element vector 2x2 lateral summation to 2-element *single-precision* vector.

Table 3.138: *f16v4sum* instruction definition

f16v4sum			worker	aux
Syntax	`f16v4sum $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<Half,4> op1 = {` `  pickHalf($aSrc0:Src0+1[0], 0),` `  pickHalf($aSrc0:Src0+1[0], 1),` `  pickHalf($aSrc0:Src0+1[1], 0),` `  pickHalf($aSrc0:Src0+1[1], 1)` `};` `array<bool,2> specialCase;` `array<Single,2> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `specialCase[0] = TFPU_DoAddPreExecute(op1[0], op1[1], &fpExcpt, &result[0]);` `specialCase[1] = TFPU_DoAddPreExecute(op1[2], op1[3], &fpExcpt, &result[1]);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  Double res = op1[2*i] + op1[(2*i)+1];` `  result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  }` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.139: *f16v8absacc* instruction definition

f16v8absacc			worker	aux
Syntax	`f16v8absacc $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Half,8> op0 = {` `  pickHalf($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($aSrc0:Src0+3[3], 1)` `};` `array<Single,8> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 8; ++i) {` `  if (op0[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 8; ++i) {` `  result[i] = TFPU_Add(op0[i], $AACC[i * 2], TFPU_FP32, TFPU_ABS);` `}`		
	Commit	`for (i = 0; i < 8; ++i) {` `  $AACC[i * 2] = result[i];` `}`		

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 3.140: *f16v8acc* instruction definition

f16v8acc			worker	aux
Syntax	`f16v8acc $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Half,8> op0 = {` `  pickHalf($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($aSrc0:Src0+3[3], 1)` `};` `array<Single,8> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 8; ++i) {` `  if (op0[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 8; ++i) {` `  result[i] = TFPU_Add(op0[i], $AACC[i * 2], TFPU_FP32);` `}`		
	Commit	`for (i = 0; i < 8; ++i) {` `  $AACC[i * 2] = result[i];` `}`		

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 3.141: *f16v8sqacc* instruction definition

f16v8sqacc			worker	aux
Syntax	`f16v8sqacc $a`*Src0*`:`*Src0*`+3`			
Semantics	**Prepare**	`array<Half,8> op0 = {` `  pickHalf($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($aSrc0:Src0+3[3], 1)` `};` `array<Single,8> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 8; ++i) {` `  if (op0[i].issNaN()) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`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);` `}`		
	Commit	`for (i = 0; i < 8; ++i) {` `  $AACC[i * 2] = result[i];` `}`		

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 3.142: *f32v2absadd* instruction definition

f32v2absadd			worker	aux
Syntax	`f32v2absadd $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($aSrc0:Src0+1[0]),` `  TFPU_F32FromBits($aSrc0:Src0+1[1])` `};` `array<Single,2> op2 = {` `  TFPU_F32FromBits($aSrc1:Src1+1[0]),` `  TFPU_F32FromBits($aSrc1:Src1+1[1])` `};` `array<Single,2> result;` `array<bool,2> specialCase;`		
	Except In	`for (i = 0; i < 2; ++i) {` `  op1[i] = fabs(op1[i]);` `  op2[i] = fabs(op2[i]);` `}` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NANOO);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.143: *f32v2absmax* instruction definition

f32v2absmax			worker	aux
Syntax	`f32v2absmax $a`*Dst0*`:`*Dst0+1*`, $a`*Src1*`:`*Src1+1*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  result[i] = TFPU_Max(fabs(op1[i]), fabs(op2[i]));` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.144: *f32v2add* instruction definition

f32v2add			worker	aux
Syntax	`f32v2add $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1` `f32v2add $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:B, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<Single,2> result;` `array<bool,2> specialCase;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NANOO);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.145: *f32v2aop* instruction definition

f32v2aop			worker	aux
Syntax	`f32v2aop` $a*Src0*:*Src0+1*, $a*Src1*:*Src1+1*, *enumFlags*			
Semantics	**Prepare**	`array<Single,2> op0 = {` `  TFPU_F32FromBits($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])` `};` `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], $AACC[0],` `  &fpExcpt, &result[0]);` `specialCase[1] = TFPU_DoMacPreExecute(op0[1], op1[0], $AACC[2],` `  &fpExcpt, &result[1]);` `specialCase[2] = TFPU_DoMacPreExecute(op0[0], op1[1], $AACC[4],` `  &fpExcpt, &result[2]);` `specialCase[3] = TFPU_DoMacPreExecute(op0[1], op1[1], $AACC[6],` `  &fpExcpt, &result[3]);` `$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[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[0], TFPU_FP32, rmode);` `  result[1] = TFPU_Add(mRes, $AACC[2], TFPU_FP32);` `}` `if (!specialCase[2]) {` `  Single mRes = TFPU_Mul(op0[0], op1[1], TFPU_FP32, rmode);` `  result[2] = TFPU_Add(mRes, $AACC[4], TFPU_FP32);` `}` `if (!specialCase[3]) {` `  Single mRes = TFPU_Mul(op0[1], op1[1], TFPU_FP32, rmode);` `  result[3] = TFPU_Add(mRes, $AACC[6], TFPU_FP32);` `}`		
	Commit	`$AACC[0] = result[0];` `$AACC[2] = result[1];` `$AACC[4] = result[2];` `$AACC[6] = result[3];`		

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

Function references: *TFPU_F32FromBits* , *TFPU_DoMacPreExecute* , *TFPU_IsMalign* , *TFPU_Mul* , *TFPU_Add*

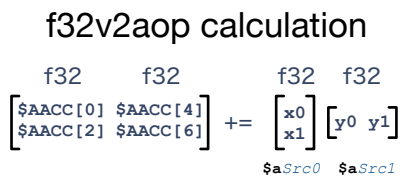


Fig. 3.23: f32v2aop

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 3.146: *f32v2axpy* instruction definition

f32v2axpy			worker	aux
Syntax	`f32v2axpy $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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 = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  Double res = a * op2[i];` `  Single mRes = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode);` `  result[i] = TFPU_Add(mRes, op1[i], TFPU_FP32);` `  }` `  // Output processing` `  if (isnan(out[i])) {` `  out[i] = TFPU_F32_QNan();` `  }` `}`		
	Except Out	`// Output exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$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])` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_DoAxpbyPreExecute* , *TFPU_RoundFP64ToFmt* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.5.6 f32v2clamp

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

Table 3.147: *f32v2clamp* instruction definition

f32v2clamp			worker	aux
Syntax	`f32v2clamp $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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])` `};`		

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 3.148: *f32v2class* instruction definition

f32v2class			worker	aux
Syntax	`f32v2class $a`*Dst0*`, $a`*Src0*`:`*Src0+1*			
Semantics	**Prepare**	`array<QuarterDataWord,4> op0;` `array<Single,2> op1 = {` `  TFPU_F32FromBits($aSrc0:Src0+1[0]),` `  TFPU_F32FromBits($aSrc0:Src0+1[1])` `};`		
	Compute	`op0 = { 0, 0, 0, 0 };` `// Denorms will have been flushed to zero` `for (i = 0; i < 2; i++) {` `  DataWord clss;` `  Single s = op1[i];` `  bool sign = signbit(s);` `  if (TFPU_F32_IsSNan(s)) {` `  clss = TFPU_CLASS_SNAN;` `  } else if (TFPU_F32_IsQNan(s)) {` `  clss = TFPU_CLASS_QNAN;` `  } else if (isinf(s)) {` `  clss = ( sign ? TFPU_CLASS_NEG_INF : TFPU_CLASS_POS_INF );` `  } else if (fabs(s) == 0.0) {` `  clss = ( sign ? TFPU_CLASS_NEG_ZERO : TFPU_CLASS_POS_ZERO );` `  } else {` `  clss = ( sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM );` `  }` `  op0[i] = clss;` `}`		
	Commit	`$aDst0 = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

Function references: *TFPU_F32FromBits* , *TFPU_F32_IsSNan* , *TFPU_F32_IsQNan*

3.7.3.5.8 f32v2cmpeq

Single-precision floating-point vector equality test

Table 3.149: *f32v2cmpeq* instruction definition

f32v2cmpeq			worker	aux
Syntax	`f32v2cmpeq $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2cmpeq $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = (rl[i] == TFPU_RELATION_EQ);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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 3.150: *f32v2cmpge* instruction definition

f32v2cmpge			worker	aux
Syntax	`f32v2cmpge $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2cmpge $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = ((rl[i] == TFPU_RELATION_EQ) || (rl[i] == TFPU_RELATION_GT));` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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 3.151: *f32v2cmpgt* instruction definition

f32v2cmpgt			worker	aux
Syntax	`f32v2cmpgt $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1` `f32v2cmpgt $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:B, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = (rl[i] == TFPU_RELATION_GT);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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 3.152: *f32v2cmple* instruction definition

f32v2cmple			worker	aux
Syntax	`f32v2cmple $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2cmple $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = ((rl[i] == TFPU_RELATION_EQ) || (rl[i] == TFPU_RELATION_LT));` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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 3.153: *f32v2cmplt* instruction definition

f32v2cmplt			worker	aux
Syntax	`f32v2cmplt $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2cmplt $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = (rl[i] == TFPU_RELATION_LT);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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 3.154: *f32v2cmpne* instruction definition

f32v2cmpne			worker	aux
Syntax	`f32v2cmpne $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2cmpne $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<TileFPRelation_t,2> rl;` `array<bool,2> result;` `for (i = 0; i < 2; i++) {` `  rl[i] = TFPU_Relation(op1[i], op2[i]);` `}`		
	Except In	`// IEEE 754-2008: 5.11, 7.2` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  if (rl[i] == TFPU_RELATION_UN) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; i++) {` `  result[i] = (rl[i] != TFPU_RELATION_EQ);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  (result[0] ? 0xffffffff : 0x00000000),` `  (result[1] ? 0xffffffff : 0x00000000)` `};`		

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:

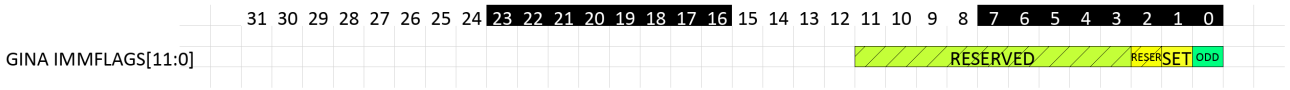


Fig. 3.24: f32v2gina immediate format

Table 3.155: *f32v2gina* instruction definition

f32v2gina			worker	aux
Syntax	`f32v2gina $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, `*zimm12*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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;`		
	Except In	`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],` `  $AACC[b+3] };` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(result, TFPU_FP32, TFPU_NO_NANOO);` `$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];` `}`		

Continued on next page

Table 3.155: *f32v2gina* instruction definition (continued)

f32v2gina			worker	aux
Syntax	`f32v2gina $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, `*zimm12*			
Semantics	**Commit cont'd**	`if (isnan(result[0])) {` `  result[0] = TFPU_F32_QNan();` `}` `if (isnan(result[1])) {` `  result[1] = TFPU_F32_QNan();` `}` `$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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*
- *f32mac example*
- *ldb16b16 example*

3.7.3.5.15 f32v2grand

Gaussian distribution, 2-element *single-precision* random vector

Table 3.156: *f32v2grand* instruction definition

f32v2grand			worker	aux
Syntax	`f32v2grand $a`*Dst0*`:`*Dst0+1*			
Semantics	**Prepare**	`array<Single,2> result;`		
	Compute	`array<uint64_t,2> randomBits;` `TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `// Calculate overall summation of outputs from successive` `// applications of xoroshiro128aox` `int16_t rsum[2] = { 0, 0 };` `for (i = 0; i < 12; i++) {` `  rsum[0] += ((randomBits[0] >> (i * 5)) & 0x1f);` `  rsum[1] += ((randomBits[1] >> (i * 5)) & 0x1f);` `}` `result[0] = rsum[0] - 186 / 32.0;` `result[1] = rsum[1] - 186 / 32.0;`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.157: *f32v2mac* instruction definition

f32v2mac			worker	aux
Syntax	`f32v2mac $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Single,2> op0 = {` `  TFPU_F32FromBits($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;`		
	Except In	`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);` `}`		
	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[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, $AACC[2], TFPU_FP32);` `}`		
	Commit	`$AACC[0] = result[0];` `$AACC[2] = result[1];`		

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 3.158: *f32v2max* instruction definition

f32v2max			worker	aux
Syntax	`f32v2max $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  result[i] = TFPU_Max(op1[i], op2[i]);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.159: *f32v2min* instruction definition

f32v2min			worker	aux
Syntax	`f32v2min $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  result[i] = TFPU_Min(op1[i], op2[i]);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.160: *f32v2mul* instruction definition

f32v2mul			worker	aux
Syntax	`f32v2mul $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:`*Src0*`+1, $a`*Src1*`:`*Src1*`+1` `f32v2mul $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`:B, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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<Single,2> result;` `array<bool,2> specialCase;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; i++) {` `  specialCase[i] =` `  TFPU_DoMulPreExecute(op1[i], op2[i], &fpExcpt, &result[i]);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  Double res = op1[i] * op2[i];` `  fpExcpt |= TFPU_GenOFLOCheck(res, TFPU_FP32, TFPU_NO_NANOO);` `  result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 $a*Src1*[16] == 1, no masking is applied (the result is a copy of the input vector)
- else if $a*Src1*[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 3.161: *f32v2rmask* instruction definition

f32v2rmask			worker	aux
Syntax	`f32v2rmask $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*			
Semantics	**Prepare**	`array<DataWord,2> op0;` `array<DataWord,2> op1 = {` `  $aSrc0:Src0+1[0],` `  $aSrc0:Src0+1[1]` `};` `DataWord op2 = $aSrc1;`		
	Compute	`bool always = ((op2 >> 16) & 1) == 1;` `bool never = (op2 & 0x1ffff) == 0;` `array<uint64_t,2> randomBits;` `TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);` `op0 = { 0, 0 };` `if (always) {` `  // No masking required` `  op0[0] = op1[0];` `  op0[1] = op1[1];` `} else if (!never) {` `  uint16_t prob = op2 & 0xffff;` `  // Mask out the 32-bit elements based on the random bit-patterns and prob` `  if (((randomBits[0] >>  0) & 0xffff) < prob) {` `  op0[0] = op1[0];` `  }` `  if (((randomBits[0] >> 32) & 0xffff) < prob) {` `  op0[1] = op1[1];` `  }` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  op0[0],` `  op0[1]` `};`		

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 3.162: *f32v2sub* instruction definition

f32v2sub			worker	aux
Syntax	`f32v2sub $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:`*Src0+1*`, $a`*Src1*`:`*Src1+1* `f32v2sub $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`:B, $a`*Src1*`:`*Src1+1*			
Semantics	**Prepare**	`array<Single,2> op1 = {` `  TFPU_F32FromBits(<($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($aSrc1:Src1+1[1])` `};` `array<Single,2> result;` `array<bool,2> specialCase;`		
	Except In	`for (i = 0; i < 2; ++i) {` `  op2[i] = -op2[i];` `}` `uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 2; ++i) {` `  specialCase[i] = TFPU_DoAddPreExecute(op1[i], op2[i], &fpExcpt,` `  &result[i]);` `}`		
	Compute	`for (i = 0; i < 2; ++i) {` `  if (!specialCase[i]) {` `  double res = static_cast<double>(op1[i]) +` `  static_cast<double>(op2[i]);` `  result[i] = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result[i], TFPU_FP32, TFPU_NO_NANOO);` `  }` `}`		
	Except Out	`$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`$aDst0:Dst0+1 = {` `  TFPU_BitsFromF32(result[0]),` `  TFPU_BitsFromF32(result[1])` `};`		

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 3.163: *f32v4absacc* instruction definition

f32v4absacc			worker	aux
Syntax	`f32v4absacc $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Single,4> op0 = {` `  TFPU_F32FromBits($aSrc0:Src0+3[0]),` `  TFPU_F32FromBits($aSrc0:Src0+3[1]),` `  TFPU_F32FromBits($aSrc0:Src0+3[2]),` `  TFPU_F32FromBits($aSrc0:Src0+3[3])` `};` `array<Single,4> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (TFPU_F32_IsSNan(op0[i])) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`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], $AACC[6], TFPU_FP32, TFPU_ABS);`		
	Commit	`$AACC[0] = result[0];` `$AACC[2] = result[1];` `$AACC[4] = result[2];` `$AACC[6] = result[3];`		

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 3.164: *f32v4acc* instruction definition

f32v4acc			worker	aux
Syntax	`f32v4acc $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Single,4> op0 = {` `  TFPU_F32FromBits($aSrc0:Src0+3[0]),` `  TFPU_F32FromBits($aSrc0:Src0+3[1]),` `  TFPU_F32FromBits($aSrc0:Src0+3[2]),` `  TFPU_F32FromBits($aSrc0:Src0+3[3])` `};` `array<Single,4> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (TFPU_F32_IsSNan(op0[i])) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`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], $AACC[6], TFPU_FP32);`		
	Commit	`$AACC[0] = result[0];` `$AACC[2] = result[1];` `$AACC[4] = result[2];` `$AACC[6] = result[3];`		

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 3.165: *f32v4sqacc* instruction definition

f32v4sqacc			worker	aux
Syntax	`f32v4sqacc $a`*Src0*`:`*Src0+3*			
Semantics	**Prepare**	`array<Single,4> op0 = {` `  TFPU_F32FromBits($aSrc0:Src0+3[0]),` `  TFPU_F32FromBits($aSrc0:Src0+3[1]),` `  TFPU_F32FromBits($aSrc0:Src0+3[2]),` `  TFPU_F32FromBits($aSrc0:Src0+3[3])` `};` `array<Single,4> result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 4; ++i) {` `  if (TFPU_F32_IsSNan(op0[i])) {` `  fpExcpt = TFPEXCPT_INV;` `  }` `}` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`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], $AACC[6], TFPU_FP32, rmode);`		
	Commit	`$AACC[0] = result[0];` `$AACC[2] = result[1];` `$AACC[4] = result[2];` `$AACC[6] = result[3];`		

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 3.166: *f32absadd* instruction definition

f32absadd			worker	aux
Syntax	`f32absadd $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single op2 = TFPU_F32FromBits($aSrc1);` `Single result;`		
	Except In	`op1 = fabs(op1);` `op2 = fabs(op2);` `uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);`		
	Compute	`if (!specialCase) {` `  double res = static_cast<double>(op1) + static_cast<double>(op2);` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`$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 3.167: *f32absmax* instruction definition

f32absmax			worker	aux
Syntax	`f32absmax $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 3.168: *f32add* instruction definition

f32add			worker	aux
Syntax	`f32add $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single op2 = TFPU_F32FromBits($aSrc1);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);`		
	Compute	`if (!specialCase) {` `  double res = static_cast<double>(op1) + static_cast<double>(op2);` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`$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*

f32add occurs in the following code examples:

- *f16v4cmac example*

3.7.3.7.4 f32clamp

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

Table 3.169: *f32clamp* instruction definition

f32clamp			worker	aux
Syntax	`f32clamp $a`*Dst0*`, $a`*Src0*`, $a`*Src1*`:`*Src1*`+1`			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `array<Single,2> op2 = {` `  TFPU_F32FromBits($aSrc1:Src1+1[0]),` `  TFPU_F32FromBits($aSrc1:Src1+1[1])` `};` `Single result;`		
	Except In	`Single ops[3] = { op1,` `  op2[0], op2[1] };` `uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 3);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Single lower = op2[0];` `Single upper = op2[1];` `// Unlike min/max, clamp explicitly propagates (regenerates)` `// NaN inputs` `if (isnan(lower) || isnan(upper)) {` `  result = TFPU_F32_QNan();` `} else {` `  if (isnan(op1)) {` `  result = TFPU_F32_QNan();` `  } else if (op1 > upper) {` `  result = upper;` `  } else {` `  result = TFPU_Max(op1, lower);` `  }` `}`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

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 3.170: *f32class* instruction definition

f32class			worker	aux
Syntax	`f32class $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`DataWord op0;` `Single op1 = TFPU_F32FromBits($aSrc0);`		
	Compute	`bool sign = signbit(op1);` `// 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 );` `}`		
	Commit	`$aDst0 = op0;`		

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 3.171: *f32cmpeq* instruction definition

f32cmpeq			worker	aux
Syntax	`f32cmpeq $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` ` EXCEPT(TEXCPT_FP);` `}`		
	Compute	`bool result = (relation == TFPU_RELATION_EQ);`		
	Commit	`$aDst0 = (result ? 0xffffffff : 0x00000000);`		

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

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.7.7 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 3.172: *f32cmpge* instruction definition

f32cmpge			worker	aux
Syntax	`f32cmpge $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`bool result = ((relation == TFPU_RELATION_GT) ||` `(relation == TFPU_RELATION_EQ));`		
	Commit	`$aDst0 = (result ? 0xffffffff : 0x00000000);`		

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

Function references: *TFPU_F32FromBits* , *TFPU_Relation* , *TFPU_IsMalign*

3.7.3.7.8 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 3.173: *f32cmpgt* instruction definition

f32cmpgt			worker	aux
Syntax	`f32cmpgt $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`bool result = (relation == TFPU_RELATION_GT);`		
	Commit	`$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 3.174: *f32cmple* instruction definition

f32cmple			worker	aux
Syntax	`f32cmple $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`bool result = ((relation == TFPU_RELATION_LT) ||` `  (relation == TFPU_RELATION_EQ));`		
	Commit	`$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 3.175: *f32cmplt* instruction definition

f32cmplt			worker	aux
Syntax	`f32cmplt $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 3.176: *f32cmpne* instruction definition

f32cmpne			worker	aux
Syntax	`f32cmpne $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`bool result = (relation != TFPU_RELATION_EQ);`		
	Commit	`$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 3.177: *f32div* instruction definition

f32div			worker	aux
Syntax	`f32div $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single op2 = TFPU_F32FromBits($aSrc1);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoDivPreExecute(op1, op2, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  Double res = op1 / op2;` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`if (TFPU_F32DivExceptIsImprecise(fpExcpt, $FP_CTL)) {` `  state.exceptImprecise = TEXCPT_FP;` `}` `$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_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 3.178: *f32exp* instruction definition

f32exp			worker	aux
Syntax	`f32exp $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoExpPreExecute(op1, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  result = TFPU_F32Exp(op1, TFPU_BASE_E, rmode);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`$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_DoExpPreExecute* , *TFPU_F32Exp* , *TFPU_GenOFLOCheck* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.14 f32exp2

Table 3.179: *f32exp2* instruction definition

f32exp2			worker	aux
Syntax	`f32exp2 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoExpPreExecute(op1, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  result = TFPU_F32Exp(op1, TFPU_BASE_2, rmode);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`$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_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 3.180: *f32int* instruction definition

f32int			worker	aux
Syntax	`f32int $a`*Dst0*`, $a`*Src1*`, `*enumRnd*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc1);` `DataWord op2 = enumRnd;`		
	Except In	`// IEEE 754-2008: 5.9` `Single ops[] = { op1 };` `uint32_t fpExcpt = TFPU_GenSNanCheck(ops, 1);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`TileRoundMode_t mode = op2 & 0x7;` `Single result = TFPU_RoundFP32ToIntegral(op1, mode);`		
	Commit	`$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 3.181: *f32ln* instruction definition

f32ln			worker	aux
Syntax	`f32ln $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 3.182: *f32log2* instruction definition

f32log2			worker	aux
Syntax	`f32log2 $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 3.183: *f32mac* instruction definition

f32mac			worker	aux
Syntax	`f32mac $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op0 = TFPU_F32FromBits($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, $azeros, 0
```

3.7.3.7.19 f32max

Determine the maximum floating-point value from two register source values.

Table 3.184: *f32max* instruction definition

f32max			worker	aux
Syntax	`f32max $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Single result = TFPU_Max(op1, op2);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

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

Function references: *TFPU_F32FromBits* , *TFPU_GenSNanCheck* , *TFPU_IsMalign* , *TFPU_Max* , *TFPU_BitsFromF32*

3.7.3.7.20 f32min

Determine the minimum floating-point value from two register source values.

Table 3.185: *f32min* instruction definition

f32min			worker	aux
Syntax	`f32min $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Compute	`Single result = TFPU_Min(op1, op2);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

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 3.186: *f32mul* instruction definition

f32mul			worker	aux
Syntax	`f32mul $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 = $FP_CTL.RND;` `if (!specialCase) {` `  Double res = op1 * op2;` `  fpExcpt |= TFPU_GenOFLOCheck(res, TFPU_FP32, TFPU_NO_NANOO);` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, rmode);` `}`		
	Except Out	`$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 3.187: *f32oorx* instruction definition

f32oorx			worker	aux
Syntax	`f32oorx $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($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 3.188: *f32oox* instruction definition

f32oox			worker	aux
Syntax	`f32oox $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoRecipPreExecute(op1, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  result = TFPU_F32Recip(op1, rmode);` `}`		
	Except Out	`$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_DoRecipPreExecute* , *TFPU_F32Recip* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.24 f32sigm

Table 3.189: *f32sigm* instruction definition

f32sigm			worker	aux
Syntax	`f32sigm $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);`		
	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);` `}`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `Single result = specialCase ? result : TFPU_F32Sigmoid(op1, rmode);`		
	Commit	`$aDst0 = TFPU_BitsFromF32(result);`		

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 **a**ccumulating **m**atrix-vector **p**roduct. Input partial-sums and result values are *single-precision*.

enumFlags format:

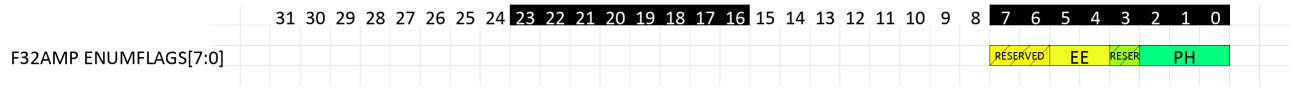


Fig. 3.25: f32sisoamp immediate format

8 output channels are processed/produced.

Table 3.190: *f32sisoamp* instruction definition

f32sisoamp			worker	aux
Syntax	`f32sisoamp $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `array<Single,2> op2 = {` `  TFPU_F32FromBits($aSrc1:Src1+1[0]),` `  TFPU_F32FromBits($aSrc1:Src1+1[1])` `};` `DataWord op3 = enumFlags;` `const unsigned ampUnits = 8;` `vector<Single> out;` `array<Single,ampUnits> weight;` `array<Single,ampUnits> result;` `array<Single,ampUnits> addend;` `array<bool,ampUnits> 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);` `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 * TREG_CCCS_WEIGHT_GROUP_SIZE) +` `  (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]);` `  }` `}`		

Continued on next page

Table 3.190: *f32sisoamp* instruction definition (continued)

f32sisoamp			worker	aux
Syntax	`f32sisoamp $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*			
Semantics	**Compute**	`for (unsigned k = 0; k < ampUnits; k++) {` `  unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  if (!specialCase[k]) {` `  vector<float> v0(1, op1), v1(1, weight[k]);` `  result[k] = TFPU_F32DotProduct(v0, v1, fp32Prec);` `  }` `  result[k] = TFPU_Add(addend[k], result[k], TFPU_FP32);` `  }` `}`		
	Except **Out**	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// 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;`		

Continued on next page

Table 3.190: *f32sisoamp* instruction definition (continued)

f32sisoamp			worker	aux
Syntax	`f32sisoamp $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*			
Semantics	**Commit cont'd**	`} 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])` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_SNanCheck* , *TFPU_DoMulPreExecute* , *TFPU_F32DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_BitsFromF32*

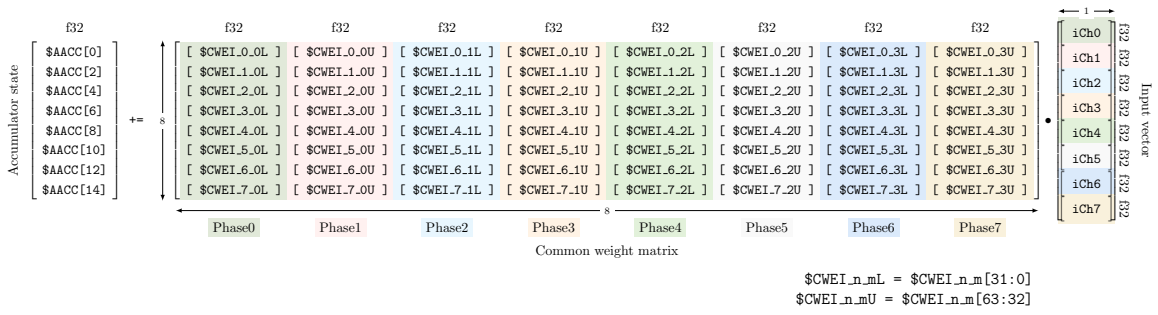


Fig. 3.26: f32sisoamp

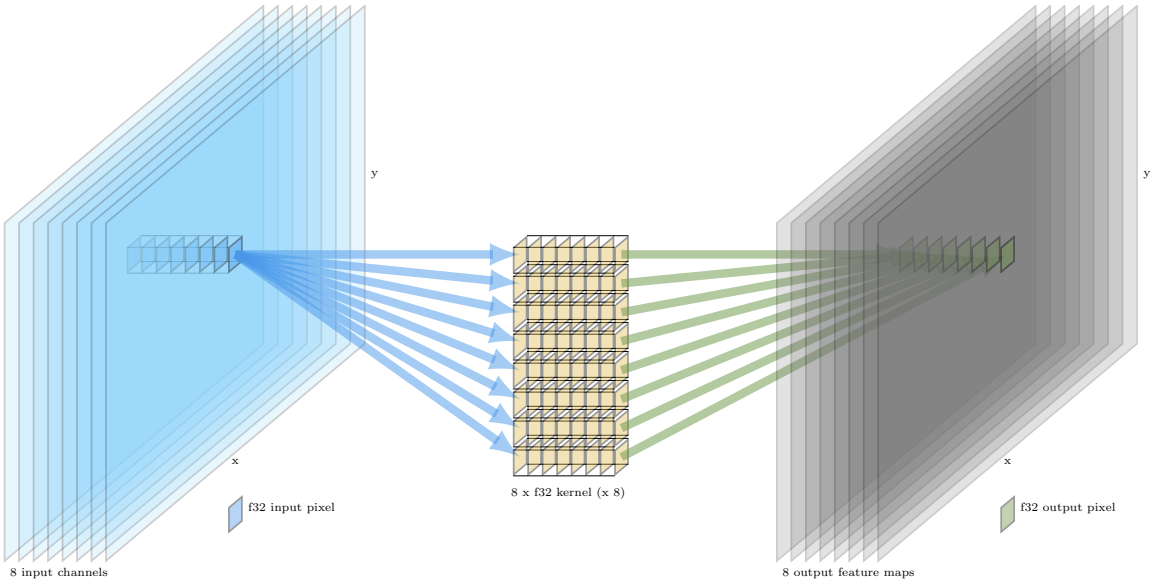


Fig. 3.27: f32sisoamp

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 intput. 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, $triPtr+=, $mzero, 0
    f32sisoamp    $azeros, $inData1, $azeros, TAMP_F32_E4_P7
```

```
  }
_loop_end:
```

3.7.3.7.26 f32sisoslic

Single-precision floating-point **sli**m **c**onvolution. Input partial-sums are *single-precision*. Results are *single-precision*.

Table 3.191: f32sisoslic, 2x1x3x2 example sequence

$a*Src0*|*1*	$AACC[14]	$AACC[10]	$AACC[6]	$AACC[2]	$AACC[12]	$AACC[8]	$AACC[4]	$AACC[0]	$a*Dst0*
x_0^L|P_0,P_1	-	$R_1 = x_0^L \times \mathbf{CW}_{5,0}{}^L + P_1$	-	-	-	$R_0 = x_0^L \times \mathbf{CW}_{4,0}{}^L + P_0$	-	-	-
x_0^U|D_0,D_1	-	$R_1 += x_0^U \times \mathbf{CW}_{5,0}{}^U$	-	-	-	$R_0 += x_0^U \times \mathbf{CW}_{4,0}{}^U$	-	-	-
x_1^L|P_2,P_3	-	$R_3 = x_1^L \times \mathbf{CW}_{5,0}{}^L + P_3$	$R_1 += x_1^L \times \mathbf{CW}_{3,0}{}^L$	-	-	$R_2 = x_1^L \times \mathbf{CW}_{4,0}{}^L + P_2$	$R_0 += x_1^L \times \mathbf{CW}_{2,0}{}^L$	-	-
x_1^U|D_2,D_3	-	$R_3 += x_1^U \times \mathbf{CW}_{5,0}{}^U$	$R_1 += x_1^U \times \mathbf{CW}_{3,0}{}^U$	-	-	$R_2 += x_1^U \times \mathbf{CW}_{4,0}{}^U$	$R_0 += x_1^U \times \mathbf{CW}_{2,0}{}^U$	-	-
x_2^L|P_4,P_5	-	$R_5 = x_2^L \times \mathbf{CW}_{5,0}{}^L + P_5$	$R_3 += x_2^L \times \mathbf{CW}_{3,0}{}^L$	$R_1 += x_2^L \times \mathbf{CW}_{1,0}{}^L$	-	$R_4 = x_2^L \times \mathbf{CW}_{4,0}{}^L + P_4$	$R_2 += x_2^L \times \mathbf{CW}_{2,0}{}^L$	$R_0 += x_2^L \times \mathbf{CW}_{0,0}{}^L$	-
x_2^U|D_4,D_5	-	$R_5 += x_2^U \times \mathbf{CW}_{5,0}{}^U$	$R_3 += x_2^U \times \mathbf{CW}_{3,0}{}^U$	$R_1 += x_2^U \times \mathbf{CW}_{1,0}{}^U$	-	$R_4 += x_2^U \times \mathbf{CW}_{4,0}{}^U$	$R_2 += x_2^U \times \mathbf{CW}_{2,0}{}^U$	$R_0 += x_2^U \times \mathbf{CW}_{0,0}{}^U$	-
x_3^L|P_6,P_7	-	$R_7 = x_3^L \times \mathbf{CW}_{5,0}{}^L + P_7$	$R_5 += x_3^L \times \mathbf{CW}_{3,0}{}^L$	$R_3 += x_3^L \times \mathbf{CW}_{1,0}{}^L$	-	$R_6 = x_3^L \times \mathbf{CW}_{4,0}{}^L + P_6$	$R_4 += x_3^L \times \mathbf{CW}_{2,0}{}^L$	$R_2 += x_3^L \times \mathbf{CW}_{0,0}{}^L$	R_0,R_1
x_3^U|D_6,D_7	-	$R_7 += x_3^U \times \mathbf{CW}_{5,0}{}^U$	$R_5 += x_3^U \times \mathbf{CW}_{3,0}{}^U$	$R_3 += x_3^U \times \mathbf{CW}_{1,0}{}^U$	-	$R_6 += x_3^U \times \mathbf{CW}_{4,0}{}^U$	$R_4 += x_3^U \times \mathbf{CW}_{2,0}{}^U$	$R_2 += x_3^U \times \mathbf{CW}_{0,0}{}^U$	-
x_4^L|P_8,P_9	-	$R_9 = x_4^L \times \mathbf{CW}_{5,0}{}^L + P_9$	$R_7 += x_4^L \times \mathbf{CW}_{3,0}{}^L$	$R_5 += x_4^L \times \mathbf{CW}_{1,0}{}^L$	-	$R_8 = x_4^L \times \mathbf{CW}_{4,0}{}^L + P_8$	$R_6 += x_4^L \times \mathbf{CW}_{2,0}{}^L$	$R_4 += x_4^L \times \mathbf{CW}_{0,0}{}^L$	R_2,R_3
x_4^U|D_8,D_9	-	$R_9 += x_4^U \times \mathbf{CW}_{5,0}{}^U$	$R_7 += x_4^U \times \mathbf{CW}_{3,0}{}^U$	$R_5 += x_4^U \times \mathbf{CW}_{1,0}{}^U$	-	$R_8 += x_4^U \times \mathbf{CW}_{4,0}{}^U$	$R_6 += x_4^U \times \mathbf{CW}_{2,0}{}^U$	$R_4 += x_4^U \times \mathbf{CW}_{0,0}{}^U$	-
x_5^L|P_{10},P_{11}	-	$R_{11} = x_5^L \times \mathbf{CW}_{5,0}{}^L + P_{11}$	$R_9 += x_5^L \times \mathbf{CW}_{3,0}{}^L$	$R_7 += x_5^L \times \mathbf{CW}_{1,0}{}^L$	-	$R_{10} = x_5^L \times \mathbf{CW}_{4,0}{}^L + P_{10}$	$R_8 += x_5^L \times \mathbf{CW}_{2,0}{}^L$	$R_6 += x_5^L \times \mathbf{CW}_{0,0}{}^L$	R_4,R_5
x_5^U|D_{10},D_{11}	-	$R_{11} += x_5^U \times \mathbf{CW}_{5,0}{}^U$	$R_9 += x_5^U \times \mathbf{CW}_{3,0}{}^U$	$R_7 += x_5^U \times \mathbf{CW}_{1,0}{}^U$	-	$R_{10} += x_5^U \times \mathbf{CW}_{4,0}{}^U$	$R_8 += x_5^U \times \mathbf{CW}_{2,0}{}^U$	$R_6 += x_5^U \times \mathbf{CW}_{0,0}{}^U$	-

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
$\mathbf{CW}_{m,n}{}^L$ is the least significant 32-bits of common weight state **$CWEI_*m*_*n***
$\mathbf{CW}_{m,n}{}^U$ 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:

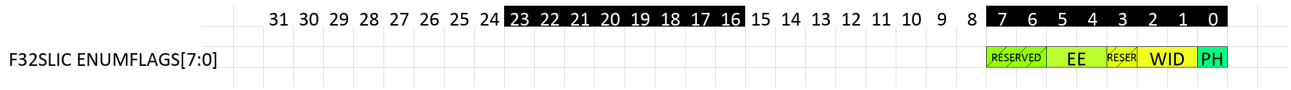


Fig. 3.28: f32sisoslic immediate format

2 output channels are processed/produced.

Table 3.192: *f32sisoslic* instruction definition

f32sisoslic			worker	aux
Syntax	`f32sisoslic $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*			
Semantics	**Prepare**	See code below		

```
Single op1 = TFPU_F32FromBits($aSrc0);
array<Single,2> op2 = {
  TFPU_F32FromBits($aSrc1:Src1+1[0]),
  TFPU_F32FromBits($aSrc1:Src1+1[1])
};
DataWord op3 = enumFlags;
const unsigned ampUnits = 8;
array<Single,ampUnits> result;
array<Single,ampUnits> incomingp;
array<Single,ampUnits> weight;
array<bool,ampUnits> specialCase;

// Extract immediate config
unsigned phase    = F32SLIC_ENUMFLAGS__PH__GET(op3);

// 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 * TREG_CCCS_WEIGHT_GROUP_SIZE) + 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] };
```

Continued on next page

Table 3.192: *f32sisoslic* instruction definition (continued)

f32sisoslic			worker	aux
Syntax	`f32sisoslic $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*			
Semantics	**Except In**	`uint32_t fpExcpt = TFPEXCPT_NONE;` `fpExcpt |= TFPU_SNanCheck(op2[0]);` `fpExcpt |= TFPU_SNanCheck(op2[1]);` `fpExcpt |= TFPU_SNanCheck(op1);` `for (unsigned u = 0; u < ampUnits; u++) {` `  unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  fpExcpt |= TFPU_SNanCheck(weight[u]);` `  specialCase[u] = TFPU_DoMulPreExecute(` `  op1, weight[u], &fpExcpt, &result[u]);` `  }` `}`		
	Compute	`for (unsigned u = 0; u < ampUnits; u++) {` `  unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  if (!specialCase[u]) {` `  vector<float> v0(1, op1), v1(1, weight[u]);` `  result[u] = TFPU_F32DotProduct(v0, v1, fp32Prec);` `  }` `  // Combine internal, incoming partial-result` `  // with result of local dot-product` `  result[u] = TFPU_Add(incomingp[u], result[u], TFPU_FP32);` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$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])` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_SNanCheck* , *TFPU_DoMulPreExecute* , *TFPU_F32DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32* , *TFPU_F32_QNan*

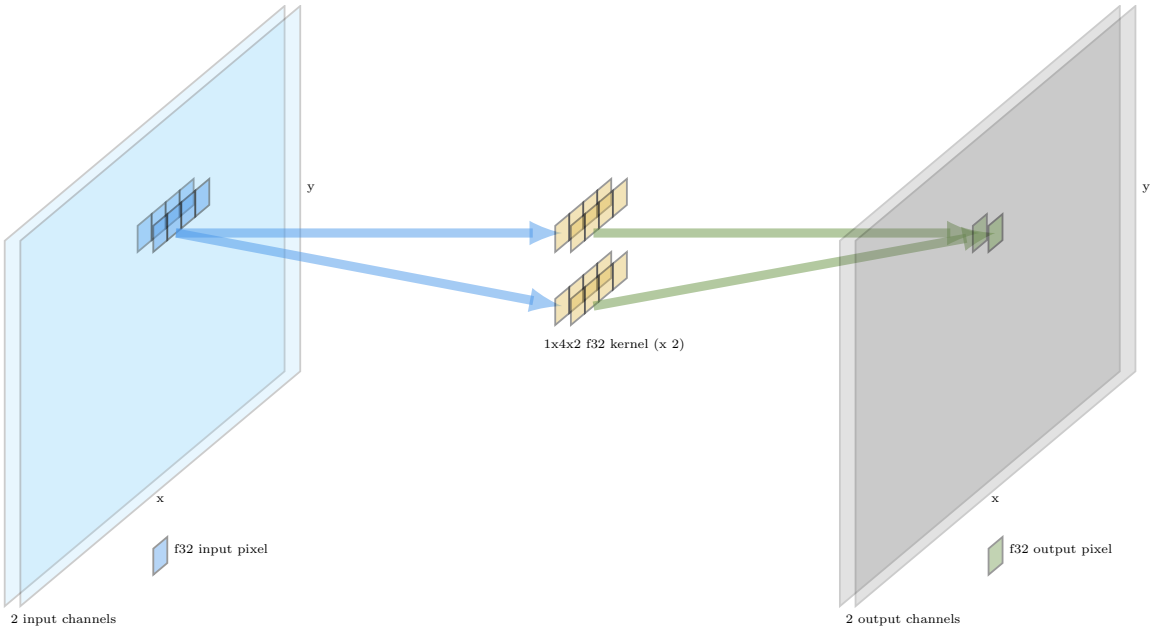


Fig. 3.29: f32sisoslic

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, $triPtr+=, $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, $triPtr+=, $mzero, 0
    f32sisoslic   $azeros, $inData1, $azeros, TSLIC_F32_1x3_W0_P1
  }
_loop_end:

  // Store out final f32v2 result (2 output channels)
  st64pace      $outPartials, $triPtr+=, $mzero, 0
```

3.7.3.7.27 f32sisov2amp

Single-precision floating-point **a**ccumulating **m**atrix-vector **p**roduct. Input partial-sums and result values are *single-precision*.

enumFlags format:

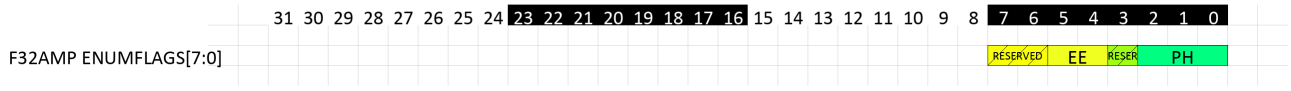


Fig. 3.30: f32sisoamp immediate format

16 output channels are processed/produced.

Table 3.193: *f32sisov2amp* instruction definition

f32sisov2amp			worker	aux
Syntax	f32sisov2amp $a*Dst0*:*Dst0+1*, $a*Src0*, $a*Src1*:*Src1+1*, *enumFlags*			
Semantics	**Prepare**	Single op1 = TFPU_F32FromBits($aSrc0); array<Single,2> op2 = { TFPU_F32FromBits($aSrc1:Src1+1[0]), TFPU_F32FromBits($aSrc1:Src1+1[1]) }; DataWord op3 = enumFlags; const unsigned ampUnits = 16; vector<Single> out; array<Single,ampUnits> input; array<Single,ampUnits> weight; array<Single,ampUnits> result; array<Single,ampUnits> addend; array<bool,ampUnits> 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 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], $AACC[baseReg+3] }; } for (unsigned k = 0; k < ampUnits; k++) { unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1; if (engineEnable[engine]) { // Weight selection unsigned set = k / TFPU_AMP_UNITS_PER_SET; unsigned lPhase = (phase - setPhaseLag[set]) & 0x7; uint64_t cwei = context.getCCCSState((k * TREG_CCCS_WEIGHT_GROUP_SIZE) + (lPhase >> 1)); weight[k] = TFPU_F32FromBits(cwei >> (32 * (lPhase & 1))); // Input selection input[k] = (set == 0) ? op1 : TFPU_F32FromBits($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)]; } } }		

Continued on next page

Table 3.193: *f32sisov2amp* instruction definition (continued)

f32sisov2amp			worker	aux
Syntax	`f32sisov2amp $a`*Dst0*`:`*Dst0*`+1, $a`*Src0*`, $a`*Src1*`:`*Src1*`+1, `*enumFlags*			
Semantics	**Except In**	`uint32_t fpExcpt = TFPEXCPT_NONE;` `fpExcpt |= TFPU_SNanCheck(op2[0]);` `fpExcpt |= TFPU_SNanCheck(op2[1]);` `fpExcpt |= TFPU_SNanCheck(op1);` `fpExcpt |= TFPU_SNanCheck(TFPU_F32FromBits($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]) {` `  vector<float> v0(1, input[k]), v1(1, weight[k]);` `  result[k] = TFPU_F32DotProduct(v0, v1, fp32Prec);` `  }` `  result[k] = TFPU_Add(addend[k], result[k], TFPU_FP32);` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP32, TFPU_NO_NANOO);` `$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];`		

Continued on next page

Table 3.193: *f32sisov2amp* instruction definition (continued)

f32sisov2amp			worker	aux
Syntax	f32sisov2amp $a*Dst0*:*Dst0+1*, $a*Src0*, $a*Src1*:*Src1+1*, *enumFlags*			
Semantics	**Commit cont'd**	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) == propagateOddAccs[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]) };		

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

Function references: *TFPU_F32FromBits* , *TFPU_SNanCheck* , *TFPU_DoMulPreExecute* , *TFPU_F32DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_F32_QNan* , *TFPU_BitsFromF32*

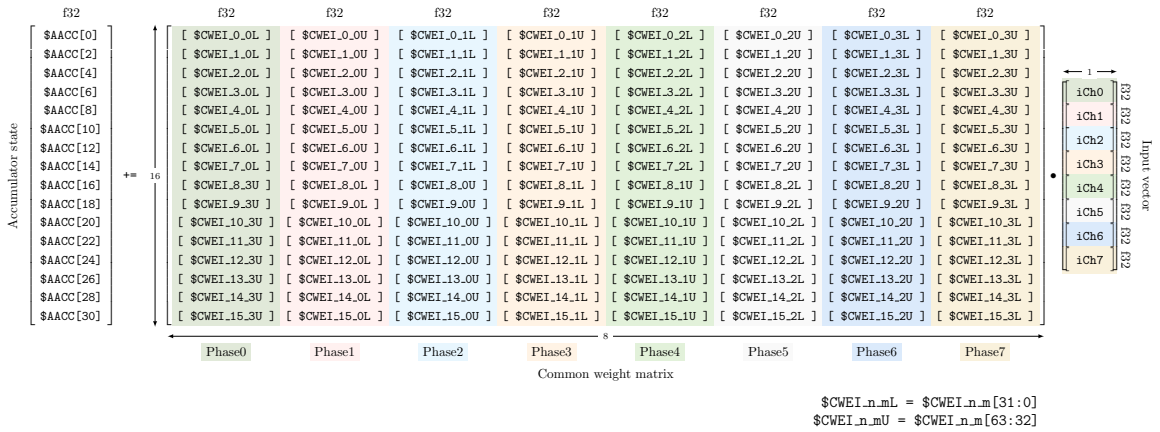


Fig. 3.31: f32sisov2amp

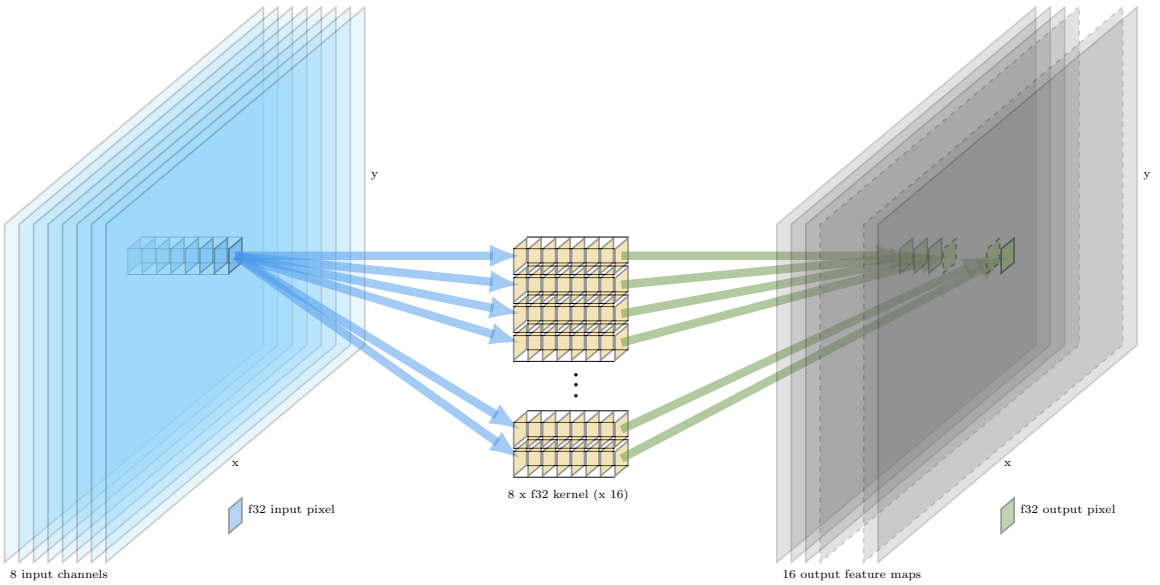


Fig. 3.32: f32sisov2amp

3.7.3.7.28 f32sisov2slic

Single-precision floating-point **sli**m **c**onvolution. Input partial-sums are *single-precision*. Results are *single-precision*.

4 output channels are processed/produced

Table 3.194: *f32sisov2slic* instruction definition

f32sisov2slic			worker	aux
Syntax	f32sisov2slic $a*Dst0*:*Dst0+1*, $a*Src0*, $a*Src1*:*Src1+1*, *enumFlags*			
Semantics	**Prepare**			

```
Single op1 = TFPU_F32FromBits($aSrc0);
array<Single,2> op2 = {
  TFPU_F32FromBits($aSrc1:Src1+1[0]),
  TFPU_F32FromBits($aSrc1:Src1+1[1])
};
DataWord op3 = enumFlags;
const unsigned ampUnits = 16;
array<Single,ampUnits> result;
array<Single,ampUnits> incomingp;
array<Single,ampUnits> input;
array<Single,ampUnits> weight;
array<bool,ampUnits> specialCase;

// AMP set arrangement
unsigned ioPhase[2]     = {0, 1};
unsigned setPhaseLag[2] = {0, 1};

// Extract immediate config
unsigned phase    = F32SLIC_ENUMFLAGS__PH__GET(op3);

// 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 * TREG_CCCS_WEIGHT_GROUP_SIZE) + wid);
    weight[u] = TFPU_F32FromBits(cwei >> (32 * lPhase));

    // Input selection
    input[u] = (set == 0) ? op1 : TFPU_F32FromBits($TAS);
```

Continued on next page

Table 3.194: *f32sisov2slic* instruction definition (continued)

f32sisov2slic		worker	aux
Syntax	`f32sisov2slic $a`*Dst0*`:`*Dst0+1*`, $a`*Src0*`, $a`*Src1*`:`*Src1+1*`, `*enumFlags*		

Semantics

Prepare cont'd

```
      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] };
```

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 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]);
    }
  }
```

Compute

```
  for (unsigned u = 0; u < ampUnits; u++) {
    unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;

    if (engineEnable[engine]) {
      if (!specialCase[u]) {
        vector<float> v0(1, input[u]), v1(1, weight[u]);
        result[u] = TFPU_F32DotProduct(v0, v1, fp32Prec);
      }

      // 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_NANOO);

  $FP_STS = $FP_STS | fpExcpt;
  if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {
    EXCEPT(TEXCPT_FP);
  }
```

Continued on next page

Table 3.194: *f32sisov2slic* instruction definition (continued)

f32sisov2slic			worker	aux
Syntax	f32sisov2slic $a*Dst0*:*Dst0+1*, $a*Src0*, $a*Src1*:*Src1+1*, *enumFlags*			
Semantics	**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];` `  }` `}` `// 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])` `};`		

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

Function references: *TFPU_F32FromBits* , *TFPU_SNanCheck* , *TFPU_DoMulPreExecute* , *TFPU_F32DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_BitsFromF32* , *TFPU_F32_QNan*

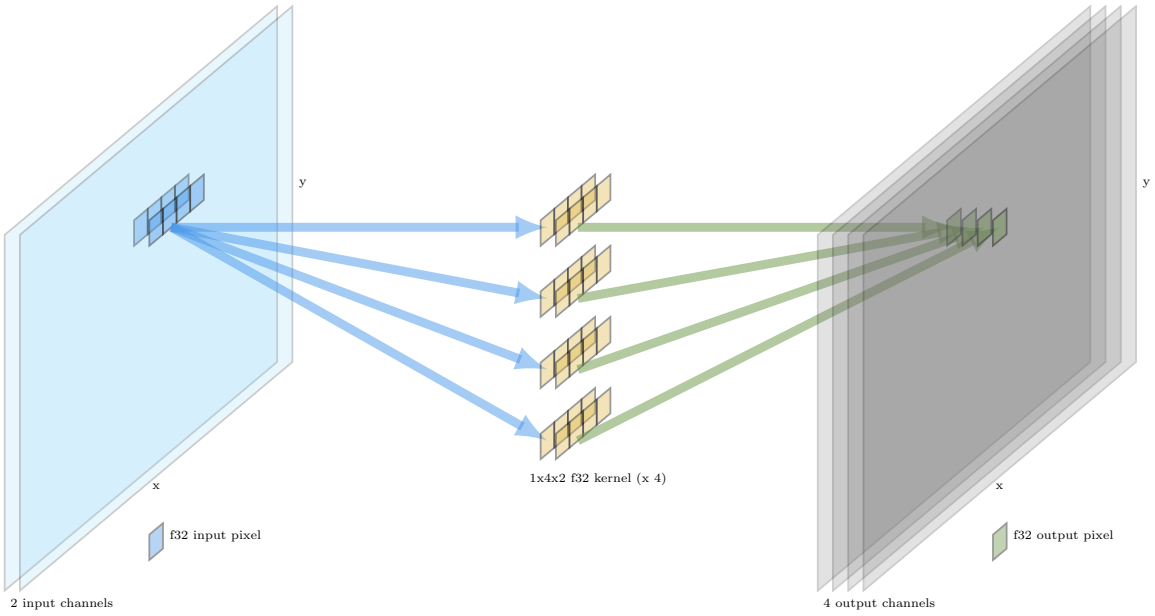


Fig. 3.33: f32sisov2slic

3.7.3.7.29 f32sqrt

Computes the square root of a single precision floating-point register source.

Table 3.195: *f32sqrt* instruction definition

f32sqrt			worker	aux
Syntax	`f32sqrt $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoSqrtPreExecute(op1, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  result = TFPU_F32Sqrt(op1, rmode);` `}`		
	Except Out	`$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_DoSqrtPreExecute* , *TFPU_F32Sqrt* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.7.30 f32sub

Subtracts two floating-point values.

Table 3.196: *f32sub* instruction definition

f32sub			worker	aux
Syntax	`f32sub $a`*Dst0*`, $a`*Src0*`, $a`*Src1*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single op2 = TFPU_F32FromBits($aSrc1);` `Single result;`		
	Except In	`op2 = -op2;` `uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoAddPreExecute(op1, op2, &fpExcpt, &result);`		
	Compute	`if (!specialCase) {` `  double res = static_cast<double>(op1) + static_cast<double>(op2);` `  result = TFPU_RoundFP64ToFmt(res, TFPU_FP32, TFPU_ROUND_EVEN);` `  fpExcpt |= TFPU_GenOFLOCheck(result, TFPU_FP32, TFPU_NO_NANOO);` `}`		
	Except Out	`$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*

3.7.3.7.31 f32tanh

Single-precision floating-point hyperbolic tangent.

Table 3.197: *f32tanh* instruction definition

f32tanh			worker	aux
Syntax	`f32tanh $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`Single op1 = TFPU_F32FromBits($aSrc0);` `Single result;`		
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `bool specialCase = TFPU_DoTanhPreExecute(op1, &fpExcpt, &result);`		
	Compute	`TileRoundMode_t rmode = $FP_CTL.RND;` `if (!specialCase) {` `  result = TFPU_F32Tanh(op1, rmode);` `}`		
	Except Out	`$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_DoTanhPreExecute* , *TFPU_F32Tanh* , *TFPU_IsMalign* , *TFPU_BitsFromF32*

3.7.3.8 f8 4-element vector

3.7.3.8.1 f8v4class

Quarter-precision 4-element floating-point vector classifier. IEEE 754-2008: 5.7.2

Table 3.198: *f8v4class* instruction definition

f8v4class			worker	aux
Syntax	`f8v4class $a`*Dst0*`, $a`*Src0*			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<QuarterDataWord,4> op0;` `array<Quart,4> op1 = {` `  pickQuart($aSrc0[0], 0, qArfFmt),` `  pickQuart($aSrc0[0], 1, qArfFmt),` `  pickQuart($aSrc0[0], 2, qArfFmt),` `  pickQuart($aSrc0[0], 3, qArfFmt)` `};`		
	Compute	`for (i = 0; i < 4; i++) {` `  DataWord clss;` `  bool sign = op1[i].sign();` `  if (op1[i].isError()) {` `  clss = TFPU_CLASS_SNAN;` `  } else if (op1[i].isZero()) {` `  clss = TFPU_CLASS_POS_ZERO;` `  } else if (op1[i].isDenorm()) {` `  clss = ( sign ? TFPU_CLASS_NEG_DENORM : TFPU_CLASS_POS_DENORM );` `  } else {` `  clss = ( sign ? TFPU_CLASS_NEG_NORM : TFPU_CLASS_POS_NORM );` `  }` `  op0[i] = clss;` `}`		
	Commit	`$aDst0 = {` `  ((op0[0] & 0xff) << 0) |` `  ((op0[1] & 0xff) << 8) |` `  ((op0[2] & 0xff) << 16) |` `  ((op0[3] & 0xff) << 24)` `};`		

Architectural state references: *$FP_NFMT*

3.7.3.9 f8 8-element vector

3.7.3.9.1 f8v8hihov4amp

Quarter-precision floating-point vector accumulating matrix-vector product. Input partial-sum and result values are *half-precision*.

Table 3.199: *f8v8hihov4amp* instruction definition

f8v8hihov4amp			worker	aux
Syntax	`f8v8hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	

```
qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;
array<Quart,8> op1 = {
  pickQuart($aSrc0:Src0+1[0], 0, qArfFmt),
  pickQuart($aSrc0:Src0+1[0], 1, qArfFmt),
  pickQuart($aSrc0:Src0+1[0], 2, qArfFmt),
  pickQuart($aSrc0:Src0+1[0], 3, qArfFmt),
  pickQuart($aSrc0:Src0+1[1], 0, qArfFmt),
  pickQuart($aSrc0:Src0+1[1], 1, qArfFmt),
  pickQuart($aSrc0:Src0+1[1], 2, qArfFmt),
  pickQuart($aSrc0:Src0+1[1], 3, qArfFmt)
};
array<Half,4> op2 = {
  pickHalf($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;
const unsigned ampUnits = 16;
bool nanoo = $FP_CTL.NANOO;
array<uint64_t,2> randomBits;
vector<Single> out;
array<vector<Single>,ampUnits> weights;
array<Single,ampUnits> resultEven;
array<Single,ampUnits> resultOdd;
int wBias;

// 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

unsigned aaccPerSet = TFPU_AMP_UNITS_PER_SET * TFPU_AACC_PER_AMP_UNIT;

// Output from even accumulators on phase 0, otherwise use odd accumulators
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],
                          op1[4], op1[5], op1[6], op1[7] };
```

Continued on next page

Table 3.199: *f8v8hihov4amp* instruction definition (continued)

f8v8hihov4amp			worker	aux
Syntax	`f8v8hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Except In**	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 8; ++i) {` `  if (op1[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `vector<Single> ops = { op2[0], op2[1], op2[2], op2[3] };` `fpExcpt |= TFPU_GenSNanCheck(ops.data(), ops.size());` `// Weight operands` `qfmt_t cweiFmt = static_cast<qfmt_t>($FP_NFMT.CWEI_FMT);` `for (int k = 0; k < ampUnits; k++) {` `  unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  // Weight selection - all fp8` `  uint64_t fullCwei =` `  context.getCCCSState((k * TREG_CCCS_WEIGHT_GROUP_SIZE) + phase);` `  uint32_t *cwei = & fullCwei;` `  vector<Quart> quarts = {` `  pickQuart(cwei[0], 0, cweiFmt),` `  pickQuart(cwei[0], 1, cweiFmt),` `  pickQuart(cwei[0], 2, cweiFmt),` `  pickQuart(cwei[0], 3, cweiFmt),` `  pickQuart(cwei[1], 0, cweiFmt),` `  pickQuart(cwei[1], 1, cweiFmt),` `  pickQuart(cwei[1], 2, cweiFmt),` `  pickQuart(cwei[1], 3, cweiFmt) };` `  for (i = 0; i < 8; ++i) {` `  if (quarts[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `  wBias = quarts[0].bias();` `  weights[k] = { quarts[0], quarts[1], quarts[2], quarts[3],` `  quarts[4], quarts[5], quarts[6], quarts[7] };` `  }` `}`		
	Compute	`int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `for (int 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]) {` `  // 8-element dot-product` `  resultEven[k] = TFPU_F8DotProduct(weights[k], inputs,` `  wBias, op1[0].bias(), scale);` `  Single incomingp;`		

Continued on next page

Table 3.199: *f8v8hihov4amp* instruction definition (continued)

f8v8hihov4amp			worker	aux
Syntax	`f8v8hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Compute cont'd**	`if (0 == phase) {` `  // Even accumulators -` `  //   combine incoming partial-sum (currently stored in our` `  //   odd-accumulator) with dot-product result` `  resultEven[k] =` `  TFPU_Add($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 = $AACC[(k + 2) * 2];` `  } else {` `  // Engines behind me are disabled (or I am the final` `  // engine in the chain). Use new partial-sum inputs` `  Half partialIn = (float)(op2[partialIndex]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  } else { // phase != 0` `  // Even accumulators -` `  //   accumulate dot-product result` `  resultEven[k] = TFPU_Add($AACC[(k * 2)], resultEven[k], TFPU_FP32);` `  // Odd accumulators -` `  //   propagate partial-sum inputs` `  if (engineEnable[engine + 1]) {` `  // Engine behind me is enabled` `  incomingp = $AACC[((k + 2) * 2) + 1];` `  } else {` `  // Engines behind me are disabled (or I am the final` `  // engine in the chain). Use new partial-sum inputs` `  Half partialIn = (float)(op2[partialIndex]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  }` `  resultOdd[k] = incomingp;` `  }` `}`		
	Except Out	`// Floating-point result exception check` `fpExcpt |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		

Continued on next page

Table 3.199: *f8v8hihov4amp* instruction definition (continued)

f8v8hihov4amp			worker	aux
Syntax	`f8v8hihov4amp $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Commit**	`for (int k = 0; k < ampUnits; k++) {` `  unsigned engine = (k & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  $AACC[(k * 2)]  = resultEven[k];` `  $AACC[(k * 2) + 1] = resultOdd[k];` `  }` `}` `for (i = 0; i < 4; i++) {` `  if (isinf(out[i]) \|\| isnan(out[i])) {` `  out[i] = TFPU_F32_QNan();` `  }` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

Architectural state references: *$FP_NFMT* , *$FP_CTL* , *$AACC* , *$FP_SCL* , *$FP_STS*

Function references: *TFPU_GenSNanCheck* , *TFPU_F8DotProduct* , *TFPU_Add* , *TFPU_F32_QNan* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

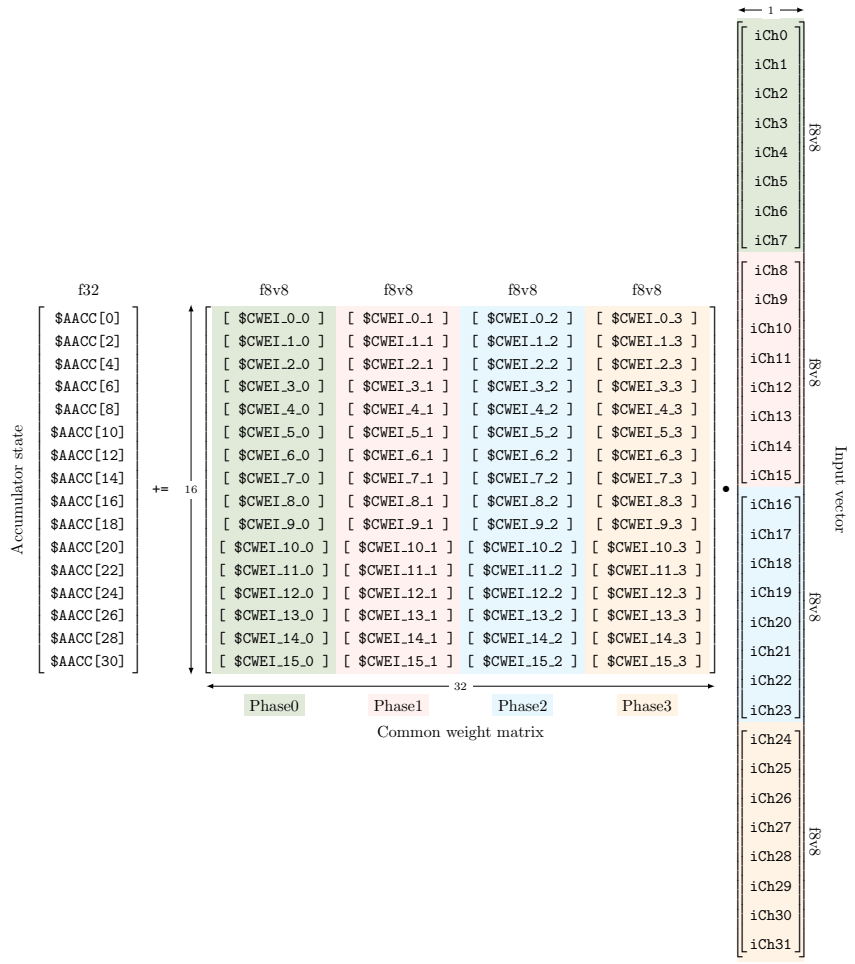


Fig. 3.34: f8v8hihov4amp

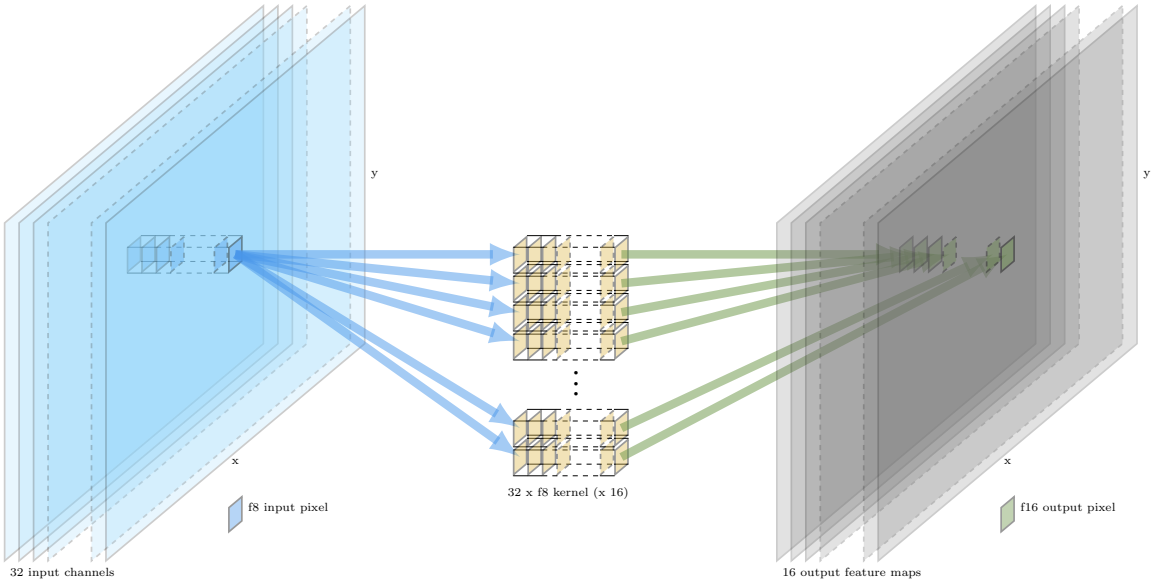


Fig. 3.35: f8v8hihov4amp

3.7.3.9.2 f8v8hihov4slic

Quarter-precision floating-point vector slim convolution. Input and result partial-sums are *half-precision* values.

Table 3.200: *f8v8hihov4slic* instruction definition

f8v8hihov4slic			worker	aux
Syntax	`f8v8hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Prepare**	`qfmt_t qArfFmt = $FP_NFMT.ARF_FMT;` `array<Quart,8> op1 = {` `  pickQuart($aSrc0:Src0+1[0], 0, qArfFmt),` `  pickQuart($aSrc0:Src0+1[0], 1, qArfFmt),` `  pickQuart($aSrc0:Src0+1[0], 2, qArfFmt),` `  pickQuart($aSrc0:Src0+1[0], 3, qArfFmt),` `  pickQuart($aSrc0:Src0+1[1], 0, qArfFmt),` `  pickQuart($aSrc0:Src0+1[1], 1, qArfFmt),` `  pickQuart($aSrc0:Src0+1[1], 2, qArfFmt),` `  pickQuart($aSrc0:Src0+1[1], 3, qArfFmt)` `};` `array<Half,4> op2 = {` `  pickHalf($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;` `const unsigned ampUnits = 16;` `bool nanoo = $FP_CTL.NANOO;` `array<uint64_t,2> randomBits;` `array<vector<Single>,ampUnits> weights;` `array<Double,ampUnits> result;` `int wBias;` `// Extract immediate config` `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], $AACC[aaccPerSet + 2] };` `vector<Single> inputs = { op1[0], op1[1], op1[2], op1[3],` `  op1[4], op1[5], op1[6], op1[7] };`
	Except In	`uint32_t fpExcpt = TFPEXCPT_NONE;` `for (i = 0; i < 8; ++i) {` `  if (op1[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `vector<Single> ops = { op2[0], op2[1], op2[2], op2[3] };` `fpExcpt |= TFPU_GenSNanCheck(ops.data(), ops.size());` `// Weight set selection` `unsigned wid = F16SLIC_ENUMFLAGS__WID__GET(op3);` `qfmt_t cweiFmt = static_cast<qfmt_t>($FP_NFMT.CWEI_FMT);` `for (unsigned u = 0; u < ampUnits; u++) {` `  unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;`		

Continued on next page

Table 3.200: *f8v8hihov4slic* instruction definition (continued)

f8v8hihov4slic			worker	aux
Syntax	`f8v8hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Except In cont'd**	`if (engineEnable[engine]) {` `  uint64_t fullCwei =` `  context.getCCCSState((u * TREG_CCCS_WEIGHT_GROUP_SIZE) + wid);` `  uint32_t *cwei = & fullCwei;` `  vector<Quart> quarts = {` `  pickQuart(cwei[0], 0, cweiFmt),` `  pickQuart(cwei[0], 1, cweiFmt),` `  pickQuart(cwei[0], 2, cweiFmt),` `  pickQuart(cwei[0], 3, cweiFmt),` `  pickQuart(cwei[1], 0, cweiFmt),` `  pickQuart(cwei[1], 1, cweiFmt),` `  pickQuart(cwei[1], 2, cweiFmt),` `  pickQuart(cwei[1], 3, cweiFmt) };` `  for (i = 0; i < 8; ++i) {` `  if (quarts[i].isError()) {` `  fpExcpt |= TFPEXCPT_INV;` `  }` `}` `  wBias = quarts[0].bias();` `  weights[u] = { quarts[0], quarts[1], quarts[2], quarts[3],` `  quarts[4], quarts[5], quarts[6], quarts[7] };` `  }` `}`		
	Compute	`int scale = Tile_SignExtend($FP_SCL.SCALE, CSR_W_FP_SCL__SCALE__SIZE);` `// 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` `  // Half-precision partials, we'll consume 2 per AMP set` `  unsigned set = u / TFPU_AMP_UNITS_PER_SET;` `  unsigned partialIndex = (set * 2) + (u & 1);` `  Half partialIn = (float)(op2[partialIndex]);` `  if (partialIn.isNaN()) {` `  incomingp = TFPU_F32_QNan();` `  } else {` `  incomingp = (Single)partialIn;` `  }` `  }` `  // 8-element dot-product` `  result[u] = TFPU_F8DotProduct(weights[u], inputs,` `  wBias, op1[0].bias(), scale);`		

Continued on next page

Table 3.200: *f8v8hihov4slic* instruction definition (continued)

f8v8hihov4slic			worker	aux
Syntax	`f8v8hihov4slic $aDst0:Dst0+1, $aSrc0:Src0+1, $aSrc1:Src1+1, enumFlags`			
Semantics	**Compute cont'd**	`    // Combine internal, incoming partial-result` `    // with result of local dot-product` `    result[u] = TFPU_Add(incomingp, result[u], TFPU_FP32);` `  }` `}`		
	Except Out	`fpExcept |= TFPU_AACCReadFlags(out, TFPU_FP16, nanoo);` `$FP_STS = $FP_STS | fpExcpt;` `if (TFPU_IsMalign(fpExcpt, $FP_CTL)) {` `  EXCEPT(TEXCPT_FP);` `}`		
	Commit	`// Internal state updates` `for (unsigned u = 0; u < ampUnits; u++) {` `  unsigned engine = (u & (TFPU_AMP_UNITS_PER_SET - 1)) >> 1;` `  if (engineEnable[engine]) {` `  $AACC[u * 2] = result[u];` `  }` `}` `for (unsigned i = 0; i < 4; i++) {` `  if (isinf(out[i]) \|\| isnan(out[i])) {` `  out[i] = TFPU_F32_QNan();` `  }` `}` `HalfSaturationMode smode = TFPU_GetNanooMode(nanoo);` `$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)` `};`		

Architectural state references: *$FP_NFMT* , *$FP_CTL* , *$AACC* , *$FP_SCL* , *$FP_STS*

Function references: *TFPU_GenSNanCheck* , *TFPU_F32_QNan* , *TFPU_F8DotProduct* , *TFPU_Add* , *TFPU_AACCReadFlags* , *TFPU_IsMalign* , *TFPU_GetNanooMode* , *Tile_SignExtend*

3.7.4 Integer

Table 3.201: integer instructions summary

Mnemonic	Super?	Worker?	main?	aux?	Brief
abs	✓	✓	✓	✗	Absolute value of signed 32-bit integer
add	✓	✓	✓	✗	Integer addition
cmpeq	✓	✓	✓	✗	Equality test
cmpne	✓	✓	✓	✗	Inequality test
cmpslt	✓	✓	✓	✗	Signed less-than test
cmpult	✓	✓	✓	✗	Unsigned less-than test
max	✓	✓	✓	✗	Maximum
min	✓	✓	✓	✗	Minimum
movz	✓	✓	✓	✗	Conditional move
mul	✓	✓	✓	✗	Signed multiplication
shl	✓	✓	✓	✗	Logical shift left
shr	✓	✓	✓	✗	Logical shift right
shrs	✓	✓	✓	✗	Signed (arithmetic) shift right
sub	✓	✓	✓	✗	Subtraction
tapack	✗	✓	✓	✗	Triple address pack
urand32	✗	✓	✗	✓	Uniform distribution, 32-bit random integer
urand64	✗	✓	✗	✓	Uniform distribution, 64-bit random integer

3.7.4.1 abs

Absolute value of signed 32-bit integer.

Table 3.202: *abs* instruction definition

abs			both	main
Syntax	`abs $m`*Dst0*`, $m`*Src0*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `SignedDataWord result;`		
	Compute	`if (op1 == INT32_MIN) {` `  // Result cannot be represented as a signed int` `  result = 0;` `} else if (op1 < 0) {` `  // op1 is negative` `  result = -op1;` `} else {` `  // op1 is positive` `  result = op1;` `}`		
	Commit	`$mDst0 = result;`		

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 3.203: *add* instruction definition

add			both	main
Syntax	add $m*Dst0*, $m*Src0*, $m*Src1* add $m*Dst0*, $m*Src0*, *simm16* add $m*Dst0*, $m*Src0*, *zimm16*			
Semantics	**Prepare**	SignedDataWord op1 = $mSrc0; SignedDataWord op2 = <($mSrc1, zimm16, Tile_SignExtend(simm16, 16))>;		
	Compute	SignedDataWord result = op1 + op2;		
	Commit	$mDst0 = result;		

Function references: *Tile_SignExtend*

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 3.204: *cmpeq* instruction definition

cmpeq			both	main
Syntax	cmpeq $m*Dst0*, $m*Src0*, $m*Src1* cmpeq $m*Dst0*, $m*Src0*, *simm16* cmpeq $m*Dst0*, $m*Src0*, *zimm16*			
Semantics	**Prepare**	DataWord op1 = $mSrc0; DataWord op2 = <($mSrc1, zimm16, Tile_SignExtend(simm16, 16))>; DataWord result;		
	Compute	if (op1 == op2) { result = 0x1; } else { result = 0x0; }		
	Commit	$mDst0 = result;		

Function references: *Tile_SignExtend*

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 3.205: *cmpne* instruction definition

cmpne			both	main
Syntax	`cmpne $m`*Dst0*`, $m`*Src0*`, $m`*Src1*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;` `DataWord op2 = $mSrc1;` `DataWord result;`		
	Compute	`if (op1 != op2) {` `  result = 0x1;` `} else {` `  result = 0x0;` `}`		
	Commit	`$mDst0 = result;`		

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 3.206: *cmpslt* instruction definition

cmpslt			both	main
Syntax	`cmpslt $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `cmpslt $m`*Dst0*`, $m`*Src0*`, `*simm16*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `SignedDataWord op2 = <($mSrc1, Tile_SignExtend(simm16, 16))>;` `DataWord result;`		
	Compute	`if (op1 < op2) {` `  result = 1;` `} else {` `  result = 0;` `}`		
	Commit	`$mDst0 = result;`		

Function references: *Tile_SignExtend*

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. to 0. The comparison operation is **unsigned**.

Table 3.207: *cmpult* instruction definition

cmpult			both	main
Syntax	`cmpult $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `cmpult $m`*Dst0*`, $m`*Src0*`, `*zimm16*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;` `DataWord op2 = <($mSrc1, zimm16)>;` `DataWord result;`		
	Compute	`if (op1 < op2) {` `  result = 1;` `} else {` `  result = 0;` `}`		
	Commit	`$mDst0 = result;`		

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 3.208: *max* instruction definition

max			both	main
Syntax	`max $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `max $m`*Dst0*`, $m`*Src0*`, `*simm16* `max $m`*Dst0*`, $m`*Src0*`, `*zimm16*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `SignedDataWord op2 = <($mSrc1, zimm16, Tile_SignExtend(simm16, 16))>;` `SignedDataWord result;`		
	Compute	`if (op1 > op2) {` `  result = op1;` `} else {` `  result = op2;` `}`		
	Commit	`$mDst0 = result;`		

Function references: *Tile_SignExtend*

3.7.4.8 min

Select the minimum of 2 signed integer values. Immediates may be *sign extended*, *zero extended* or *zero tailed*.

Table 3.209: *min* instruction definition

min			both	main
Syntax	`min $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `min $m`*Dst0*`, $m`*Src0*`, `*simm16* `min $m`*Dst0*`, $m`*Src0*`, `*zimm16*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `SignedDataWord op2 = <($mSrc1, zimm16, Tile_SignExtend(simm16, 16))>;` `SignedDataWord result;`		
	Compute	`if (op1 < op2) {` `  result = op1;` `} else {` `  result = op2;` `}`		
	Commit	`$mDst0 = result;`		

Function references: *Tile_SignExtend*

3.7.4.9 movz

Conditional copy of one register into another, gated on the value of a third.

Table 3.210: *movz* instruction definition

movz			both	main
Syntax	`movz $m`*SrcDst0*`, $m`*Src0*`, $m`*Src1*			
Semantics	**Prepare**	`DataWord op2 = $mSrc1;` `DataWord op1 = $mSrcDst0;` `DataWord op0 = $mSrc0;`		
	Compute	`DataWord result = ( (op0 != 0) ? op2 : op1);`		
	Commit	`$mSrcDst0 = 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 3.211: *mul* instruction definition

mul			both	main
Syntax	`mul $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `mul $m`*Dst0*`, $m`*Src0*`, `*simm16*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `SignedDataWord op2 = <($mSrc1, Tile_SignExtend(simm16, 16))>;`		
	Compute	`// 32 x 32 -> 32` `DataWord result = op1 * op2;`		
	Commit	`$mDst0 = result;`		

Function references: *Tile_SignExtend*

3.7.4.11 shl

Perform a logical left shift, of up-to 31-bits, on a register value.

Table 3.212: *shl* instruction definition

shl			both	main
Syntax	`shl $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `shl $m`*Dst0*`, $m`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;` `DataWord op2 = <($mSrc1, zimm12)>;`		
	Compute	`DataWord result = op1 << (op2 % 32);`		
	Commit	`$mDst0 = 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 3.213: *shr* instruction definition

shr			both	main
Syntax	`shr $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `shr $m`*Dst0*`, $m`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op1 = $mSrc0;` `DataWord op2 = <($mSrc1, zimm12)>;`		
	Compute	`DataWord result = op1 >> (op2 % 32);`		
	Commit	`$mDst0 = 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 3.214: *shrs* instruction definition

shrs			both	main
Syntax	`shrs $m`*Dst0*`, $m`*Src0*`, $m`*Src1* `shrs $m`*Dst0*`, $m`*Src0*`, `*zimm12*			
Semantics	**Prepare**	`SignedDataWord op1 = $mSrc0;` `DataWord op2 = <($mSrc1, zimm12)>;`		
	Compute	`SignedDataWord result = op1 >> (op2 % 32);`		
	Commit	`$mDst0 = result;`		

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 3.215: *sub* instruction definition

sub			both	main
Syntax	sub $m*Dst0*, $m*Src1*, $m*Src0* sub $m*Dst0*, *simm16*, $m*Src0* sub $m*Dst0*, *zimm16*, $m*Src0*			
Semantics	**Prepare**	DataWord op1 = $mSrc0; DataWord op2 = <($mSrc1, zimm16, Tile_SignExtend(simm16, 16))>;		
	Compute	DataWord result = op2 - op1;		
	Commit	$mDst0 = result;		

Function references: *Tile_SignExtend*

3.7.4.15 tapack

Convert 3 absolute addresses to the triple-packed address format.

Table 3.216: *tapack* instruction definition

tapack			worker	main
Syntax	tapack $m*Dst0*:*Dst0+1*, $m*Addr0*, $m*Addr1*, $m*Addr2*			
Semantics	**Prepare**	DataWord op0 = $mAddr0; DataWord op1 = $mAddr1; DataWord op2 = $mAddr2; array<DataWord,3> addrs;		
	Compute	addrs[0] = op0 & TMEM_FULL_ADDRESS_MASK; addrs[1] = op1 & TMEM_FULL_ADDRESS_MASK; addrs[2] = op2 & TMEM_FULL_ADDRESS_MASK;		
	Commit	// MRF result values $mDst0:Dst0+1 = { Tile_TripleAddressPack_Lower(addrs), Tile_TripleAddressPack_Upper(addrs) };		

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 3.217: *urand32* instruction definition

urand32			worker	aux
Syntax	`urand32 $a`*Dst0*			
Semantics	**Prepare**	`array<uint64_t,2> randomBits;`		
	Compute	`TPRNG_Advance($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 3.218: *urand64* instruction definition

urand64			worker	aux
Syntax	`urand64 $a`*Dst0*`:`*Dst0+1*			
Semantics	**Prepare**	`array<uint64_t,2> randomBits;`		
	Compute	`TPRNG_Advance($PRNG_0_0, $PRNG_0_1, $PRNG_1_0, $PRNG_1_1, randomBits);`		
	Commit	`$aDst0:Dst0+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 3.219: memory instructions summary

Mnemonic	Super?	Worker?	main?	aux?	Brief
atom	✗	✓	✓	✗	Copy an *arf* register value to *mrf*
ld128	✗	✓	✓	✗	Single 128-bit load from interleaved memory region
ld128putcs	✓	✗	✓	✗	128-bit load and put to common configuration space
ld128step	✗	✓	✓	✗	Post-incrementing 128-bit load from interleaved memory region.
ld2x64pace	✗	✓	✓	✗	Post-incrementing, dual 64-bit load
ld2xst64pace	✗	✓	✓	✗	Post-incrementing dual 64-bit load with simultaneous 64-bit store.
ld32	✓	✓	✓	✗	Single 32-bit load
ld32step	✓	✓	✓	✗	Post-incrementing *word* load
ld64	✗	✓	✓	✗	Single 64-bit load
ld64a32	✗	✓	✓	✗	Post-incrementing dense 64-bit plus sparse 32-bit load
ld64a32pace	✗	✓	✓	✗	Post-incrementing dual 64/32-bit load
ld64b16pace	✗	✓	✓	✗	Post-incrementing, dual 64/16-bit load
ld64putcs	✓	✗	✓	✗	64-bit load and put to common configuration space
ld64step	✗	✓	✓	✗	Post-incrementing 64-bit load
ldb16	✗	✓	✓	✗	16-bit load and broadcast
ldb16b16	✗	✓	✓	✗	Post-incrementing, lightly-sparse 16-bit with dense 16-bit load
ldb16step	✗	✓	✓	✗	Post-incrementing 16-bit load and broadcast
ldb8	✗	✓	✓	✗	8-bit load and broadcast
ldb8step	✗	✓	✓	✗	Post-incrementing 8-bit load and broadcast
ldd16a32	✗	✓	✓	✗	Post-incrementing 16-bit delta and 32-bit data load
ldd16a64	✗	✓	✓	✗	Post-incrementing 16-bit delta and 64-bit data load
ldd16b16	✗	✓	✓	✗	Post-incrementing 16-bit delta and broadcast 16-bit data load
ldd16v2a32	✗	✓	✓	✗	Post-incrementing delta-pair plus 32-bit load
lds16	✓	✓	✓	✗	Sign extending 16-bit load
lds16step	✓	✓	✓	✗	Post-incrementing sign-extending 16-bit load
lds8	✓	✓	✓	✗	Sign extending 8-bit load
lds8step	✓	✓	✓	✗	Post-incrementing sign-extending 8-bit load
ldst64pace	✗	✓	✓	✗	Post-incrementing 64-bit load with simultaneous 64-bit store.
ldz16	✓	✓	✓	✗	Zero-extending 16-bit load
ldz16step	✓	✓	✓	✗	Post-incrementing zero-extending 16-bit load
ldz8	✓	✓	✓	✗	Zero-extending 8-bit load
ldz8step	✓	✓	✓	✗	Post-incrementing *zero extended* 8-bit load
st32	✗	✓	✓	✗	32-bit store
st32step	✓	✓	✓	✗	Post-incrementing 32-bit store

Continued on next page

Table 3.219 – continued from previous page

Mnemonic	Super?	Worker?	main?	aux?	Brief
st64	✗	✓	✓	✗	64-bit store
st64pace	✗	✓	✓	✗	Post-incrementing 64-bit store, using packed addresses and offsets
st64step	✗	✓	✓	✗	Post-incrementing 64-bit store
stm32	✓	✓	✓	✗	32-bit store from MRF
stm32step	✓	✓	✓	✗	Post-incrementing 32-bit store from MRF

3.7.5.1 Load-store

Table 3.220: Load & store by access size in bits

	Store bits
Loads signature	**64**
64	✓
64,64	✓

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 3.221: *ldst64pace* instruction definition

ldst64pace			worker	main
Syntax	`ldst64pace $aDst0:Dst0+1, $aSrc0:Src0+1, $mAddr0:Addr0+1+=, $mStride0, Strimm2x2`			
Semantics	**Prepare**	`DataWord op0 = $mStride0;` `array<DataWord,2> op1 = {` `  $mAddr0:Addr0+1[0],` `  $mAddr0:Addr0+1[1]` `};` `array<DataWord,2> op3 = {` `  $aSrc0:Src0+1[0],` `  $aSrc0:Src0+1[1]` `};` `DataWord op4 = Strimm2x2;` `array<DataWord,3> addrs;` `EA[0] = Tile_ExtractPackedAddress(op1, 0);` `EA[2] = Tile_ExtractPackedAddress(op1, 2);`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[2])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[2] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[2])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`// Post-increment both addresses as specified by the immediate` `int32_t stride0 = Tile_ExtractPackedStride(op0, op4);` `int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2));` `addrs[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[1] = Tile_ExtractPackedAddress(op1, 1); // Unchanged` `addrs[2] = (EA[2] + (stride1 * 8)) & TMEM_FULL_ADDRESS_MASK;`		
	Memory	`uint64_t storeData = (op3[1] << 32ULL) | op3[0];` `uint64_t data = loadDoubleWord(EA[0]);` `storeDoubleWord(EA[2], storeData);`		
	Commit	`$mAddr0:Addr0+1 = {` `  Tile_TripleAddressPack_Lower(addrs),` `  Tile_TripleAddressPack_Upper(addrs)` `};` `$aDst0:Dst0+1 = {` `  data & 0xffffffffULL,` `  (data >> 32ULL) & 0xffffffffULL` `};`		

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

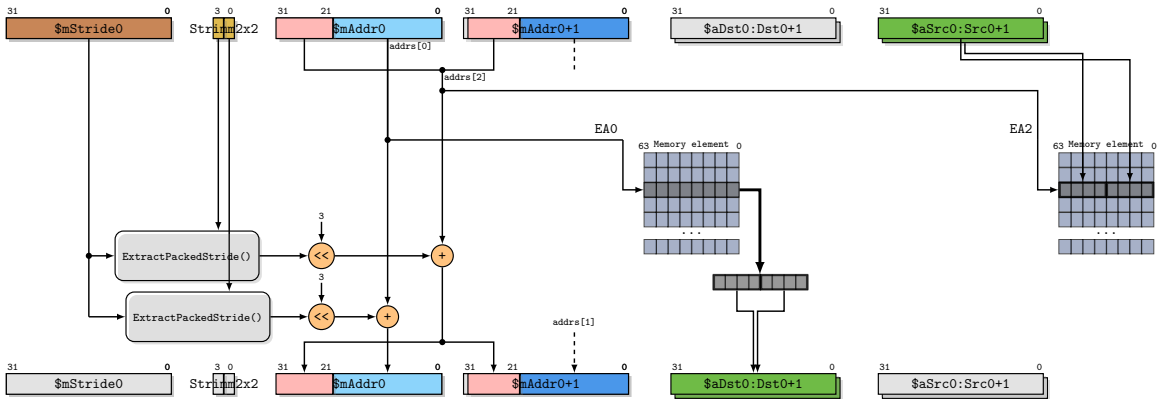


Fig. 3.36: ldst64pace

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 3.222: *ld2xst64pace* instruction definition

ld2xst64pace			worker	main
Syntax	`ld2xst64pace $aDst0:Dst0+3, $aSrc0:Src0+1, $mAddr0:Addr0+1+=, $mStride0, Strimm3x2`			
Semantics	**Prepare**	`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> addrs;` `EA[0] = Tile_ExtractPackedAddress(op1, 0);` `EA[1] = Tile_ExtractPackedAddress(op1, 1);` `EA[2] = Tile_ExtractPackedAddress(op1, 2);`		
	Except In	`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);` `}`		
	Compute	`// 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));` `addrs[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[1] = (EA[1] + (stride1 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[2] = (EA[2] + (stride2 * 8)) & TMEM_FULL_ADDRESS_MASK;`		
	Memory	`uint64_t storeData = (op3[1] << 32ULL) | op3[0];` `uint64_t data0 = loadDoubleWord(EA[0]);` `uint64_t data1 = loadDoubleWord(EA[1]);` `storeDoubleWord(EA[2], storeData);`		

Continued on next page

Table 3.222: *ld2xst64pace* instruction definition (continued)

ld2xst64pace			**worker**	**main**
Syntax	ld2xst64pace $aDst0:Dst0+3, $aSrc0:Src0+1, $mAddr0:Addr0+1+=, $mStride0, Strimm3x2			
Semantics	**Commit**	$mAddr0:Addr0+1 = { Tile_TripleAddressPack_Lower(addrs), Tile_TripleAddressPack_Upper(addrs) }; $aDst0:Dst0+3 = { data0 & 0xffffffffULL, (data0 >> 32ULL) & 0xffffffffULL, data1 & 0xffffffffULL, (data1 >> 32ULL) & 0xffffffffULL };		

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

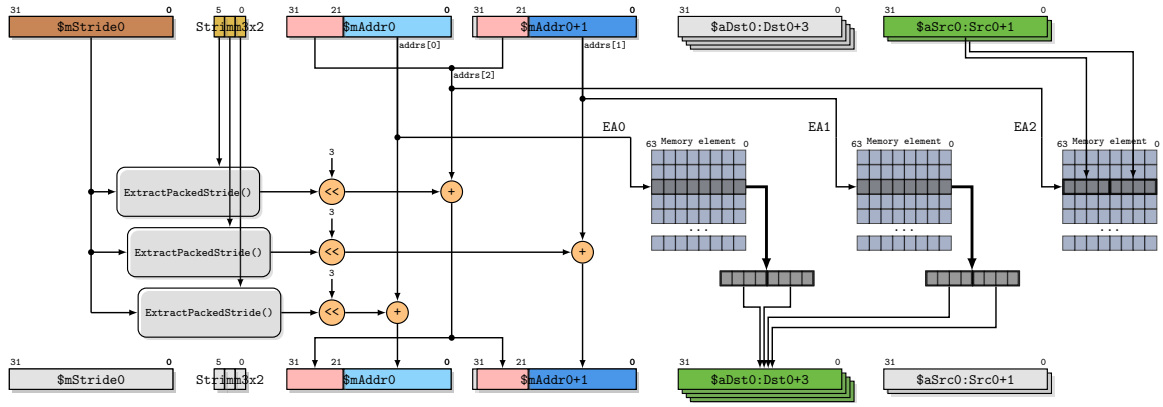


Fig. 3.37: ld2xst64pace

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

Table 3.223: Multi-load addressing modes

Index	Description	Instructions
0	instr dst0, dst1, srcDst0+=, src0, imm0	*ld2x64pace*, *ld64a32pace*, *ld64b16pace*
1	instr dst0, src0, srcDst0++, srcDst1>>	*ldb16b16*
2	instr dst0, srcDst0++, src0, src1	*ld64a32*
3	instr dst0, srcDst0++, src0, srcDst1@	*ldd16a32*, *ldd16a64*, *ldd16b16*, *ldd16v2a32*

Table 3.224: Multi-load addressing modes by access size bits

	Second access		
First access	**16**	**32**	**64**
16	1, 3		
32	3	3	2
64	0, 3	0	0

3.7.5.2.1 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 lsbs of the 4th operand.
- The mini-delta operand is also right-shifted by 4.

Table 3.225: *ldb16b16* instruction definition

ldb16b16			worker	main
Syntax	ldb16b16 $a*Dst0*:*Dst0+1*, $m*Base0*, $m*Delta0*++, $m*MiniD0*>>			
Semantics	**Prepare**	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;		
	Except In	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); }		
	Compute	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;		
	Memory	uint16_t denseData = loadHalf(EA[0]); uint16_t sparseData = loadHalf(EA[1]);		
	Commit	$mMiniD0 = nextMiniDelta; $mDelta0 = { ((nextDataOffset & 0xffff) << 0) | ((nextWeightOffset & 0xffff) << 16) }; $aDst0:Dst0+1 = { ((denseData & 0xffff) << 0) | ((denseData & 0xffff) << 16), ((sparseData & 0xffff) << 0) | ((sparseData & 0xffff) << 16) };		

Function references: *TMem_IsValidAddress*

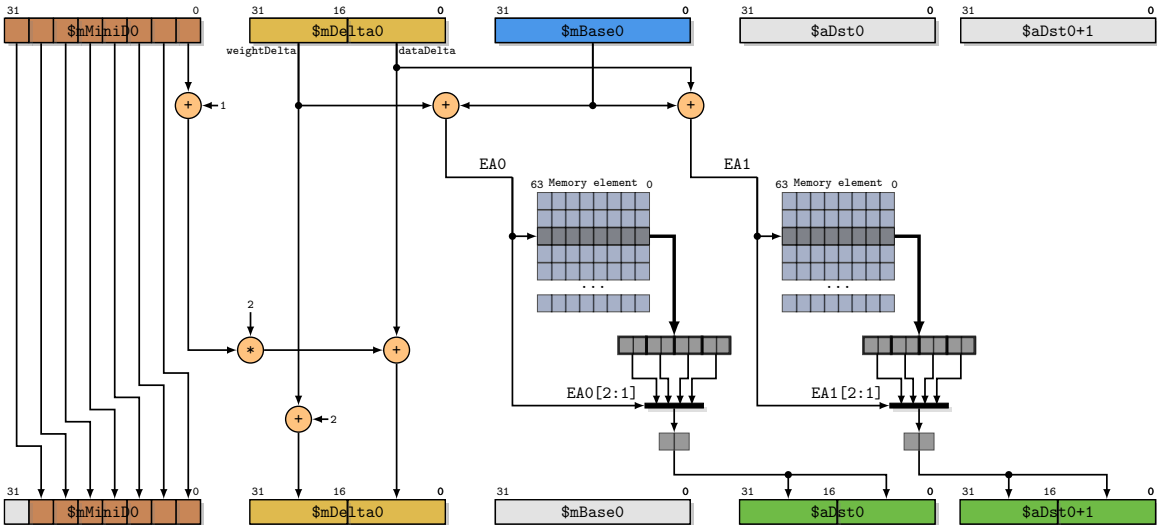


Fig. 3.38: ldb16b16

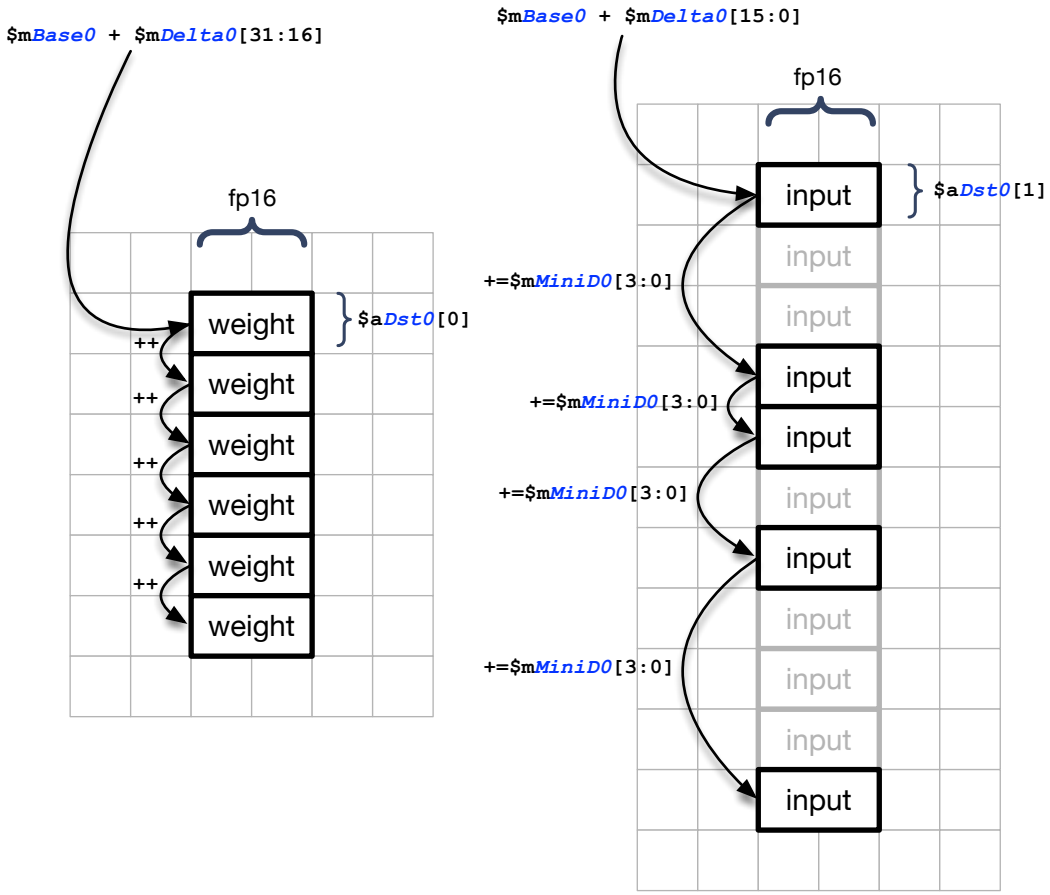


Fig. 3.39: ldb16b16

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
```

3.7.5.2.2 ldd16b16

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 3.226: *ldd16b16* instruction definition

ldd16b16			worker	main
Syntax	`ldd16b16 $a`*Dst0*`, $m`*Addr0*`++, $m`*Base0*`, $m`*Delta0*`@`			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mAddr0;` `array<HalfDataWord,2> op2 = {` `  (($mDelta0[0] >> 0) & 0xffff),` `  (($mDelta0[0] >> 16) & 0xffff)` `};` `uint16_t delta = op2[0];` `EA[0] = op0 + delta;` `EA[1] = op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x1) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x1) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextDeltaAddress = op1 + 2;`		
	Memory	`uint16_t data = loadHalf(EA[0]);` `uint16_t newDelta = loadHalf(EA[1]);`		
	Commit	`$mAddr0 = nextDeltaAddress;` `$mDelta0 = {` `  ((newDelta & 0xffff) << 0) |` `  0` `};` `$aDst0 = {` `  ((data & 0xffff) << 0) |` `  ((data & 0xffff) << 16)` `};`		

Function references: *TMem_IsValidAddress*

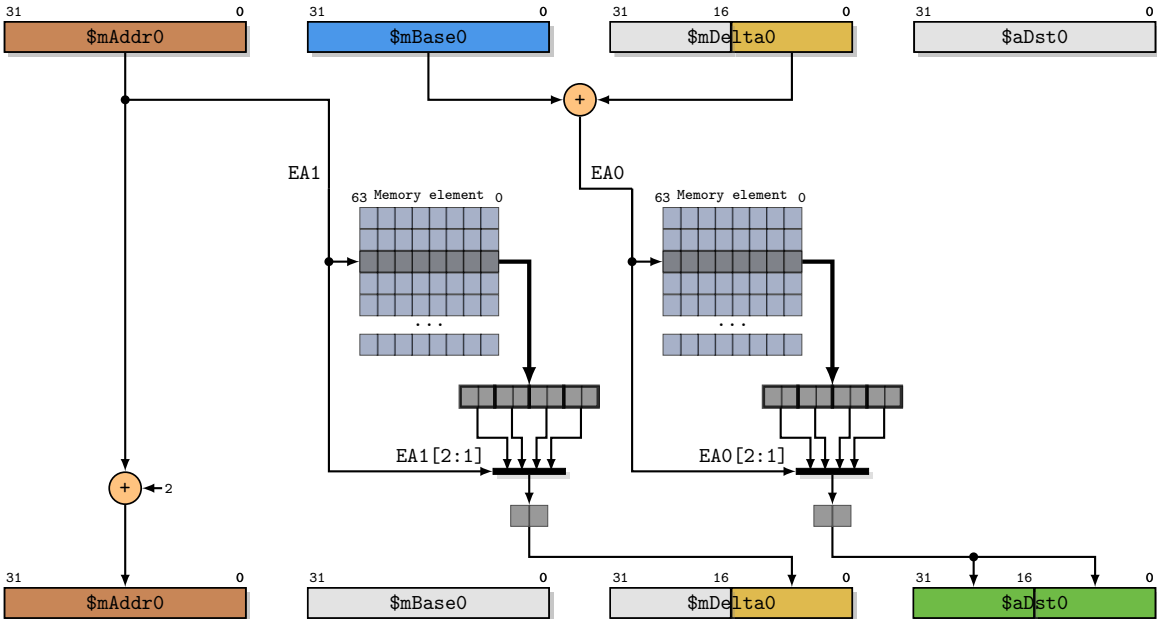


Fig. 3.40: ldd16b16

3.7.5.2.3 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 3.227: *ldd16a32* instruction definition

ldd16a32			worker	main
Syntax	`ldd16a32 $a`*Dst0*`, $m`*Addr0*`++, $m`*Base0*`, $m`*Delta0*`@`			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mAddr0;` `array<HalfDataWord,2> op2 = {` `  (($mDelta0[0] >> 0) & 0xffff),` `  (($mDelta0[0] >> 16) & 0xffff)` `};` `uint16_t delta = op2[0];` `EA[0] = op0 + delta;` `EA[1] = op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x3) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x1) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextDeltaAddress = op1 + 2;`		
	Memory	`uint32_t data = loadWord(EA[0]);` `uint16_t newDelta = loadHalf(EA[1]);`		
	Commit	`$mAddr0 = nextDeltaAddress;` `$mDelta0 = {` `  ((newDelta & 0xffff) << 0) |` `  0` `};` `$aDst0 = data;`		

Function references: *TMem_IsValidAddress*

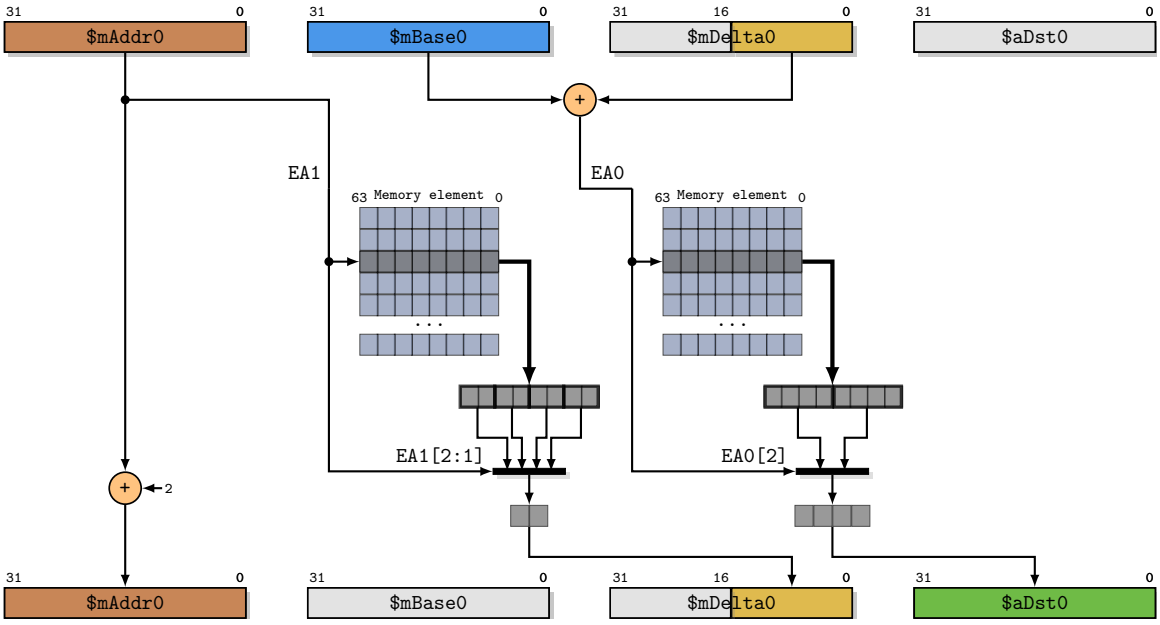


Fig. 3.41: ldd16a32

3.7.5.2.4 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 3.228: *ldd16v2a32* instruction definition

ldd16v2a32			worker	main
Syntax	`ldd16v2a32 $a`*Dst0*`, $m`*Addr0*`++, $m`*Base0*`, $m`*Delta0*`@`			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mAddr0;` `DataWord op2 = $mDelta0;` `uint16_t delta = (op2 >> 16) & 0xffff;` `EA[0] = op0 + delta;` `EA[1] = op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x3) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x3) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextAddr = op1 + 4; // Increment ptr address`		
	Memory	`uint32_t data = loadWord(EA[0]);` `uint32_t newDelta = loadWord(EA[1]);`		
	Commit	`$mAddr0 = nextAddr;` `$mDelta0 = newDelta;` `$aDst0 = data;`		

Function references: *TMem_IsValidAddress*

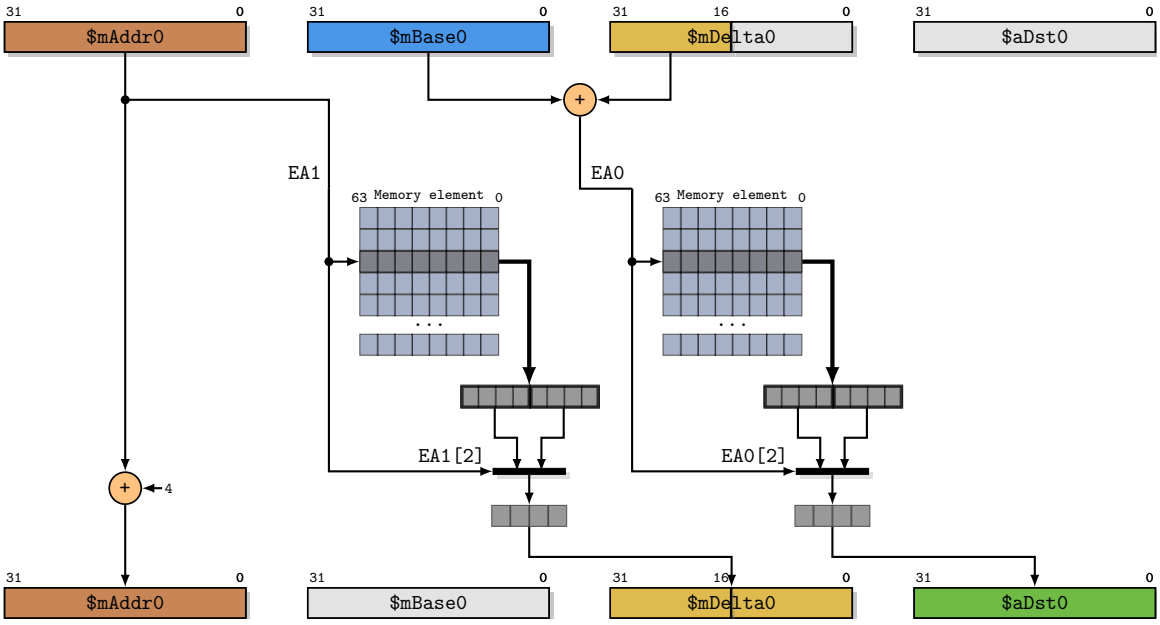


Fig. 3.42: ldd16v2a32

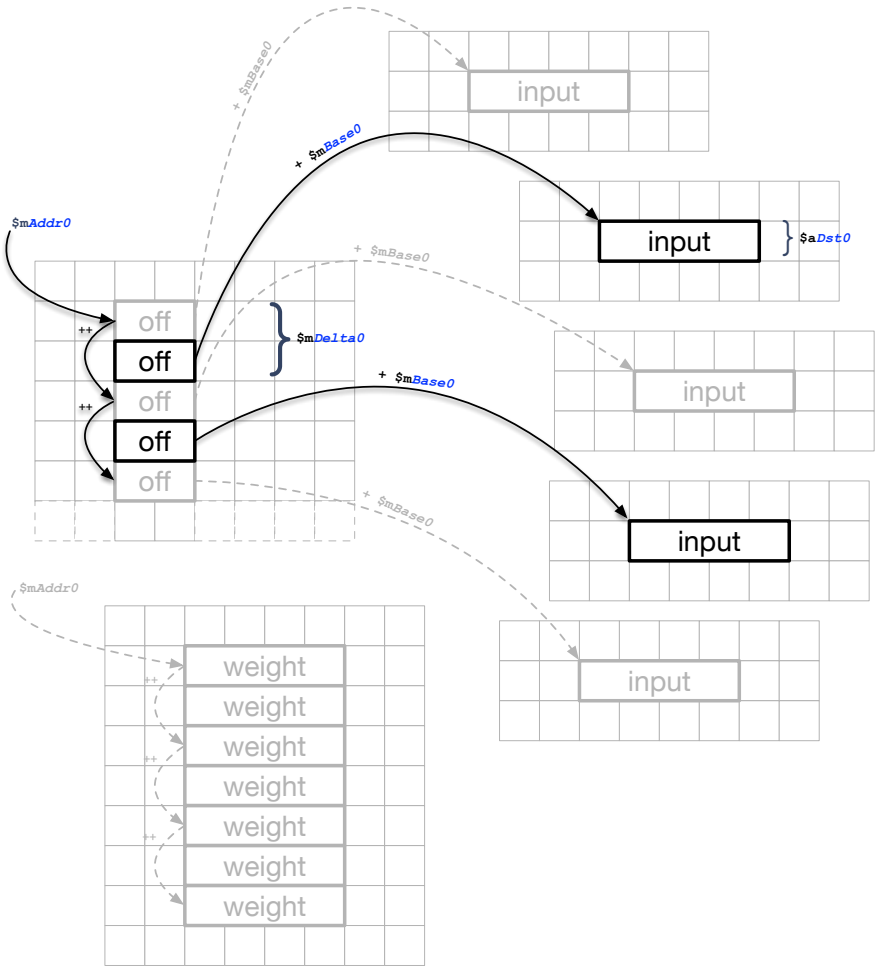


Fig. 3.43: ldd16v2a32

ldd16v2a32 occurs in the following code examples:

- *f32mac example*

3.7.5.2.5 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 3.229: *ldd16a64* instruction definition

ldd16a64			worker	main
Syntax	`ldd16a64 $a`*Dst0*`:`*Dst0+1*`, $m`*Addr0*`++, $m`*Base0*`, $m`*Delta0*`@`			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mAddr0;` `array<HalfDataWord,2> op2 = {` `  (($mDelta0[0] >> 0) & 0xffff),` `  (($mDelta0[0] >> 16) & 0xffff)` `};` `uint16_t delta = op2[0];` `EA[0] = op0 + delta;` `EA[1] = op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x1) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextDeltaAddress = op1 + 2;`		
	Memory	`uint64_t data = loadDoubleWord(EA[0]);` `uint16_t newDelta = loadHalf(EA[1]);`		
	Commit	`$mAddr0 = nextDeltaAddress;` `$mDelta0 = {` `  ((newDelta & 0xffff) << 0) |` `  0` `};` `$aDst0:Dst0+1 = {` `  data & 0xffffffffULL,` `  (data >> 32ULL) & 0xffffffffULL` `};`		

Function references: *TMem_IsValidAddress*

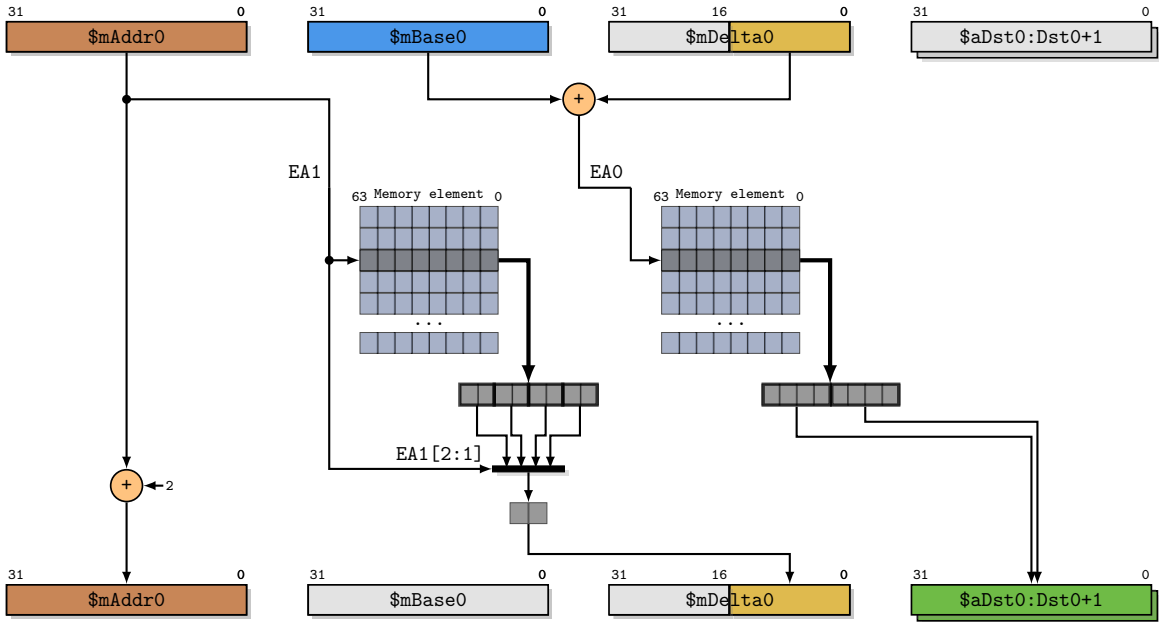


Fig. 3.44: ldd16a64

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 3.230: *ld64a32* instruction definition

ld64a32			worker	main
Syntax	`ld64a32 $a`*Dst0+1*`:`*Dst0+3*`, $m`*Addr0*`++, $m`*Base0*`, $m`*Delta0*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mAddr0;` `HalfDataWord op2 = (($mDelta0 >> 0) & 0xffff);` `DataWord delta = op2;` `EA[0] = op0 + delta;` `EA[1] = op1;`		
	Except In	`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);` `}`		
	Compute	`// Increment ptr address` `DataWord nextAddr = op1 + 8;`		
	Memory	`uint32_t sparseData = loadWord(EA[0]);` `uint64_t weightsPair = loadDoubleWord(EA[1]);`		
	Commit	`$mAddr0 = nextAddr;` `$aDst0+1:Dst0+3 = {` `  sparseData,` `  weightsPair & 0xffffffffULL,` `  (weightsPair >> 32ULL) & 0xffffffffULL` `};`		

Function references: *TMem_IsValidAddress*

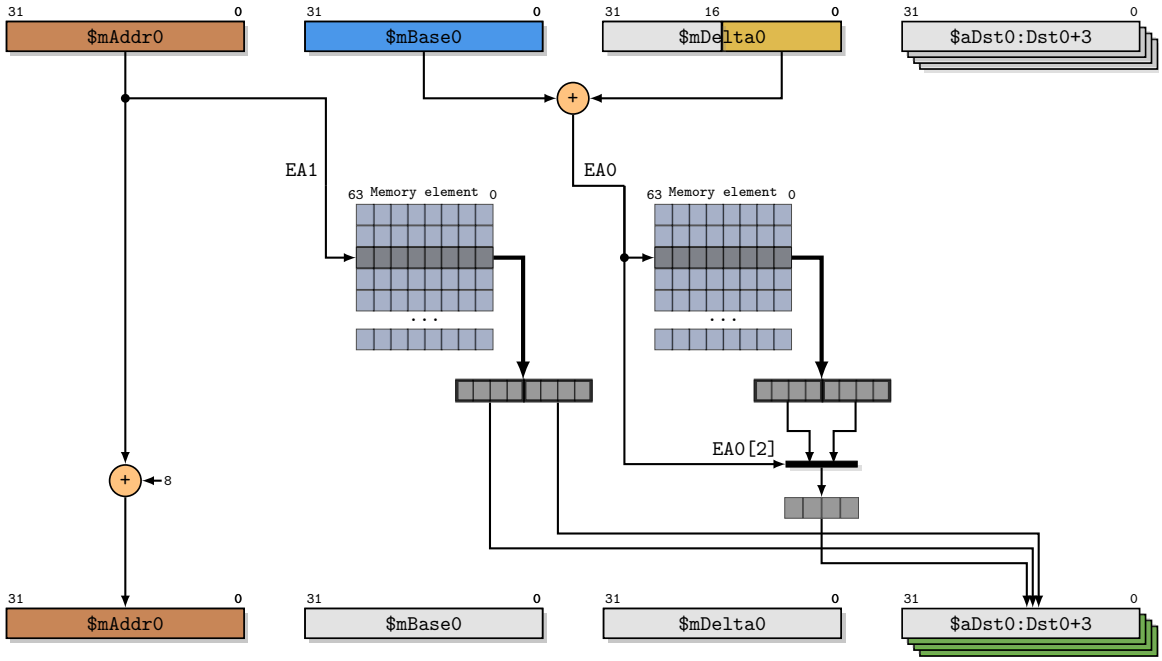


Fig. 3.45: ld64a32

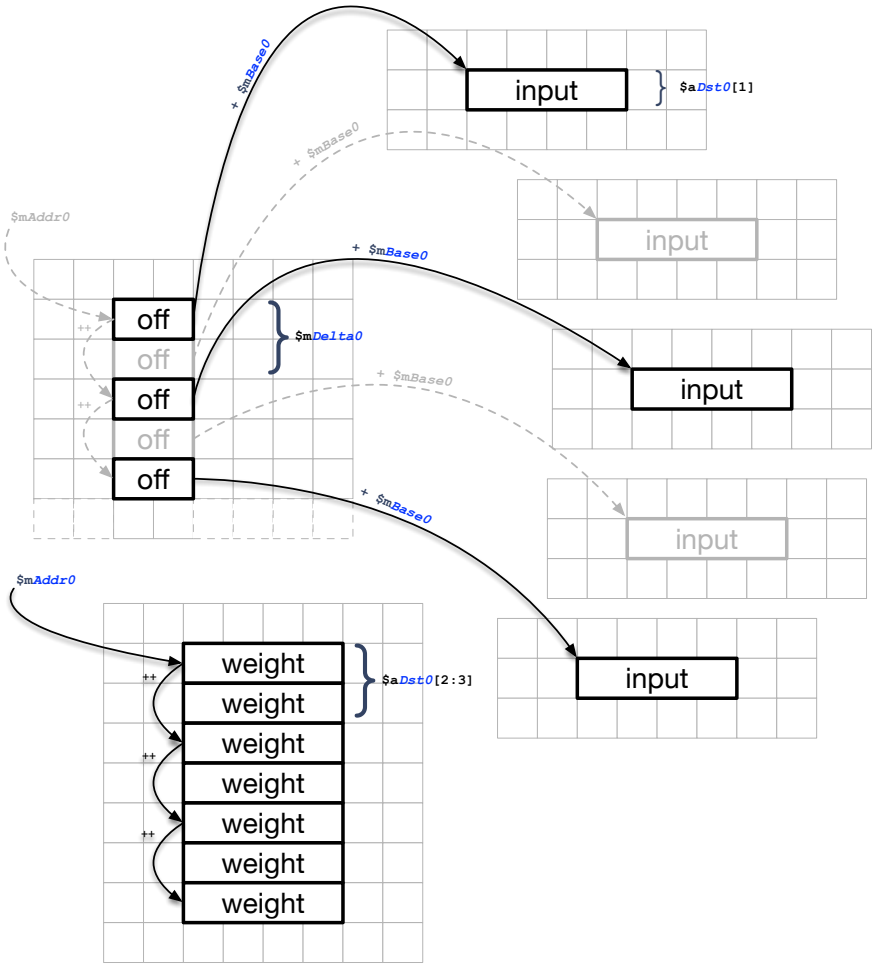


Fig. 3.46: ld64a32

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 3.231: *ld64b16pace* instruction definition

ld64b16pace			worker	main
Syntax	`ld64b16pace $aDst0:Dst0+1, $aDst1, $mAddr0:Addr0+1+=, $mStride0, Strimm2x2`			
Semantics	**Prepare**	`DataWord op0 = $mStride0;` `array<DataWord,2> op1 = {` `  $mAddr0:Addr0+1[0],` `  $mAddr0:Addr0+1[1]` `};` `DataWord op4 = Strimm2x2;` `array<DataWord,3> addrs;` `EA[0] = Tile_ExtractPackedAddress(op1, 0);` `EA[1] = Tile_ExtractPackedAddress(op1, 1);`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x1) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`// Post-increment both addresses as specified by the immediate` `int32_t stride0 = Tile_ExtractPackedStride(op0, op4);` `int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2));` `addrs[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[1] = (EA[1] + (stride1 * 2)) & TMEM_FULL_ADDRESS_MASK;` `addrs[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged`		
	Memory	`uint64_t data0 = loadDoubleWord(EA[0]);` `uint16_t data1 = loadHalf(EA[1]);`		
	Commit	`$mAddr0:Addr0+1 = {` `  Tile_TripleAddressPack_Lower(addrs),` `  Tile_TripleAddressPack_Upper(addrs)` `};` `$aDst0:Dst0+1 = {` `  data0 & 0xffffffffULL,` `  (data0 >> 32ULL) & 0xffffffffULL` `};` `$aDst1 = {` `  ((data1 & 0xffff) << 0) |` `  ((data1 & 0xffff) << 16)` `};`		

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

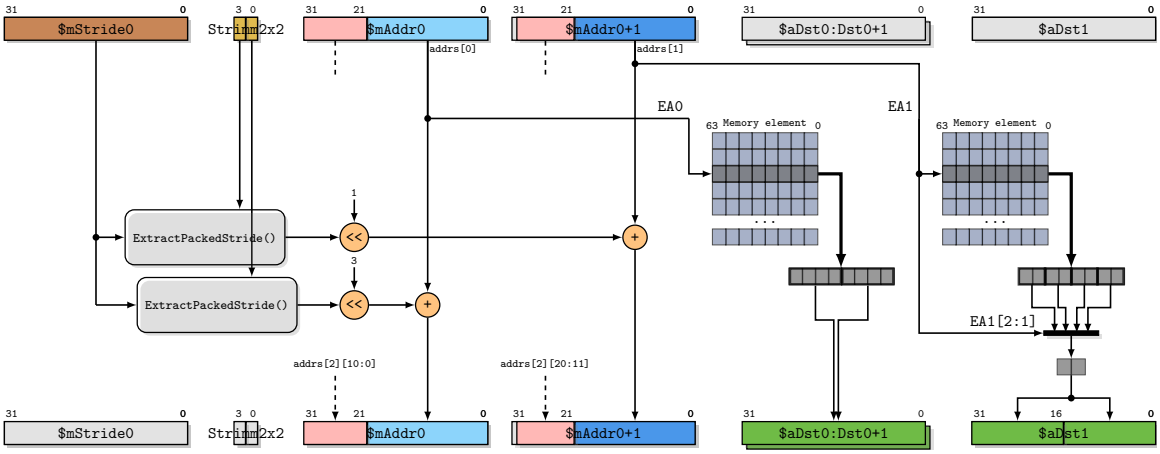


Fig. 3.47: ld64b16pace

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 3.232: *ld64a32pace* instruction definition

ld64a32pace			worker	main
Syntax	`ld64a32pace $aDst0:Dst0+1, $aDst1, $mAddr0:Addr0+1+=, $mStride0, Strimm2x2`			
Semantics	**Prepare**	`DataWord op0 = $mStride0;` `array<DataWord,2> op1 = {` `  $mAddr0:Addr0+1[0],` `  $mAddr0:Addr0+1[1]` `};` `DataWord op4 = Strimm2x2;` `array<DataWord,3> addrs;` `EA[0] = Tile_ExtractPackedAddress(op1, 0);` `EA[1] = Tile_ExtractPackedAddress(op1, 1);`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x3) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`// Post-increment both addresses as specified by the immediate` `int32_t stride0 = Tile_ExtractPackedStride(op0, op4);` `int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2));` `addrs[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[1] = (EA[1] + (stride1 * 4)) & TMEM_FULL_ADDRESS_MASK;` `addrs[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged`		
	Memory	`uint64_t data0 = loadDoubleWord(EA[0]);` `uint32_t data1 = loadWord(EA[1]);`		
	Commit	`$mAddr0:Addr0+1 = {` `  Tile_TripleAddressPack_Lower(addrs),` `  Tile_TripleAddressPack_Upper(addrs)` `};` `$aDst0:Dst0+1 = {` `  data0 & 0xffffffffULL,` `  (data0 >> 32ULL) & 0xffffffffULL` `};` `$aDst1 = data1;`		

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

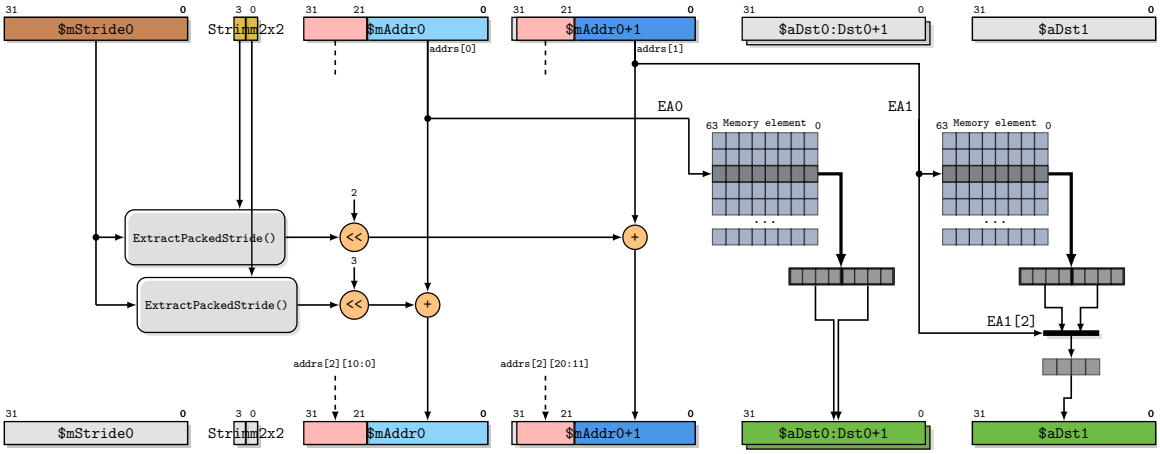


Fig. 3.48: ld64a32pace

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 3.233: *ld2x64pace* instruction definition

ld2x64pace			worker	main
Syntax	`ld2x64pace $aDst0:Dst0+1, $aDst1:Dst1+1, $mAddr0:Addr0+1+=, $mStride0, Strimm2x2`			
Semantics	**Prepare**	`DataWord op0 = $mStride0;` `array<DataWord,2> op1 = {` `  $mAddr0:Addr0+1[0],` `  $mAddr0:Addr0+1[1]` `};` `DataWord op4 = Strimm2x2;` `array<DataWord,3> addrs;` `EA[0] = Tile_ExtractPackedAddress(op1, 0);` `EA[1] = Tile_ExtractPackedAddress(op1, 1);`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (!TMem_IsValidAddress(EA[1])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[1] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `} else if (hadMemoryConflict(EA[1])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`// Post-increment both addresses as specified by the immediate` `int32_t stride0 = Tile_ExtractPackedStride(op0, op4);` `int32_t stride1 = Tile_ExtractPackedStride(op0, (op4 >> 2));` `addrs[0] = (EA[0] + (stride0 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[1] = (EA[1] + (stride1 * 8)) & TMEM_FULL_ADDRESS_MASK;` `addrs[2] = (Tile_ExtractPackedAddress(op1, 2)); // Unchanged`		
	Memory	`uint64_t data0 = loadDoubleWord(EA[0]);` `uint64_t data1 = loadDoubleWord(EA[1]);`		
	Commit	`$mAddr0:Addr0+1 = {` `  Tile_TripleAddressPack_Lower(addrs),` `  Tile_TripleAddressPack_Upper(addrs)` `};` `$aDst0:Dst0+1 = {` `  data0 & 0xffffffffULL,` `  (data0 >> 32ULL) & 0xffffffffULL` `};` `$aDst1:Dst1+1 = {` `  data1 & 0xffffffffULL,` `  (data1 >> 32ULL) & 0xffffffffULL` `};`		

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

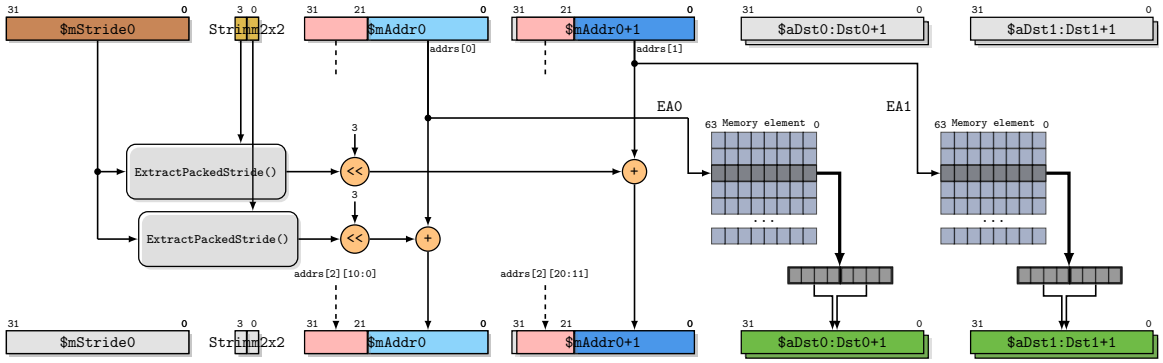


Fig. 3.49: ld2x64pace

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

Table 3.234: Loads to MRF addressing modes

Index	Description	Instructions
0	instr dst0, src0, src1, imm0	*ld32*, *lds16*, *lds8*, *ldz16*, *ldz8*
1	instr dst0, src0, src1, src2	*ld32*, *lds16*, *lds8*, *ldz16*, *ldz8*
2	instr dst0, src0, srcDst0+=, imm0	*ld32step*, *lds16step*, *lds8step*, *ldz16step*, *ldz8step*
3	instr dst0, src0, srcDst0+=, src1	*ld32step*, *lds16step*, *lds8step*, *ldz16step*, *ldz8step*

Table 3.235: Loads to MRF

	Addressing mode			
Access bits	**0**	**1**	**2**	**3**
8	✓ (w, s)	✓ (w, s)	✓ (w, s)	✓ (w, s)
16	✓ (w, s)	✓ (w, s)	✓ (w, s)	✓ (w, s)
32	✓ (w, s)	✓ (w, s)	✓ (w, s)	✓ (w, s)

Table 3.236: Loads to ARF addressing modes

Index	Description	Instructions
0	instr dst0, src0, src1, imm0	*ld128*, *ld32*, *ld64*, *ldb16*, *ldb8*
1	instr dst0, src0, src1, src2	*ld128*, *ld32*, *ld64*, *ldb16*, *ldb8*
2	instr dst0, src0, srcDst0+=, imm0	*ld128step*, *ld32step*, *ld64step*, *ldb16step*, *ldb8step*
3	instr dst0, src0, srcDst0+=, src1	*ld128step*, *ld32step*, *ld64step*, *ldb16step*, *ldb8step*

Table 3.237: Loads to ARF

	Addressing mode			
Access bits	**0**	**1**	**2**	**3**
8	✓	✓	✓	✓
16	✓	✓	✓	✓
32	✓	✓	✓	✓
64	✓	✓	✓	✓
128	✓	✓	✓	✓

3.7.5.3.1 atom

Copy an *ARF* register value to *MRF*. No format conversion is performed.

Table 3.238: *atom* instruction definition

atom			worker	main
Syntax	`atom $mDst0, $aSrc0`			
Semantics	**Prepare**	`DataWord op1 = $aSrc0;`		
	Commit	`$mDst0 = op1;`		

3.7.5.3.2 ldb8

Load and broadcast a single 8-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 (replicating) the 8-bit data value.

Table 3.239: *ldb8* instruction definition

ldb8			worker	main
Syntax	`ldb8 $aDst0, $mBase0, $mDelta0, $mOff0` `ldb8 $aDst0, $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	`$aDst0 = {` `  ((data & 0xff) << 0) |` `  ((data & 0xff) << 8) |` `  ((data & 0xff) << 16) |` `  ((data & 0xff) << 24)` `};`		

Function references: *TMem_IsValidAddress*

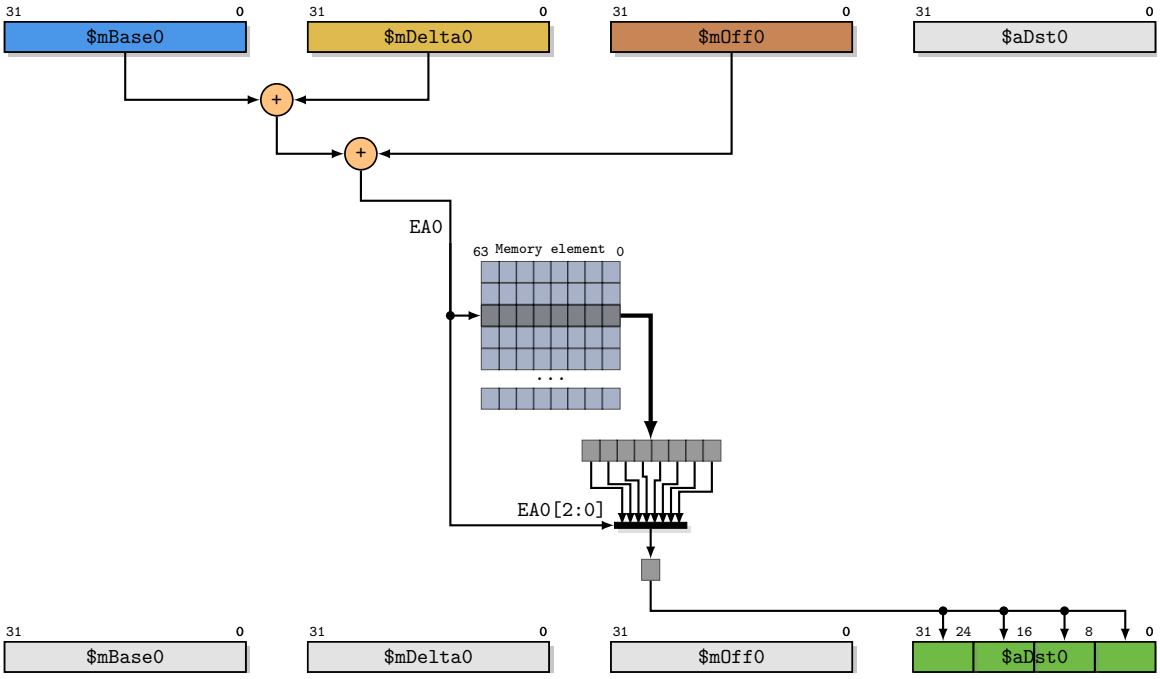


Fig. 3.50: ldb8 example

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 3.240: *lds8* instruction definition

lds8			both	main
Syntax	`lds8 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `lds8 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, `*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_SignExtend(data, 8);`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

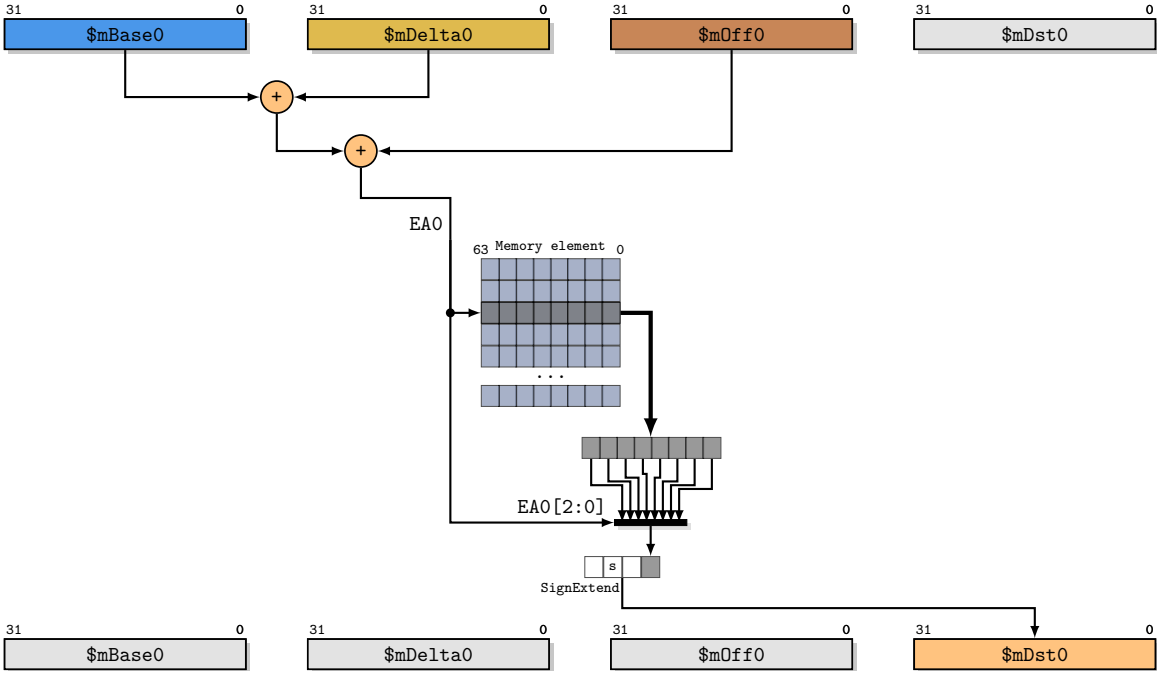


Fig. 3.51: lds8 example

3.7.5.3.4 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 3.241: *ldz8* instruction definition

ldz8			both	main
Syntax	ldz8 $m*Dst0*, $m*Base0*, $m*Delta0*, $m*Off0* ldz8 $m*Dst0*, $m*Base0*, $m*Delta0*, *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*

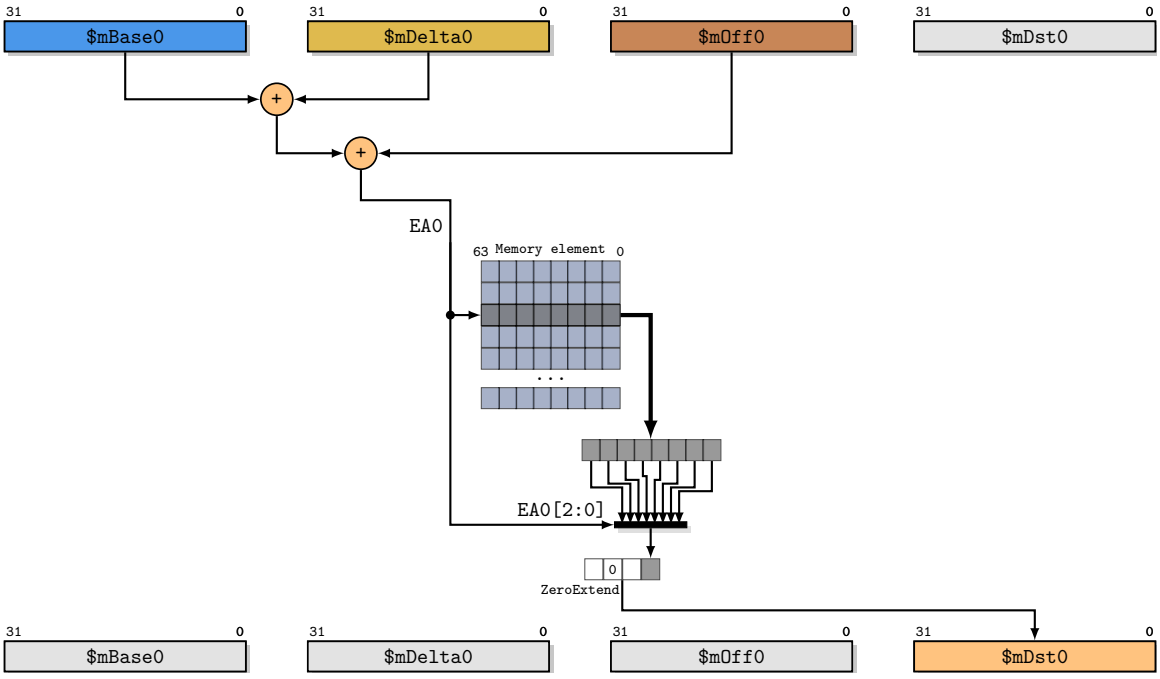


Fig. 3.52: ldz8 example

3.7.5.3.5 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 3.242: *ldb16* instruction definition

ldb16			worker	main
Syntax	ldb16 $a*Dst0*, $m*Base0*, $m*Delta0*, $m*Off0* ldb16 $a*Dst0*, $m*Base0*, $m*Delta0*, *zimm12*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOff0 << 1), (zimm12 << 1))>;` `EA[0] = op0 + op1 + op2;`		
	Except In	`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	`uint16_t data = loadHalf(EA[0]);`		
	Commit	`$aDst0 = {` `  ((data & 0xffff) << 0) |` `  ((data & 0xffff) << 16)` `};`		

Function references: *TMem_IsValidAddress*

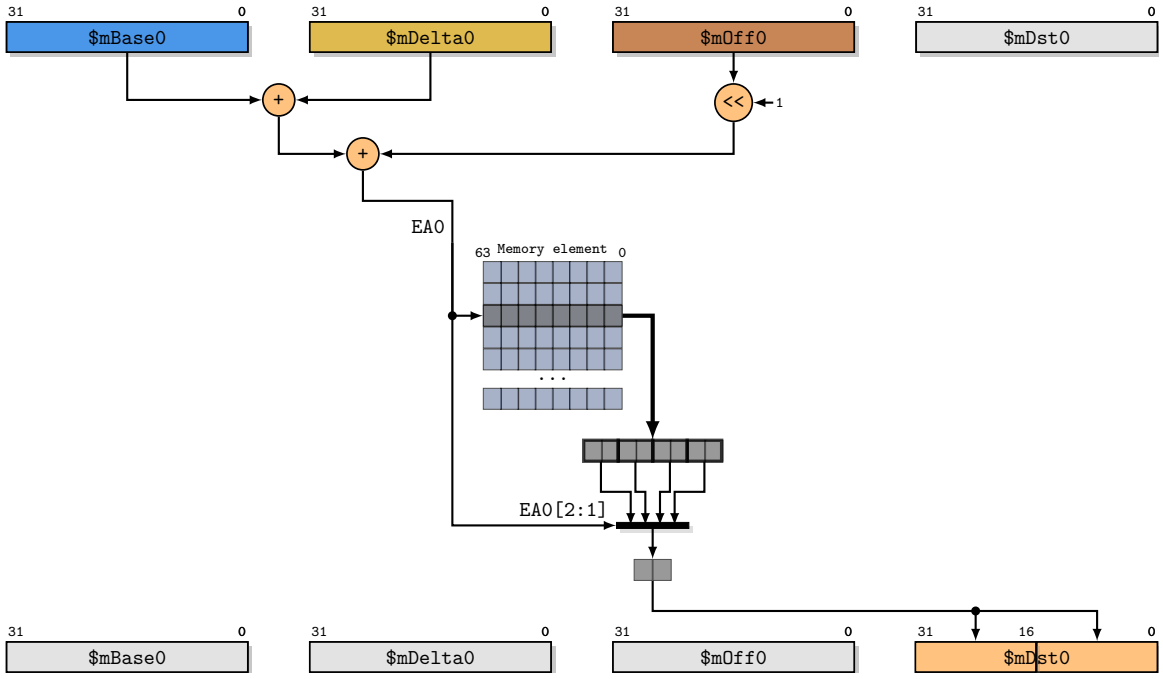


Fig. 3.53: ldb16 example

3.7.5.3.6 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 3.243: *lds16* instruction definition

lds16			both	main
Syntax	`lds16 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `lds16 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOff0 << 1), (zimm12 << 1))>;` `EA[0] = op0 + op1 + op2;`		
	Except In	`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	`uint16_t data = loadHalf(EA[0]);`		
	Commit	`$mDst0 = Tile_SignExtend(data, 16);`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

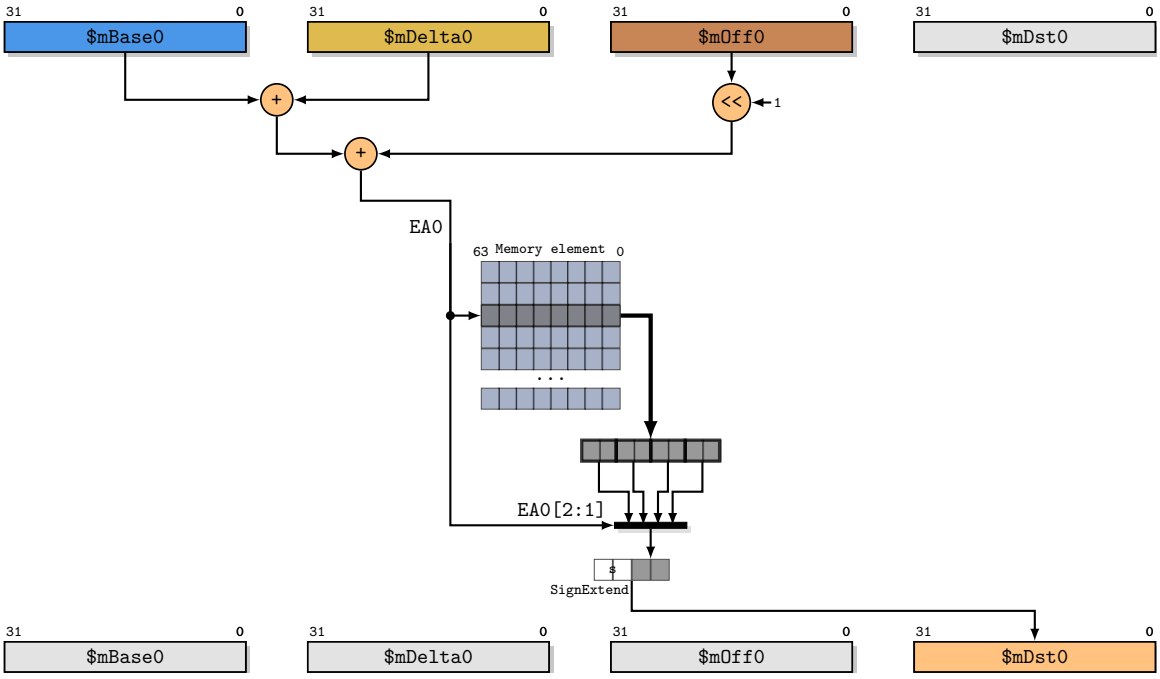


Fig. 3.54: lds16 example

3.7.5.3.7 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 3.244: *ldz16* instruction definition

ldz16			both	main
Syntax	`ldz16 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `ldz16 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOff0 << 1), (zimm12 << 1))>;` `EA[0] = op0 + op1 + op2;`		
	Except In	`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	`uint16_t data = loadHalf(EA[0]);`		
	Commit	`$mDst0 = Tile_ZeroExtend(data, 16);`		

Function references: *Tile_ZeroExtend* , *TMem_IsValidAddress*

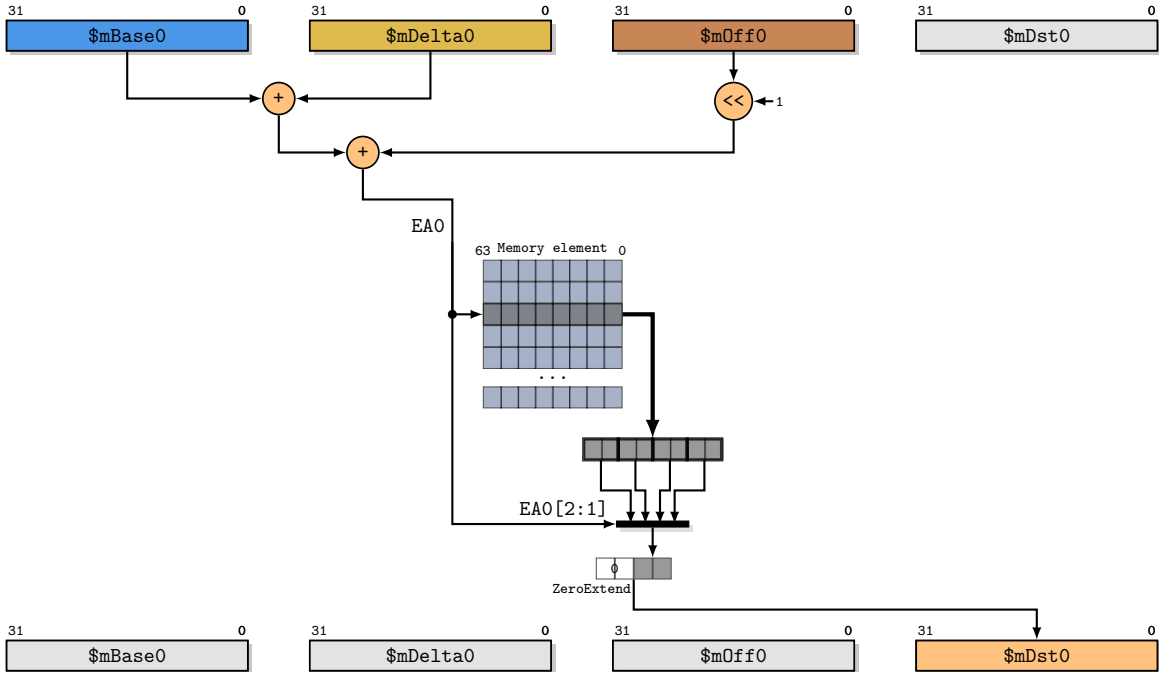


Fig. 3.55: ldz16 example

3.7.5.3.8 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 3.245: *ld32* instruction definition

ld32			both	main
Syntax	`ld32 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `ld32 $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, `*zimm12*			
ld32			**worker**	**main**
Syntax	`ld32 $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `ld32 $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOff0 << 2), (zimm12 << 2))>;` `EA[0] = op0 + op1 + op2;`		
	Except In	`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);` `}`		
	Memory	`uint32_t data = loadWord(EA[0]);`		
	Commit	`<($mDst0, $aDst0)> = data;`		

Function references: *TMem_IsValidAddress*

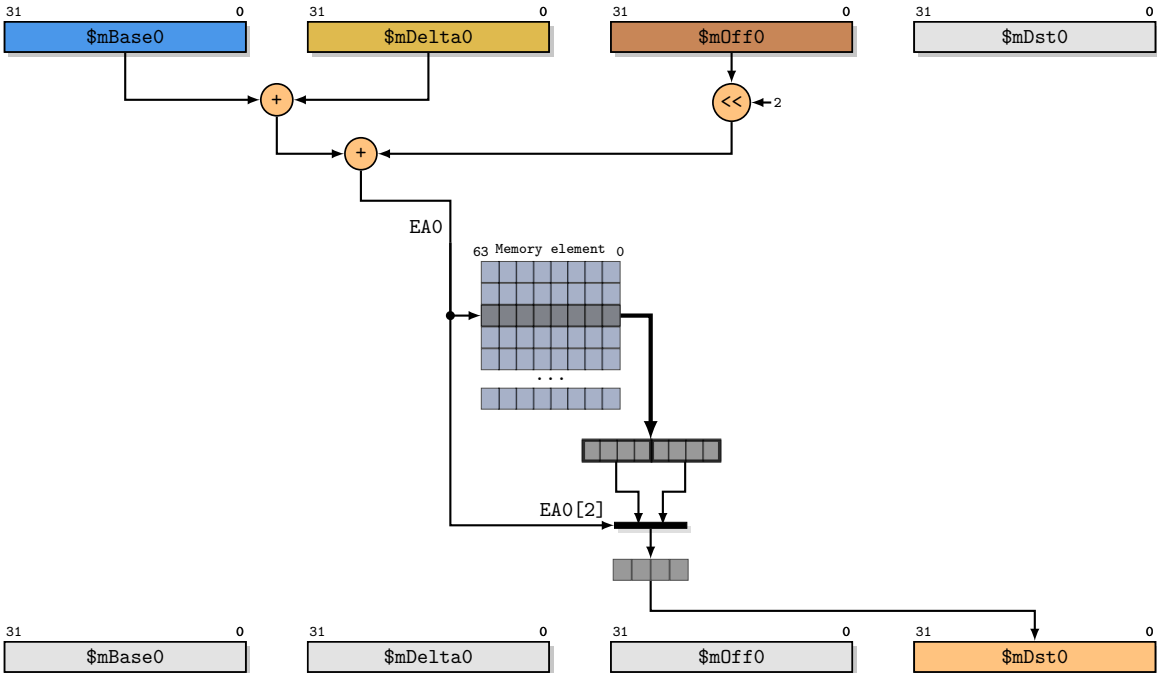


Fig. 3.56: ld32 example

3.7.5.3.9 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 3.246: *ld64* instruction definition

ld64			worker	main
Syntax	ld64 $a*Dst0*:*Dst0+1*, $m*Base0*, $m*Delta0*, $m*Off0* ld64 $a*Dst0*:*Dst0+1*, $m*Base0*, $m*Delta0*, *zimm12*			
Semantics	**Prepare**	DataWord op0 = $mBase0; DataWord op1 = $mDelta0; DataWord op2 = <(($mOff0 << 3), (zimm12 << 3))>; 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 = loadDoubleWord(EA[0]);		
	Commit	$aDst0:Dst0+1 = { data & 0xffffffffULL, (data >>32ULL) & 0xffffffffULL };		

Function references: *TMem_IsValidAddress*

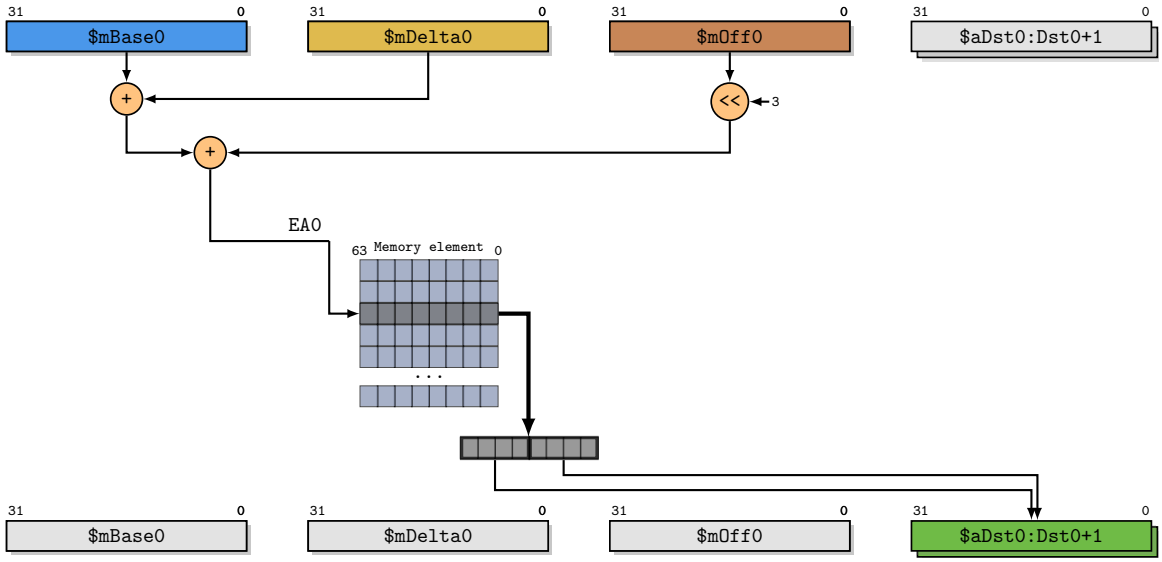


Fig. 3.57: ld64 example

3.7.5.3.10 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 3.247: *ld128* instruction definition

ld128			worker	main
Syntax	`ld128 $a`*Dst0*`:`*Dst0+3*`, $m`*Base0*`, $m`*Delta0*`, $m`*Off0* `ld128 $a`*Dst0*`:`*Dst0+3*`, $m`*Base0*`, $m`*Delta0*`, `*zimm12*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOff0 << 4), (zimm12 << 4))>;` `array<uint64_t,2> data;` `EA[0] = op0 + op1 + op2;` `EA[1] = (op0 + op1 + op2) | 0x8;`		
	Except In	`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);` `}`		
	Memory	`data[0] = loadDoubleWord(EA[0]);` `data[1] = loadDoubleWord(EA[1]);`		
	Commit	`$aDst0:Dst0+3 = {` `  data[0] & 0xffffffffULL,` `  (data[0] >>32ULL) & 0xffffffffULL,` `  data[1] & 0xffffffffULL,` `  (data[1] >>32ULL) & 0xffffffffULL` `};`		

Function references: *TMem_IsValidAddress* , *TMem_AddressInterleaveFactor*

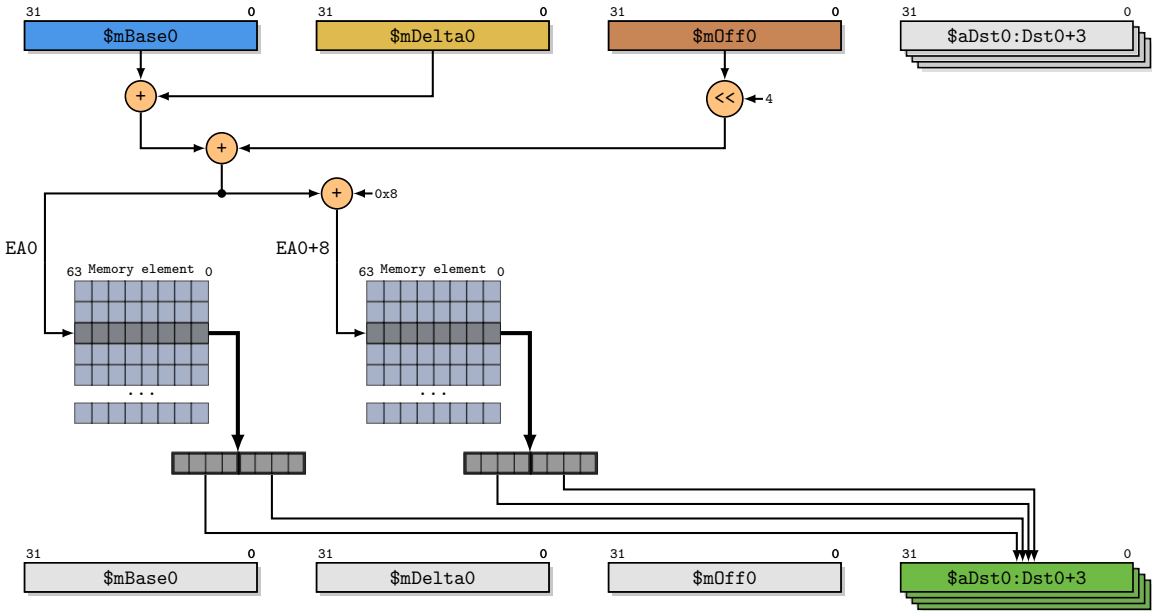


Fig. 3.58: ld128 example

3.7.5.3.11 ldb8step

8-bit load and broadcast with 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 (replicating) 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 3.248: *ldb8step* instruction definition

ldb8step		worker	main
Syntax	ldb8step $a*Dst0*, $m*Base0*, $m*Delta0*+=, $m*Stride0* ldb8step $a*Dst0*, $m*Base0*, $m*Delta0*+=, *simm8*		
Semantics	**Prepare** **Except In** **Compute** **Memory** **Commit**		

```
DataWord op0 = $mBase0;
DataWord op1 = $mDelta0;
DataWord op2 = <(Tile_SignExtend(simm8, 8), $mStride0)>;

EA[0] = op0 + op1;
```

```
if (!TMem_IsValidAddress(EA[0])) {
  EXCEPT(TEXCPT_INVALID_ADDR);

} else if (hadMemoryConflict(EA[0])) {
  EXCEPT(TEXCPT_CONFLICT);

}
```

```
DataWord nextAddr = op1 + op2;
```

```
uint8_t data = loadByte(EA[0]);
```

```
$mDelta0 = nextAddr;
$aDst0 = {
  ((data & 0xff) << 0) |
  ((data & 0xff) << 8) |
  ((data & 0xff) << 16) |
  ((data & 0xff) << 24)
};
```

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

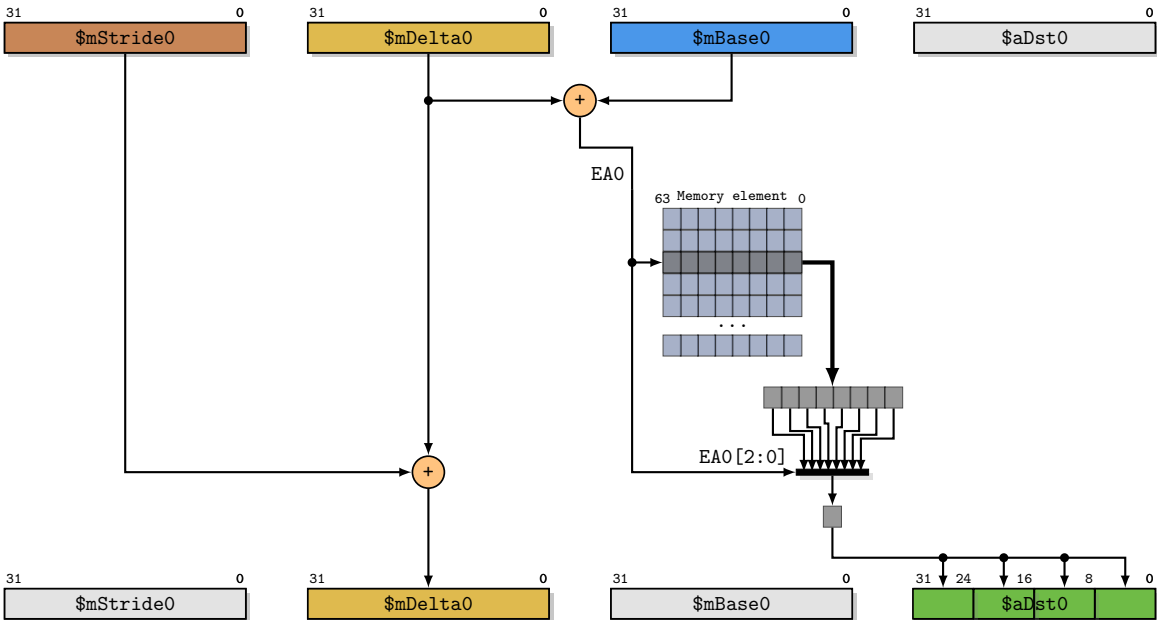


Fig. 3.59: ldb8step example

3.7.5.3.12 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 3.249: *lds8step* instruction definition

lds8step			both	main
Syntax	`lds8step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `lds8step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(Tile_SignExtend(simm8, 8), $mStride0)>;` `EA[0] = op0 + op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint8_t data = loadByte(EA[0]);`		
	Commit	`$mDelta0 = nextAddr;` `$mDst0 = Tile_SignExtend(data, 8);`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

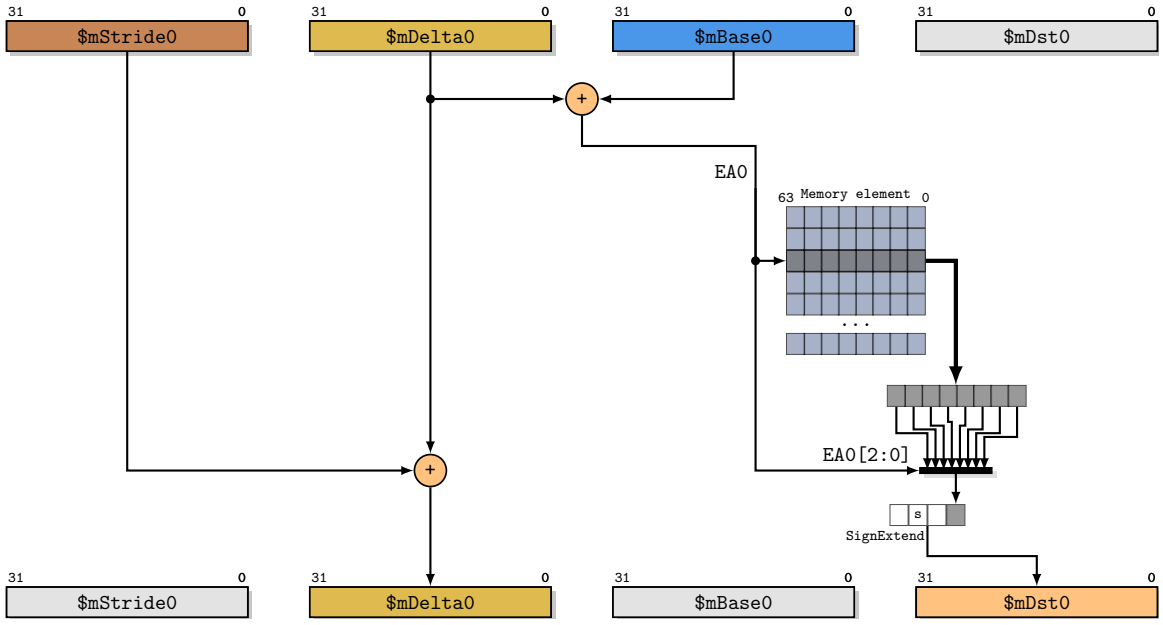


Fig. 3.60: lds8step example

3.7.5.3.13 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 3.250: *ldz8step* instruction definition

ldz8step			both	main
Syntax	`ldz8step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ldz8step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(Tile_SignExtend(simm8, 8), $mStride0)>;` `EA[0] = op0 + op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint8_t data = loadByte(EA[0]);`		
	Commit	`$mDelta0 = nextAddr;` `$mDst0 = Tile_ZeroExtend(data, 8);`		

Function references: *Tile_SignExtend* , *Tile_ZeroExtend* , *TMem_IsValidAddress*

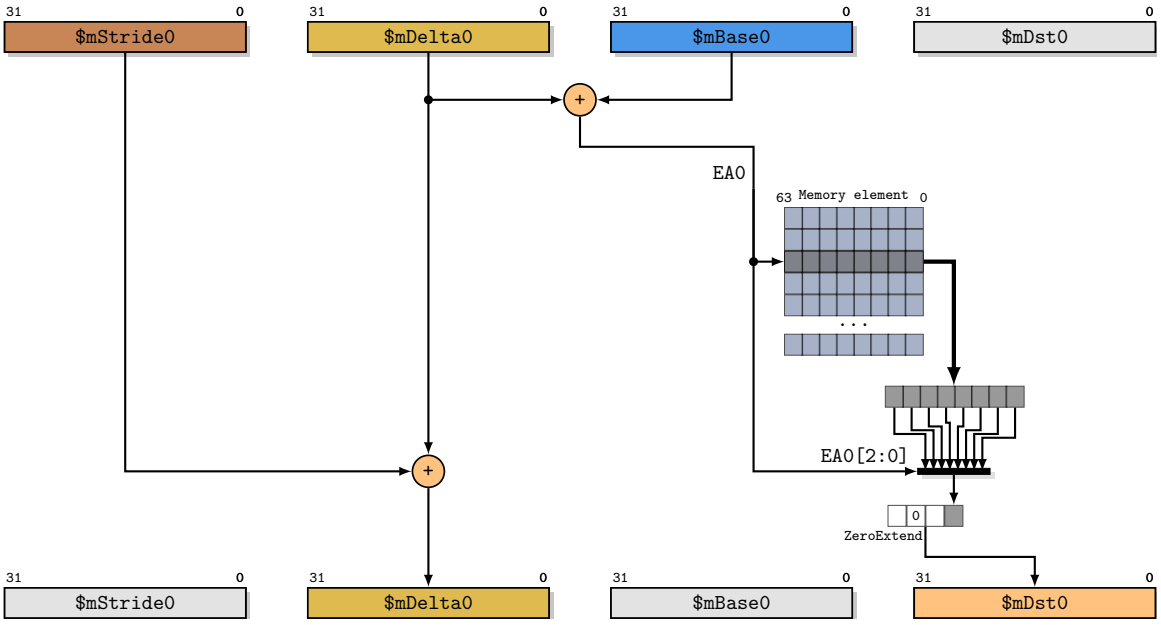


Fig. 3.61: ldz8step example

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 3.251: *ldb16step* instruction definition

ldb16step			worker	main
Syntax	`ldb16step $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ldb16step $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 1), ($mStride0 << 1))>;` `EA[0] = op0 + op1;`		
	Except In	`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)` `};`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

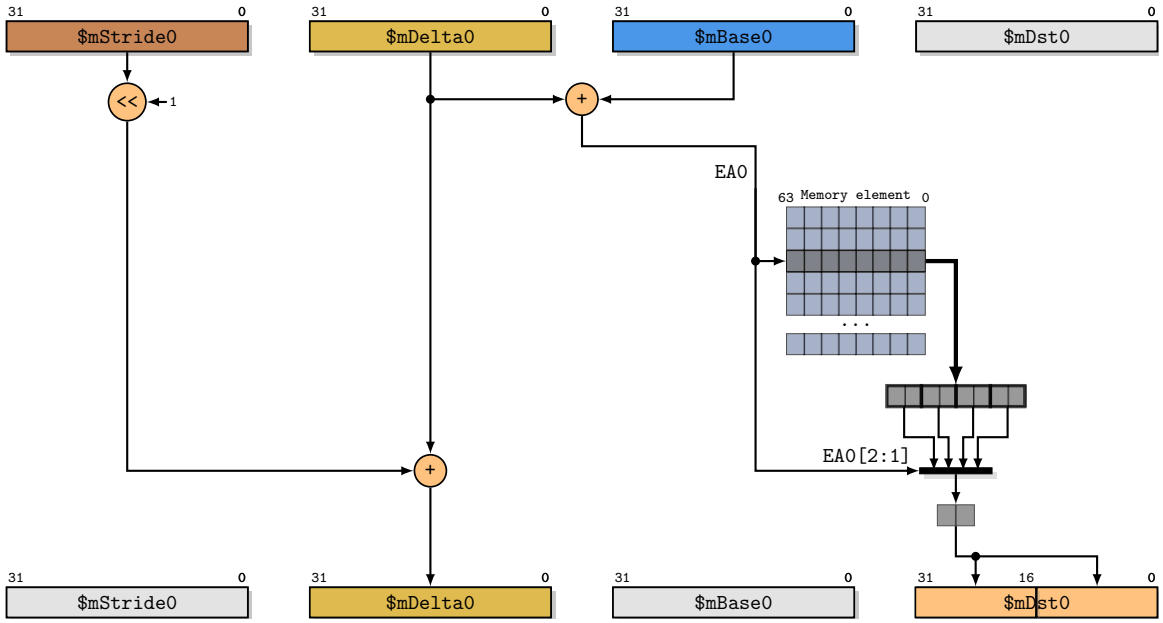


Fig. 3.62: ldb16step example

3.7.5.3.15 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 3.252: *lds16step* instruction definition

lds16step			both	main
Syntax	`lds16step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `lds16step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 1), ($mStride0 << 1))>;` `EA[0] = op0 + op1;`		
	Except In	`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	`$mDelta0 = nextAddr;` `$mDst0 = Tile_SignExtend(data, 16);`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

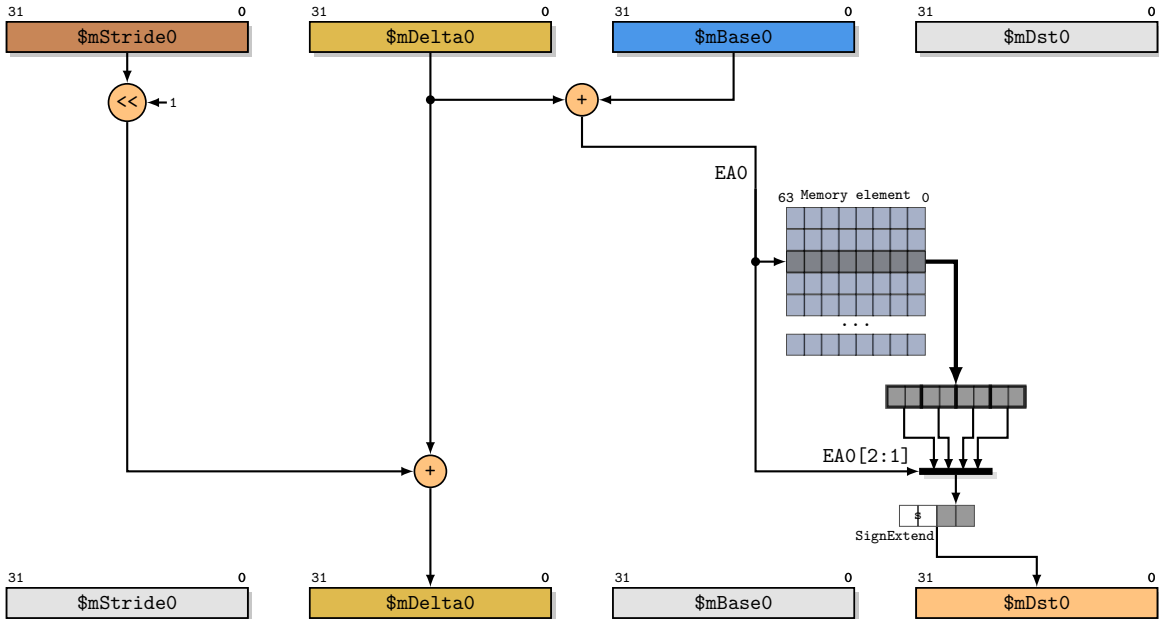


Fig. 3.63: lds16step example

3.7.5.3.16 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 3.253: *ldz16step* instruction definition

ldz16step			both	main
Syntax	`ldz16step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ldz16step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 1), ($mStride0 << 1))>;` `EA[0] = op0 + op1;`		
	Except In	`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	`$mDelta0 = nextAddr;` `$mDst0 = Tile_ZeroExtend(data, 16);`		

Function references: *Tile_SignExtend* , *Tile_ZeroExtend* , *TMem_IsValidAddress*

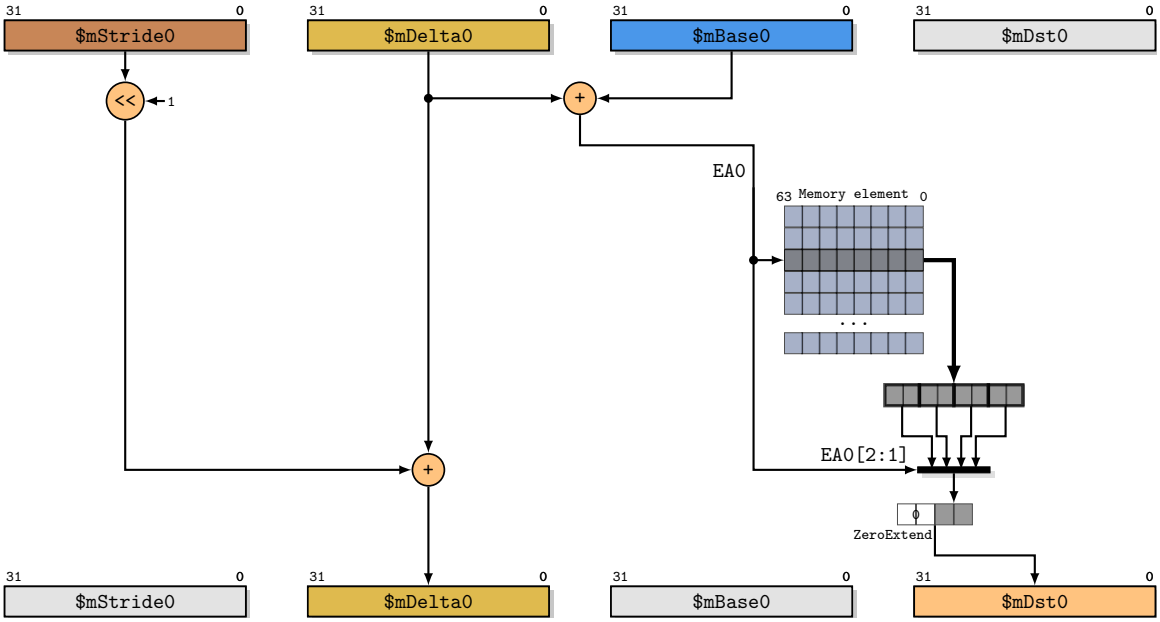


Fig. 3.64: ldz16step example

3.7.5.3.17 ld32step

Naturally aligned single *word* 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 3.254: *ld32step* instruction definition

ld32step			both	main
Syntax	`ld32step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ld32step $m`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
ld32step			**worker**	**main**
Syntax	`ld32step $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ld32step $a`*Dst0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 2), ($mStride0 << 2))>;` `EA[0] = op0 + op1;`		
	Except In	`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);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint32_t data = loadWord(EA[0]);`		
	Commit	`$mDelta0 = nextAddr;` `<($mDst0, $aDst0)> = data;`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

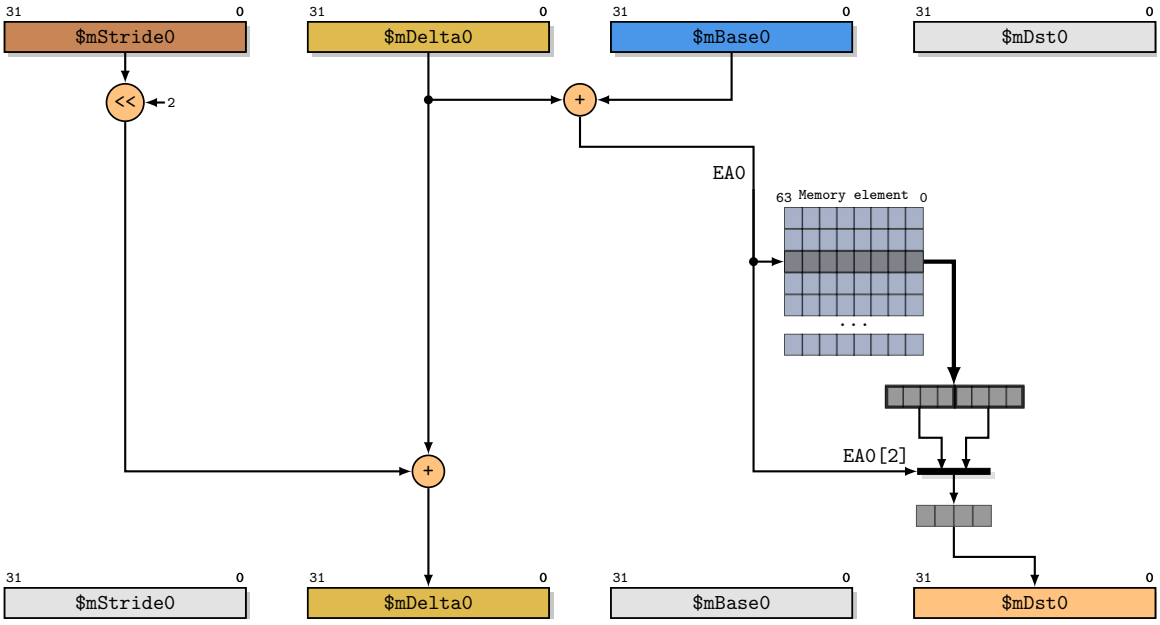


Fig. 3.65: ld32step example

ld32step occurs in the following code examples:

- *ldb16b16 example*

3.7.5.3.18 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 3.255: *ld64step* instruction definition

ld64step			worker	main
Syntax	`ld64step $a`*Dst0*`:`*Dst0*`+1, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ld64step $a`*Dst0*`:`*Dst0*`+1, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 3), ($mStride0 << 3))>;` `EA[0] = op0 + op1;`		
	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);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint64_t data = loadDoubleWord(EA[0]);`		
	Commit	`$mDelta0 = nextAddr;` `$aDst0:Dst0+1 = {` `  data & 0xffffffffULL,` `  (data >> 32ULL) & 0xffffffffULL` `};`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

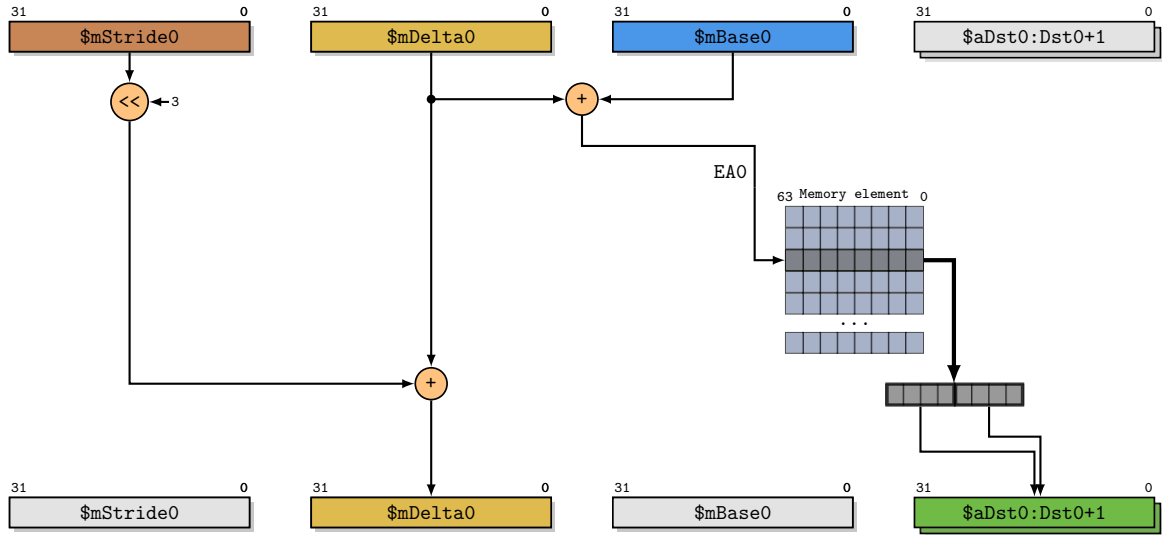


Fig. 3.66: ld64step example

ld64step occurs in the following code examples:

- *f16v4stacc example*

3.7.5.3.19 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 3.256: *ld128step* instruction definition

ld128step			worker	main
Syntax	`ld128step $a`*Dst0*`:`*Dst0+3*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `ld128step $a`*Dst0*`:`*Dst0+3*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 4), ($mStride0 << 4))>;` `array<uint64_t,2> data;` `EA[0] = op0 + op1;` `EA[1] = (op0 + op1) | 0x8;`		
	Except In	`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);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`data[0] = loadDoubleWord(EA[0]);` `data[1] = loadDoubleWord(EA[1]);`		
	Commit	`$mDelta0 = nextAddr;` `$aDst0:Dst0+3 = {` `  data[0] & 0xffffffffULL,` `  (data[0] >> 32ULL) & 0xffffffffULL,` `  data[1] & 0xffffffffULL,` `  (data[1] >> 32ULL) & 0xffffffffULL` `};`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress* , *TMem_AddressInterleaveFactor*

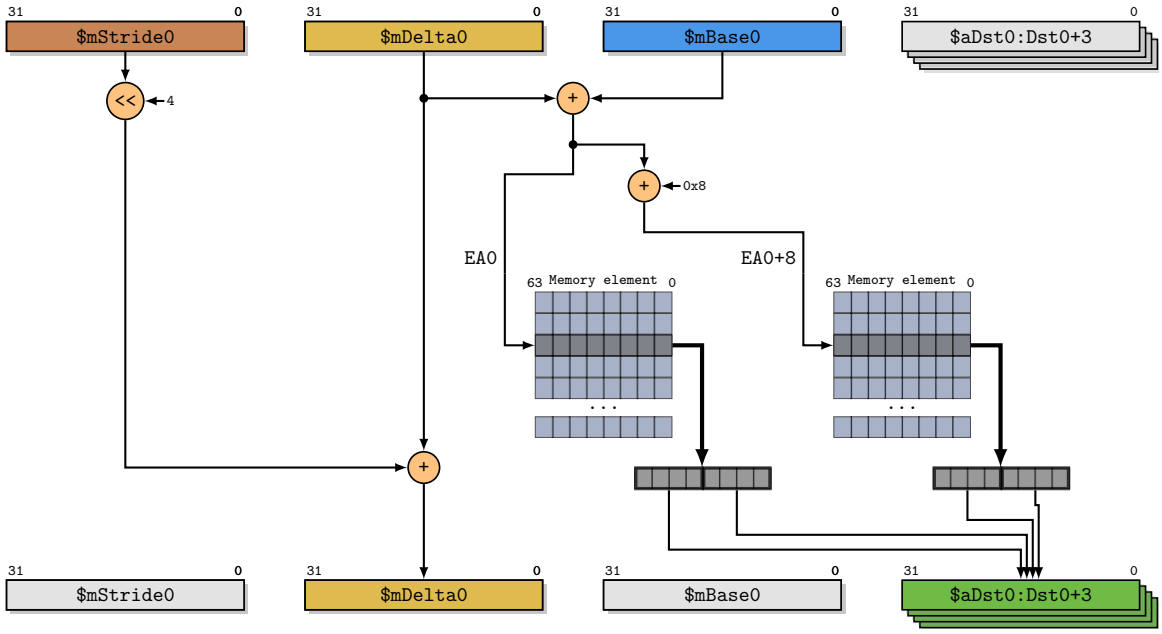


Fig. 3.67: ld128step example

3.7.5.3.20 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 3.257: *ld64putcs* instruction definition

ld64putcs			supervisor	main
Syntax	`ld64putcs` *zimm8*			
Semantics	**Prepare**	`DataWord op0 = zimm8;` `EA[0] = $CCCSLOAD;`		
	Except In	`// Check that the common configuration space address is valid` `if (!TReg_IsValidCCCS(op0)) {` `  EXCEPT(TEXCPT_INVALID_OP);` `} else if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Memory	`uint64_t data = loadDoubleWord(EA[0]);`		
	Commit	`setCCCSState(op0, data);` `$CCCSLOAD = $CCCSLOAD + 8;`		

Architectural state references: *$CCCSLOAD*

Function references: *TMem_IsValidAddress* , *TReg_IsValidCCCS*

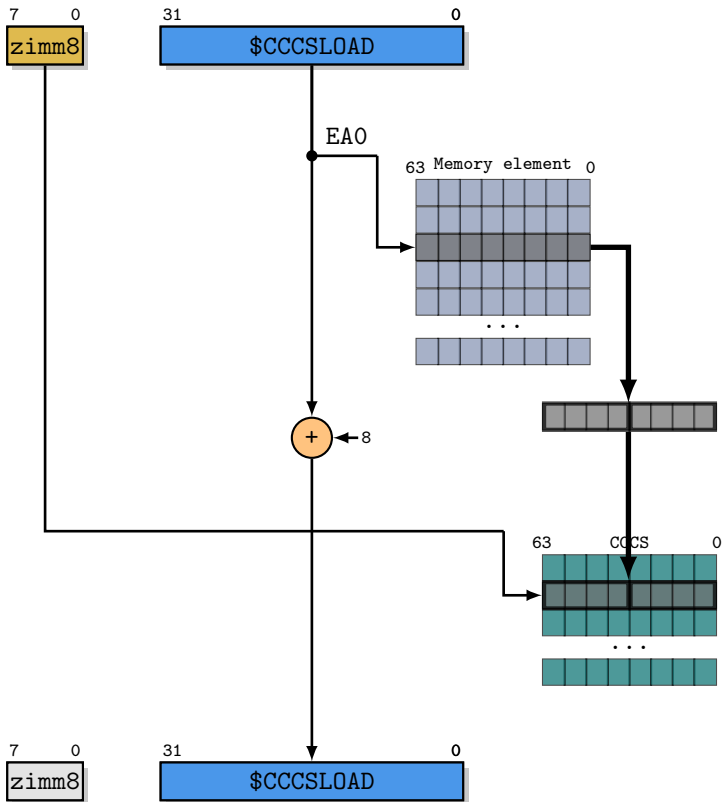


Fig. 3.68: ld64putcs

3.7.5.3.21 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 3.258: *ld128putcs* instruction definition

ld128putcs			supervisor	main
Syntax	`ld128putcs` *zimm8*			
Semantics	**Prepare**	`DataWord op0 = zimm8;` `array<uint64_t,2> data;` `EA[0] = $CCCSLOAD;` `EA[1] = ($CCCSLOAD) | 0x8;`		
	Except In	`// 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);` `}`		
	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*

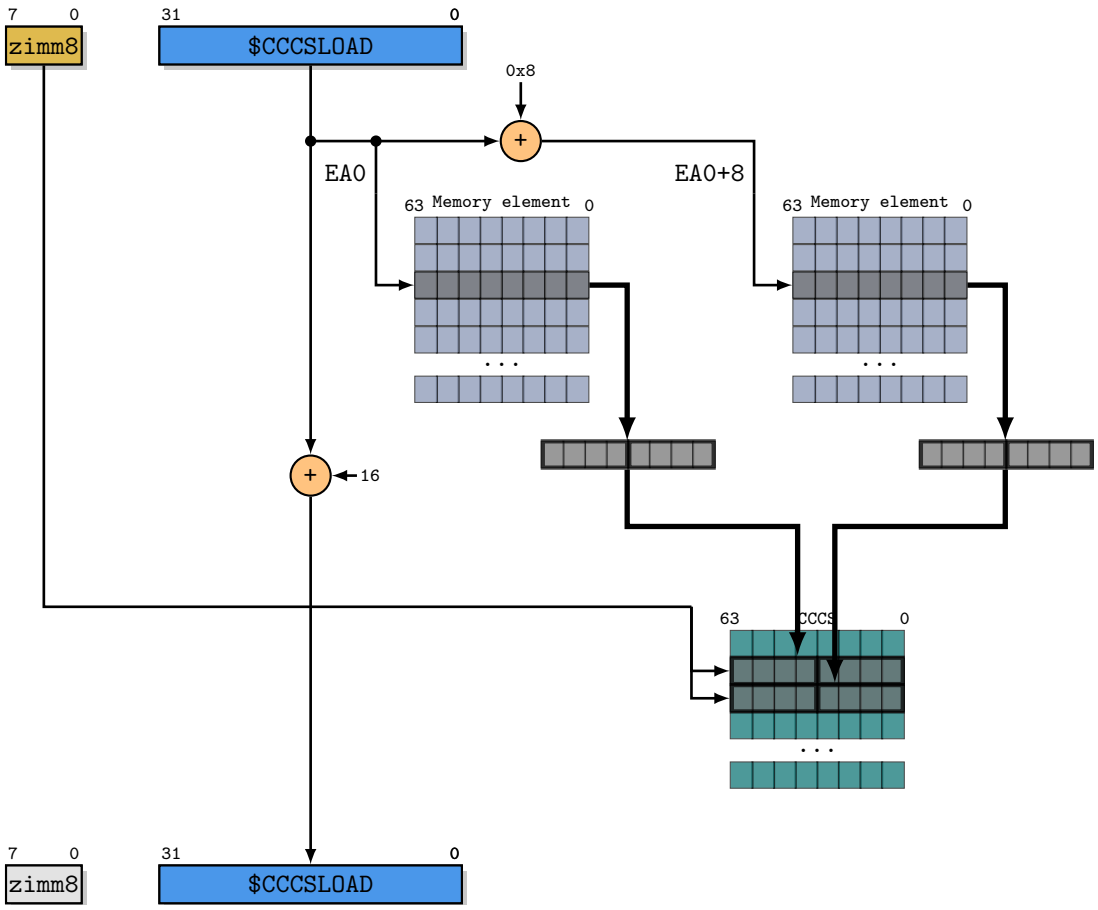


Fig. 3.69: ld128putcs

3.7.5.4 Single-store

Table 3.259: Stores from MRF addressing modes

Index	Description	Instructions
0	instr src0, src1, src2	*stm32*
1	instr src0, src1, src2, imm0	*st32*
2	instr src0, src1, srcDst0+=, imm0	*st32step*
3	instr src0, srcDst0+=, src1	*stm32step*

Table 3.260: Stores from MRF

	Addressing mode			
Access bits	**0**	**1**	**2**	**3**
32	✓ (w, s)	✓ (w, s)	✓ (w, s)	✓ (w, s)

Table 3.261: Stores from ARF addressing modes

Index	Description	Instructions
0	instr src0, src1, src2, imm0	*st32*, *st64*
1	instr src0, src1, src2, src3	*st32*, *st64*
2	instr src0, src1, srcDst0+=, imm0	*st32step*, *st64step*
3	instr src0, src1, srcDst0+=, src2	*st32step*, *st64step*
4	instr src0, srcDst0+=, src1, imm0	*st64pace*

Table 3.262: Stores from ARF

	Addressing mode				
Access bits	**0**	**1**	**2**	**3**	**4**
32	✓	✓	✓	✓	
64	✓	✓	✓	✓	✓

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 3.263: *st32* instruction definition

st32		both	main
Syntax	`st32 $m`*Src0*`, $m`*Base0*`, $m`*Delta0*`,` *zimm12*		
st32		**worker**	**main**
Syntax	`st32 $a`*Src0*`, $m`*Base0*`, $m`*Delta0*`, $m`*Offset0* `st32 $a`*Src0*`, $m`*Base0*`, $m`*Delta0*`,` *zimm12*		
Semantics	**Prepare** **Except In** **Memory**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <(($mOffset0 << 2), (zimm12 << 2))>;` `DataWord op3 = <($aSrc0, $mSrc0)>;` `EA[0] = op0 + op1 + op2;` `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);` `}` `uint32_t data = op3;` `storeWord(EA[0], data);`		

Function references: *TMem_IsValidAddress*

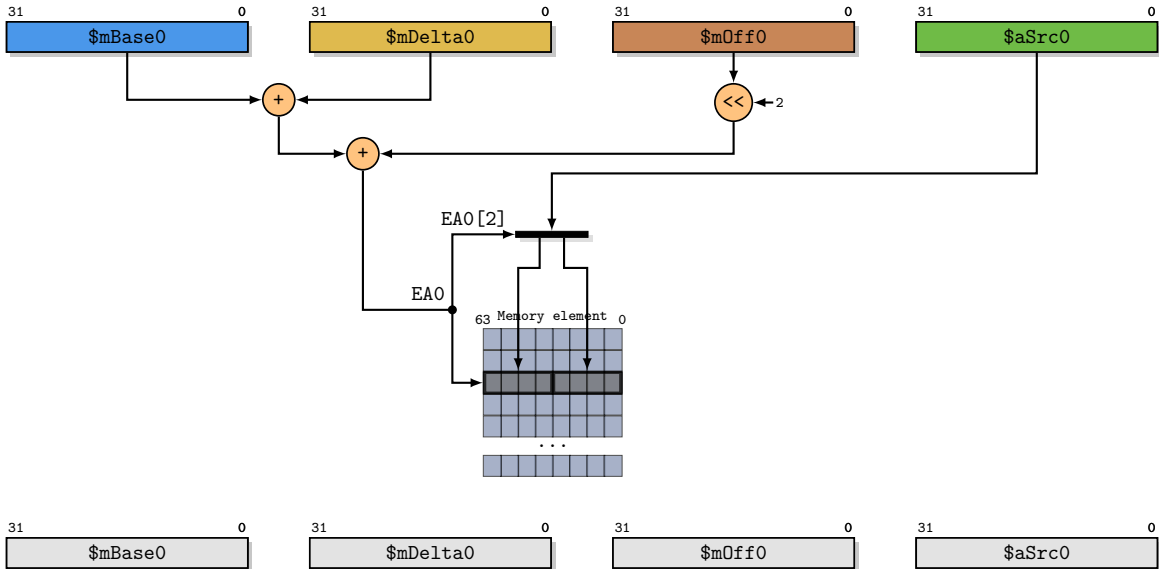


Fig. 3.70: st32 example

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 3.264: *stm32* instruction definition

stm32			both	main
Syntax	`stm32 $m`*Src0*`, $m`*Base0*`, $m`*Offset0*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = ($mOffset0 << 2);` `DataWord op2 = $mSrc0;` `EA[0] = op0 + op1;`		
	Except In	`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);` `}`		
	Memory	`uint32_t data = op2;` `storeWord(EA[0], data);`		

Function references: *TMem_IsValidAddress*

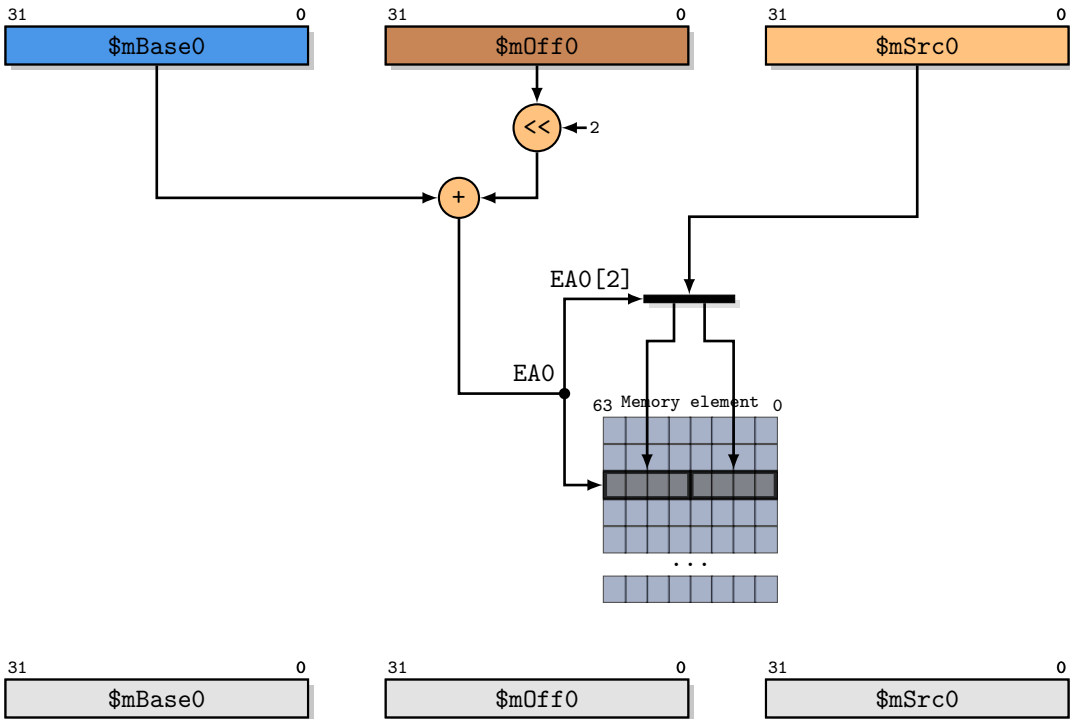


Fig. 3.71: stm32

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 3.265: *st64* instruction definition

st64			worker	main
Syntax	st64 $a*Src0*:*Src0+1*, $m*Base0*, $m*Delta0*, $m*Offset0* st64 $a*Src0*:*Src0+1*, $m*Base0*, $m*Delta0*, *zimm12*			
Semantics	**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);`		

Function references: *TMem_IsValidAddress*

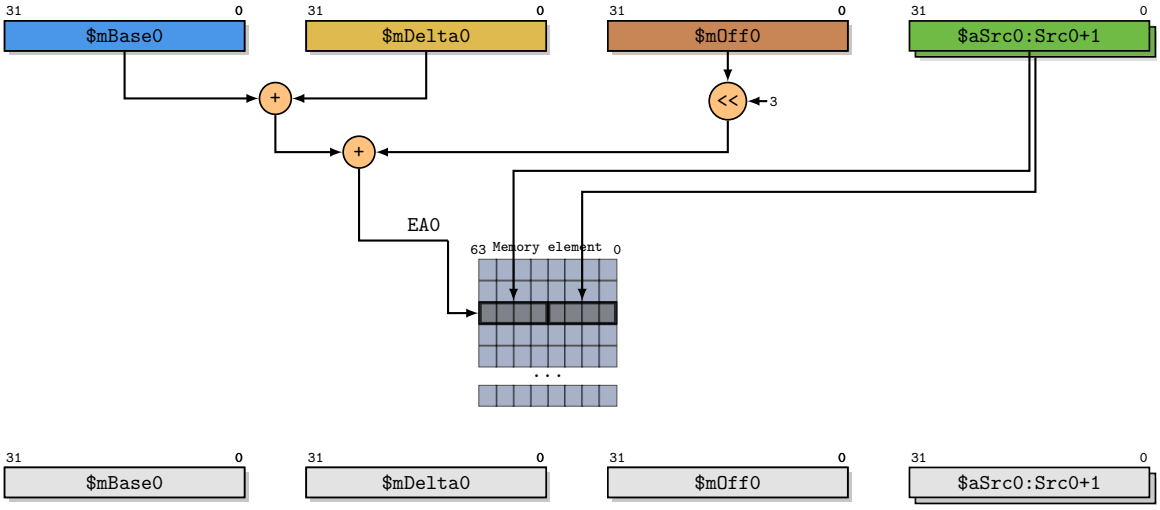


Fig. 3.72: st64 example

3.7.5.4.4 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 3.266: *st32step* instruction definition

st32step			both	main
Syntax	`st32step $m`*Src0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
st32step			worker	main
Syntax	`st32step $a`*Src0*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `st32step $a`*Src0*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 2), ($mStride0 << 2))>;` `DataWord op3 = <($mSrc0, $aSrc0)>;` `EA[0] = op0 + op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x3) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint32_t data = op3;` `storeWord(EA[0], data);`		
	Commit	`$mDelta0 = nextAddr;`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

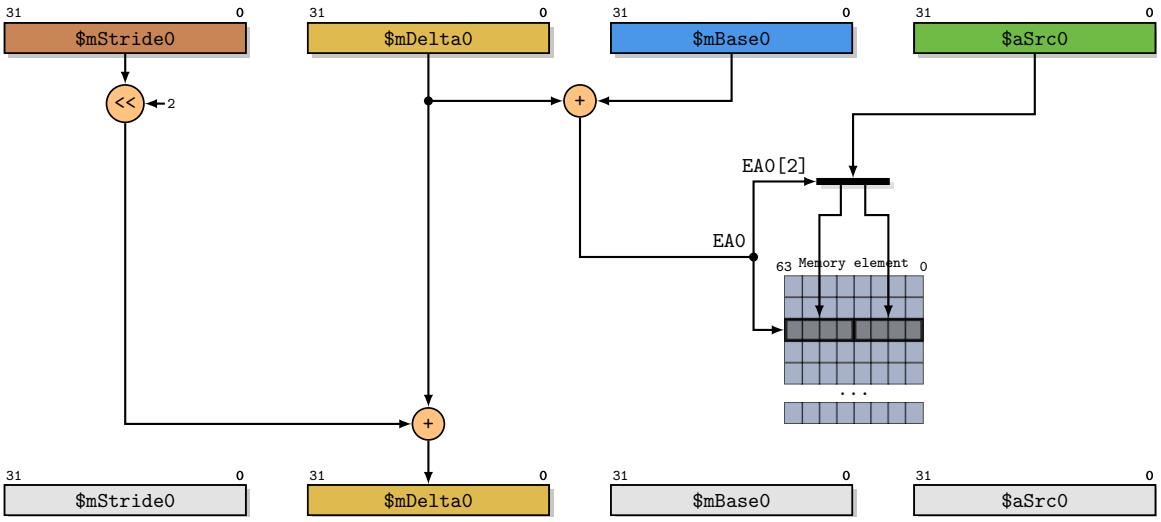


Fig. 3.73: st32step example

3.7.5.4.5 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 3.267: *stm32step* instruction definition

stm32step			both	main
Syntax	`stm32step $mSrc0, $mBase0+=, $mStride0`			
Semantics	**Prepare**	`DataWord op2 = $mSrc0;` `DataWord op1 = $mBase0;` `SignedDataWord op0 = ($mStride0 << 2);` `EA[0] = op1;`		
	Except In	`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);` `}`		
	Compute	`DataWord nextAddr = op1 + op0;`		
	Memory	`uint32_t data = op2;` `storeWord(EA[0], data);`		
	Commit	`$mBase0 = nextAddr;`		

Function references: *TMem_IsValidAddress*

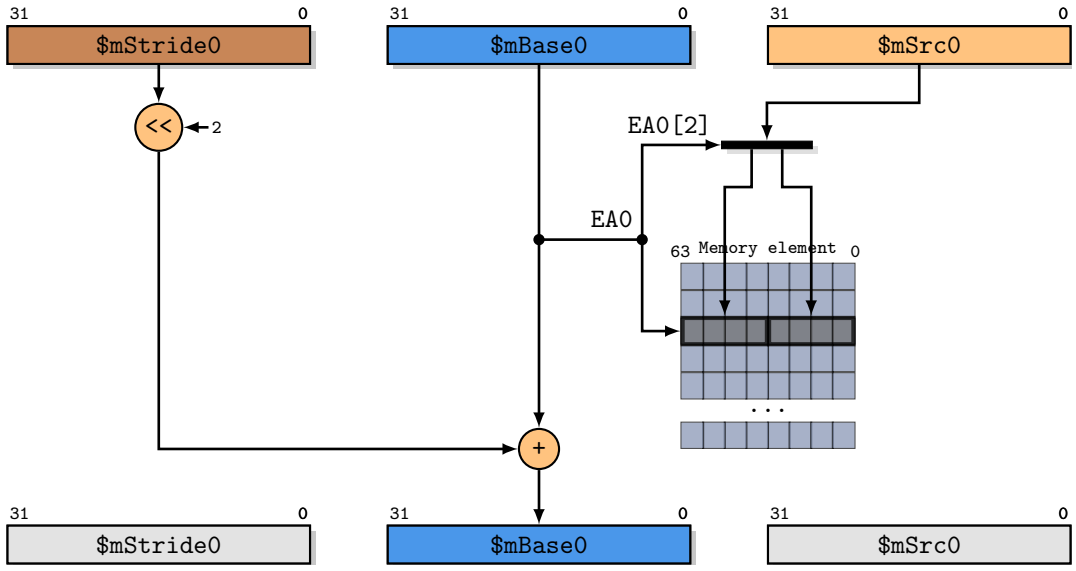


Fig. 3.74: stm32step

3.7.5.4.6 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 3.268: *st64step* instruction definition

st64step			worker	main
Syntax	`st64step $a`*Src0*`:`*Src0+1*`, $m`*Base0*`, $m`*Delta0*`+=, $m`*Stride0* `st64step $a`*Src0*`:`*Src0+1*`, $m`*Base0*`, $m`*Delta0*`+=, `*simm8*			
Semantics	**Prepare**	`DataWord op0 = $mBase0;` `DataWord op1 = $mDelta0;` `DataWord op2 = <((Tile_SignExtend(simm8, 8) << 3), ($mStride0 << 3))>;` `array<DataWord,2> op3 = {` `  $aSrc0:Src0+1[0],` `  $aSrc0:Src0+1[1]` `};` `EA[0] = op0 + op1;`		
	Except In	`if (!TMem_IsValidAddress(EA[0])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[0] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[0])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`DataWord nextAddr = op1 + op2;`		
	Memory	`uint64_t data = (((uint64_t)op3[1]) << 32ULL) |` `  ((uint64_t)op3[0]);` `storeDoubleWord(EA[0], data);`		
	Commit	`$mDelta0 = nextAddr;`		

Function references: *Tile_SignExtend* , *TMem_IsValidAddress*

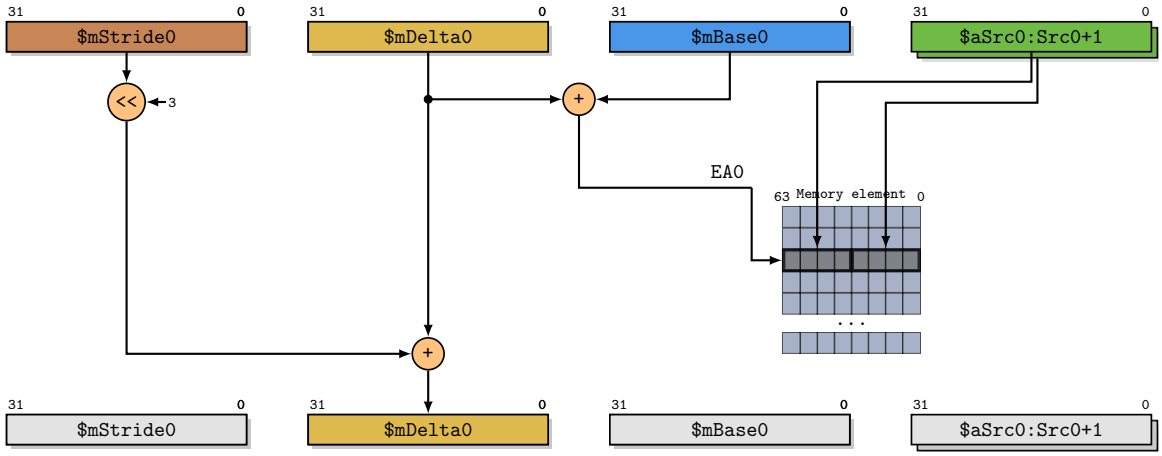


Fig. 3.75: st64step example

3.7.5.4.7 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 3.269: *st64pace* instruction definition

st64pace			worker	main
Syntax	`st64pace $a`*Src0*`:`*Src0+1*`, $m`*Addr0*`:`*Addr0+1*`+=, $m`*Stride0*`, `*Strimm2*			
Semantics	**Prepare**	`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],` `  $aSrc0:Src0+1[1]` `};` `array<DataWord,3> addrs;` `EA[2] = Tile_ExtractPackedAddress(op1, 2);`		
	Except In	`if (!TMem_IsValidAddress(EA[2])) {` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (EA[2] & 0x7) {` `  // Misaligned address` `  EXCEPT(TEXCPT_INVALID_ADDR);` `} else if (hadMemoryConflict(EA[2])) {` `  EXCEPT(TEXCPT_CONFLICT);` `}`		
	Compute	`// Auto-increment address` `int32_t stride = Tile_ExtractPackedStride(op0, op2 & 0x3);` `addrs[0] = Tile_ExtractPackedAddress(op1, 0); // Unchanged` `addrs[1] = Tile_ExtractPackedAddress(op1, 1); // Unchanged` `addrs[2] = (EA[2] + (stride * 8)) & TMEM_FULL_ADDRESS_MASK;`		
	Memory	`uint64_t data = (((uint64_t)op3[1]) << 32ULL) |` `  ((uint64_t)op3[0]);` `storeDoubleWord(EA[2], data);`		
	Commit	`$mAddr0:Addr0+1 = {` `  Tile_TripleAddressPack_Lower(addrs),` `  Tile_TripleAddressPack_Upper(addrs)` `};`		

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

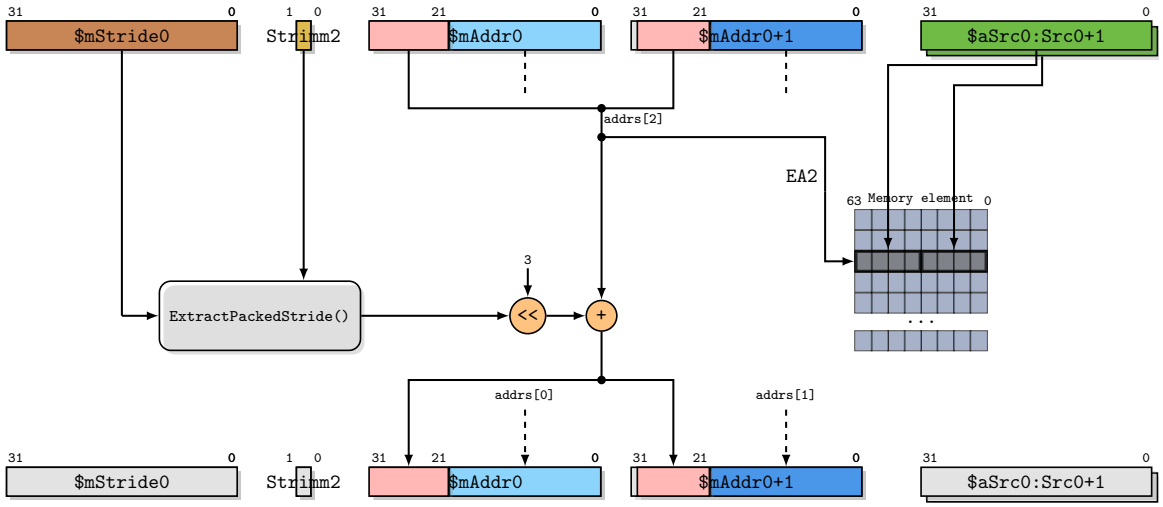


Fig. 3.76: st64pace

st64pace occurs in the following code examples:

- *f16v4sisoslic example part 1*
- *f16v4sisoslic example part 2*
- *f32sisoslic example*

3.7.6 System

Table 3.270: system instructions summary

Mnemonic	Super?	Worker?	main?	aux?	Brief
get	✓	✓	✓	✗	Lower control register read
put	✓	✓	✓	✗	Write to a lower control register
run	✓	✗	✓	✗	Launch a worker thread
runall	✓	✗	✓	✗	Launch a batch of worker threads
trap	✓	✓	✓	✗	Patched BREAKPOINT
uget	✗	✓	✗	✓	Upper control register read
uput	✗	✓	✗	✓	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 3.271: *get* instruction definition

get			both	main
Syntax	`get $m`*Dst0*`, `*zimm8*			
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 3.272: *put* instruction definition

put			both	main
Syntax	put *zimm8*, $m*Src0*			
Semantics	**Prepare**	`DataWord op0 = $mSrc0;` `DataWord op1 = zimm8;`		
	Except In	`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))) {` `  // Attempt to switch mux during reception of a TLink packet` `  EXCEPT(TEXCPT_EXCONF);` `}`		
	Commit	`writeState(op1, op0);`		

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 $m*Entry0*, using the vertex address calculated by summing:

1. the register operand $m*VBase0*
2. the 16-bit immediate offset *zimm16* × 4
3. the constant TMEM_BASE_ADDR

(i.e. the address formed by adding the register value $m*VBase0* to the scaled immediate offset *zimm16* is relative to TMEM_BASE_ADDR)

*Exception event*s will be raised for any of the following conditions:

- $m*Entry0* is not 4-byte aligned
- $m*VBase0* is not 4-byte aligned

- $m*Entry0* 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 3.273: *run* instruction definition

run			supervisor	main
Syntax	`run $mEntry0, $mVBase0, zimm16`			
Semantics	**Prepare**	`DataWord op0 = $mEntry0;` `DataWord op1 = $mVBase0;` `DataWord op2 = (zimm16 << 2);`		
	Except In	`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 $PC` `  EXCEPT(TEXCPT_INVALID_OP);` `}`		
	Commit	`DataWord vertexBase = op1 + op2 + TMEM_BASE_ADDR;` `Context &worker = getNextFreeWorker();` `worker.$PC = op0;` `worker.$VERTEX_BASE = vertexBase;` `worker.$FP_STS = CSR_W_FP_STS__RESET;` `worker.$FP_CTL = $FP_ICTL;` `worker.$FP_NFMT = $FP_INFMT;` `worker.$FP_SCL = $FP_ISCL;` `worker.setRunMode(TRUNM_EXECUTING);`		

Architectural state references: *$PC* , *$VERTEX_BASE* , *$FP_STS* , *$FP_CTL* , *$FP_NFMT* , *$FP_SCL* , *$FP_ICTL* , *$FP_INFMT* , *$FP_ISCL*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.6.4 runall

Allocate execution time and context state to a batch of worker *thread*s:

- The total number of *thread*s launched is equal to the number of hardware *worker* contexts (CTXT_WORKERS)
- All *thread*s use the same entry point ($m*Entry0*)
- The value of *$VERTEX_BASE* assigned to the first allocated *worker* is provided by the register $m*VBase0*
- The value of *$VERTEX_BASE* assigned to every other *worker* is:
  - $m*VBase0* + ($n \times$ *zimm16* $\times$ 4) ($n \in [1, CTXT\_WORKERS - 1]$)

*Exception event*s will be raised for any of the following conditions:

- $m*Entry0* is not 4-byte aligned
- $m*VBase0* is not 4-byte aligned

- $m*Entry0* is not a valid, executable address
- There are any active Worker contexts

If there are active Worker contexts, **$SSR.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 3.274: *runall* instruction definition

runall			supervisor	main
Syntax	`runall $m`*Entry0*`, $m`*VBase0*`, `*zimm16*			
Semantics	**Prepare**	`DataWord op0 = $mEntry0;` `DataWord op1 = $mVBase0;` `DataWord op2 = zimm16;`		
	Except In	`// Raise an exception if there are any active worker contexts` `bool raiseExcpt = false;` `for (int w = 0; w < CTXT_WORKERS; w++) {` `  if (((`$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)` `  `$SSR`.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.`$FP_NFMT` = `$FP_INFMT`;` `  worker.`$FP_SCL` = `$FP_ISCL`;` `  worker.setRunMode(TRUNM_EXECUTING);` `}`		

Architectural state references: *$PC* , *$VERTEX_BASE* , *$FP_STS* , *$FP_CTL* , *$FP_NFMT* , *$FP_SCL* , *$CTXT_STS* , *$FP_ICTL* , *$FP_INFMT* , *$FP_ISCL*

Function references: *TMem_IsValidAddress* , *TMem_AddressIsExecutable*

3.7.6.5 trap

Unconditionally raise a *patched breakpoint exception event*.

Table 3.275: *trap* instruction definition

trap			both	main
Syntax	`trap` *zimm4*			
Semantics	**Prepare**	`DataWord op0 = zimm4;`		
	Except In	`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);` `  }` `}`		

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 3.276: *uget* instruction definition

uget			worker	aux
Syntax	`uget $a`*Dst0*`,` *zimm8*			
Semantics	**Prepare**	`DataWord op1 = zimm8;` `LOADED_STATE = readState(TREG_UPPER_CSR_SPACE_BASE + op1);`		
	Except In	`if (0 != $REPEAT_COUNT && !isExecutingTDI()) {` `  // Cannot execute get in the body of rpt` `  EXCEPT(TEXCPT_INVALID_INSTR);` `}` `if (!TReg_IsValidCSR(TREG_UPPER_CSR_SPACE_BASE + op1, false)) {` `  EXCEPT(TEXCPT_INVALID_OP);` `}`		
	Commit	`$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 3.277: *uput* instruction definition

uput			worker	aux
Syntax	`uput` *zimm8*, `$a`*Src0*			
Semantics	**Prepare**	`DataWord op0 = $aSrc0;` `DataWord op1 = zimm8;`		
	Except In	`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);` `}`		
	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*
- *ldb8*
- *ldb8step*
- *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*
- *f16v2tof8*
- *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*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4stacc*
- *f16v4sub*
- *f16v4sufromui*
- *f16v4sum*
- *f16v8absacc*
- *f16v8acc*
- *f16v8sqacc*
- *f16v8tof8*
- *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*
- *f8v2tof16*
- *f8v4class*
- *f8v4tof16*
- *f8v8hihov4amp*
- *f8v8hihov4slic*
- *not*
- *not64*
- *or*
- *or64*
- *roll16*
- *roll32*
- *roll8l*
- *roll8r*
- *setzi*
- *shuf8x8hi*
- *shuf8x8lo*
- *sort4x16hi*
- *sort4x16lo*
- *sort4x32hi*
- *sort4x32lo*
- *sort8*
- *sort8x8hi*
- *sort8x8lo*
- *swap8*
- *uget*
- *uput*
- *urand32*
- *urand64*

3.8.1.3 both

- *and*
- *andc*
- *or*
- *roll16*
- *roll8l*
- *roll8r*
- *setzi*
- *shuf8x8hi*
- *shuf8x8lo*
- *sort4x16hi*
- *sort4x16lo*
- *sort8*
- *sort8x8hi*
- *sort8x8lo*
- *swap8*

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*
- *f16v2tof8*
- *f16v4absadd*
- *f16v4add*
- *f16v4clamp*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4mix*
- *f16v4mul*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4sub*
- *f16v4sum*
- *f16v8tof8*
- *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*
- *f8v2tof16*
- *f8v4tof16*
- *f8v8hihov4amp*
- *f8v8hihov4slic*

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*
- *f16v2tof8*
- *f16v4absadd*
- *f16v4add*
- *f16v4gacc*
- *f16v4hihoamp*
- *f16v4hihoslic*
- *f16v4hihov4amp*
- *f16v4hihov4slic*
- *f16v4mix*
- *f16v4mul*
- *f16v4sisoamp*
- *f16v4sisoslic*
- *f16v4sub*
- *f16v8tof8*
- *f32absadd*
- *f32add*
- *f32div*
- *f32exp*
- *f32exp2*
- *f32mul*
- *f32sisoamp*
- *f32sisoslic*
- *f32sisov2amp*
- *f32sisov2slic*
- *f32sub*
- *f32tof16*
- *f32v2absadd*
- *f32v2add*
- *f32v2axpy*
- *f32v2gina*
- *f32v2mul*
- *f32v2sub*
- *f32v2tof16*
- *f32v4tof16*
- *f8v2tof16*
- *f8v4tof16*
- *f8v8hihov4amp*
- *f8v8hihov4slic*

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 3.278: Floating-point operations

	8-bit			16-bit				32-bit		
Operation	**v2**	**v4**	**v8**	**v1**	**v2**	**v4**	**v8**	**v1**	**v2**	**v4**
absacc						✓	✓			✓
acc						✓	✓			✓
aop									✓	
axpy									✓	
cmac					✓	✓				
gacc						✓				
gina					✓				✓	
hihoamp						✓				
hihoslic						✓				
hihov4amp			✓			✓				
hihov4slic			✓			✓				
istacc						✓				
mac								✓	✓	
mix						✓				
sihoamp						✓				
sihoslic						✓				
sisoamp						✓		✓		
sisoslic						✓		✓		
sisov2amp								✓		
sisov2slic								✓		
sqacc							✓			✓
stacc						✓				
grand					✓				✓	
rmask						✓			✓	
tof16								✓	✓	✓
tof8					✓		✓			
absadd					✓	✓		✓	✓	
absmax					✓	✓		✓	✓	
add					✓	✓		✓	✓	
clamp					✓	✓		✓	✓	
class		✓			✓	✓		✓	✓	
div								✓		
exp					✓			✓		
exp2					✓			✓		
int								✓		
ln					✓			✓		
log2					✓			✓		
max					✓	✓		✓	✓	
maxc					✓	✓				
min					✓	✓		✓	✓	

Continued on next page

Table 3.278 – continued from previous page

	8-bit			16-bit				32-bit		
Operation	**v2**	**v4**	**v8**	**v1**	**v2**	**v4**	**v8**	**v1**	**v2**	**v4**
mul					✓	✓		✓	✓	
oorx								✓		
oox								✓		
sigm					✓			✓		
sqrt								✓		
sub					✓	✓		✓	✓	
sum					✓	✓				
tanh					✓			✓		
cmpeq					✓	✓		✓	✓	
cmpge					✓	✓		✓	✓	
cmpgt					✓	✓		✓	✓	
cmple					✓	✓		✓	✓	
cmplt					✓	✓		✓	✓	
cmpne					✓	✓		✓	✓	
fromi32								✓		
fromui32								✓		
sufromui					✓	✓		✓	✓	
tof16	✓	✓								
tof32				✓	✓					
toi32								✓		
toui32								✓		

3.8.5 8-bit Floating-Point Operations x Number Format and Vector Length

Table 3.279: 8-bit floating-point operations

Operation	v2	v4	v8
hihov4amp			✓
hihov4slic			✓
class		✓	
tof16	✓	✓	

3.8.6 16-bit Floating-Point Operations x Number Format and Vector Length

Table 3.280: 16-bit floating-point operations

Operation	v1	v2	v4	v8
absacc			✓	✓
acc			✓	✓
cmac		✓	✓	
gacc			✓	
gina		✓		
hihoamp			✓	
hihoslic			✓	
hihov4amp			✓	
hihov4slic			✓	
istacc			✓	
mix			✓	
sihoamp			✓	
sihoslic			✓	
sisoamp			✓	
sisoslic			✓	
sqacc				✓
stacc			✓	
grand		✓		
rmask			✓	
tof8		✓		✓
absadd		✓	✓	
absmax		✓	✓	
add		✓	✓	
clamp		✓	✓	
class		✓	✓	
exp		✓		
exp2		✓		
ln		✓		
log2		✓		
max		✓	✓	
maxc		✓	✓	
min		✓	✓	
mul		✓	✓	
sigm		✓		
sub		✓	✓	
sum		✓	✓	
tanh		✓		
cmpeq		✓	✓	
cmpge		✓	✓	
cmpgt		✓	✓	
cmple		✓	✓	

Continued on next page

Table 3.280 – continued from previous page

Operation	v1	v2	v4	v8
cmplt		✓	✓	
cmpne		✓	✓	
sufromui		✓	✓	
tof32	✓	✓		

3.8.7 32-bit Floating-Point Operations x Number Format and Vector Length

Table 3.281: 32-bit floating-point operations

Operation	v1	v2	v4
absacc			✓
acc			✓
aop		✓	
axpy		✓	
gina		✓	
mac	✓	✓	
sisoamp	✓		
sisoslic	✓		
sisov2amp	✓		
sisov2slic	✓		
sqacc			✓
grand		✓	
rmask		✓	
tof16	✓	✓	✓
absadd	✓	✓	
absmax	✓	✓	
add	✓	✓	
clamp	✓	✓	
class	✓	✓	
div	✓		
exp	✓		
exp2	✓		
int	✓		
ln	✓		
log2	✓		
max	✓	✓	
min	✓	✓	
mul	✓	✓	
oorx	✓		
oox	✓		
sigm	✓		
sqrt	✓		
sub	✓	✓	
tanh	✓		
cmpeq	✓	✓	
cmpge	✓	✓	
cmpgt	✓	✓	
cmple	✓	✓	
cmplt	✓	✓	
cmpne	✓	✓	
fromi32	✓		

Continued on next page

Table 3.281 – continued from previous page

Operation	v1	v2	v4
fromui32	✓		
sufromui	✓	✓	
toi32	✓		
toui32	✓		

CHAPTER

FOUR

IMPLEMENTATION SPECIFICS

4.1 [IPU21]

4.1.1 General

4.1.1.1 TileRunMode

Context run modes. See *Run Modes*.

Table 4.1: Enumeration: TileRunMode

Identifier	Value	Description
TRUNM_INACTIVE	0	Inactive run mode
TRUNM_EXECUTING	1	Executing run mode
TRUNM_EXCEPTED	2	Excepted run mode
TRUNM_REPEATING	3	Repeating run mode

4.1.1.2 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: }
```

4.1.1.3 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: }
```

4.1.1.4 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: }
```

4.1.1.5 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: }
```

4.1.1.6 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: }
```

4.1.1.7 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 TileInstrPhase

Instruction lifespan phase.

Table 4.2: Enumeration: TileInstrPhase

Identifier	Value	Description
TPHASE_FETCH	0	Instruction opcode read from *Tile Memory*
TPHASE_ISSUE	1	Instruction issue phase
TPHASE_EXECUTE	2	Instruction functional execution
TPHASE_RETIRE	3	Instruction commit and retirement

4.1.2.2 TInstr_GetLatency

```
unsigned TInstr_GetLatency(const Opcode opcode, const float src0[], const float src1[], const unsigned brTargetPc)
{
  // Only worker instructions are supported

  unsigned latency = 1;
  if (isFtuOpcode(opcode)) {
    latency = TInstr_GetFtuLatency(opcode, (float *)src0, (float *)src1);
  } else if (isBranchOpcode(opcode) || opcode == Opcode::call_mmmn_zi) {
    latency = TInstr_GetBranchLatency(brTargetPc);
  } else if (opcode == Opcode::get_mmmn_zi) {
    latency = TInstr_GetGetLatency();
  }

  return latency;
}
```

4.1.3 Floating_Point

4.1.3.1 Transcendental, Divide, Square-Root and Reciprocal Instructions

Table 4.3: Transcendental, divide, square-root and reciprocal instructions

Instruction	Implemented?	Latency	Domain	Monotonic?	Accuracy
f32div	✓	3		✓	*TFPU_ACCURATE*
f32sqrt	✓	5	$x \in [0, +\infty]$	✓	*TFPU_ACCURATE*
f32oox	✓	3	$x \in [-\infty, +\infty]$	✓	*TFPU_ACCURATE*
f32oorx	✓	4	$x \in (0, +\infty]$	✗	*TFPU_INACCURATE_1P5*
f32log2	✓	6	$x \in [0, +\infty]$	✓	*TFPU_INACCURATE_1P5*
f32ln	✓	6	$x \in [0, +\infty]$	✓	*TFPU_INACCURATE_1P5*
f32exp2	✓	3	$x \in [-\infty, +\infty]$	✓	*TFPU_INACCURATE_1P5*
f32exp	✓	3	$x \in [-\infty, +\infty]$	✓	*TFPU_INACCURATE_1P5*
f32sigm	✓	5	$x \in [-\infty, +\infty]$	✓	*TFPU_INACCURATE_2P5*
f32tanh	✓	5	$x \in [-\infty, +\infty]$	✓	*TFPU_INACCURATE_1P5*
f16v2log2	✓	2	$x \in [0, 65504]$	✓	*TFPU_ACCURATE*
f16v2ln	✓	2	$x \in [0, 65504]$	✓	*TFPU_ACCURATE*
f16v2exp2	✓	2	$x \in [-65504, 65504]$	✓	*TFPU_ACCURATE*
f16v2exp	✓	2	$x \in [-65504, 65504]$	✓	*TFPU_ACCURATE*
f16v2sigm	✓	2	$x \in [-65504, 65504]$	✓	*TFPU_INACCURATE_16P5*
f16v2tanh	✓	1	$x \in [-65504, 65504]$	✓	*TFPU_ACCURATE*

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 4.4: Floating_Point parameters

Parameter name	Value	Description
`TFPU_FP32_FALSE`	0	False representation for *Single-precision*.
`TFPU_NUM_ACCUM`	32/0x20	The total number of individual accumulators

4.1.3.3 TileRoundMode

Floating-point rounding modes.

Table 4.5: 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 4.5 – 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 4.6: 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_F32_Decompose

break *op0* into its constituent parts

```
1: void TFPU_F32_Decompose(const float op0, uint32_t *sign, uint32_t *uexponent, uint32_t *mantissa)
2: {
3:   uint32_t uivalue;
4:
5:   std::memcpy(&uivalue, &op0, sizeof(uivalue));
6:   *sign      = TFPU_FIELD(F32_S, uivalue);
7:   *uexponent = TFPU_FIELD(F32_E, uivalue);
8:   *mantissa  = TFPU_FIELD(F32_M, uivalue);
9: }
```

4.1.3.6 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: }
```

4.1.3.7 TFPU_BitsFromF32

float to raw 32-bit bit-field conversion.

Returns: raw bit representation of f32 input val

```
uint32_t TFPU_BitsFromF32(float val, const bool flushDenorms=true)
{
  uint32_t result;

  if (flushDenorms) {
    val = TFPU_FlushFP32DenormToZero(val);
  }

  std::memcpy(&result, &val, sizeof(result));
  return result;
}
```

4.1.3.8 TFPU_F32_QNan

Returns: internally generated single-precision quiet NaN

```
float TFPU_F32_QNan()
{
  return TFPU_F32FromBits(TFPU_F32_GEN_QNAN);
}
```

4.1.3.9 TFPU_F32_IsQNan

Returns: true if the single-precision value *op0* is a quiet NaN. false otherwise

```
bool TFPU_F32_IsQNan(const float op0)
{
  uint32_t uivalue;
  std::memcpy(&uivalue, &op0, sizeof(uivalue));
  return std::isnan(op0) && (((uivalue >> (TFPU_F32_E_OFFSET - 1)) & 1) != 0);
}
```

4.1.3.10 TFPU_F32_IsSNan

Returns: true if the single-precision value *op0* is a signaling NaN. false otherwise

```
bool TFPU_F32_IsSNan(const float op0)
{
  uint32_t uivalue;
  std::memcpy(&uivalue, &op0, sizeof(uivalue));
  return std::isnan(op0) && (((uivalue >> (TFPU_F32_E_OFFSET - 1)) & 1) == 0);
}
```

4.1.3.11 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).

```
float TFPU_RoundFP64ToFmt(double value, TileFloatFormat_t fmt, TileRoundMode_t mode)
{
  uint64_t sign, mantissa, uexponent;
  int sexponent, oflow, effectivePrecision, minNormExp, minDenormExp;

  if (TFPU_FP16 == fmt) {
    effectivePrecision = TFPU_F16_M_SIZE;
    minNormExp = TFPU_F16_MIN_NORM_EXP;
    minDenormExp = TFPU_F16_MIN_DENORM_EXP;

  } else {
    effectivePrecision = TFPU_F32_M_SIZE;
    minNormExp = TFPU_F32_MIN_NORM_EXP;
    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:
48:       } else {
49:         uexponent++;
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: }
```

4.1.3.12 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: }
```

4.1.3.13 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: }
```

4.1.3.14 TFPU_SNanCheck

Check an op for the presence of a signaling NaN. Return invalid operation exception flag if signaling NaN detected.

```
1: TileFPException_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: }
```

4.1.3.15 TFPU_GenSNanCheck

Check ops for the presence of a signaling NaN. Return invalid operation exception flag if signaling NaN detected.

```
 1: TileFPException_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: }
```

4.1.3.16 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

```
uint32_t TFPU_GenOFLOCheck(double result, TileFloatFormat_t format, bool nanoo)
{
  uint64_t osign, omantissa, oexponent, nmantissa, nexponent;
  double rresult, maxNorm;
  int tmSize;
  int oflow = 0;

  if (std::isnan(result)) {
    return TFPEXCPT_NONE;
  }

  if (std::fabs(result) == 0.0) {
    return TFPEXCPT_NONE;
  }

  // Extract the various parts of the double-precision result
  TFPU_F64_Decompose(result, &osign, &oexponent, &omantissa);

  switch (format) {
    case TFPU_FP16:
      tmSize = TFPU_F16_M_SIZE;
      maxNorm = exp2(TFPU_F16_MAX_NORM_EXP + 1) -
                exp2(TFPU_F16_MAX_NORM_EXP - TFPU_F16_M_SIZE);
      break;


    default:
      // TFPU_FP32:
      tmSize = TFPU_F32_M_SIZE;
      maxNorm = std::numeric_limits<float>::max();
      break;
  }

  // Construct a double-precision result with a mantissa rounded to target precision
  nexponent = oexponent;
  nmantissa = TFPU_RoundMantissa(omantissa, TFPU_F64_M_SIZE, tmSize,
                                 TFPU_ROUND_EVEN, &oflow);

  // Did we overflow the mantissa?
  if (oflow) {
    nexponent++;
  }

  rresult = TFPU_F64FromFields(osign, nexponent, nmantissa);

  if (std::abs(rresult) > maxNorm) {
    // IEEE 754-2008: 7.4
    if (nanoo) {
      return TFPEXCPT_OFLO | TFPEXCPT_INV;
    }
    return TFPEXCPT_OFLO;
  }

  return TFPEXCPT_NONE;
}
```

4.1.3.17 TFPU_GenOFLOCheckF8

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
- *tmSize*: Target mantissa size in bits
- *qMax*: Maximum value representable by the quarter-precision number
- *nanoo*: The value of NaN On Overflow (NANOO) flag

```
uint32_t TFPU_GenOFLOCheckF8(double result, int tmSize, float qMax, bool nanoo)
{
  uint64_t osign, omantissa, oexponent, nmantissa, nexponent;
  double rresult;
  int oflow = 0;

  if (std::isnan(result)) {
    return TFPEXCPT_NONE;
  }

  if (std::fabs(result) == 0.0) {
    return TFPEXCPT_NONE;
  }

  // Extract the various parts of the double-precision result
  TFPU_F64_Decompose(result, &osign, &oexponent, &omantissa);

  // Construct a double-precision result with a mantissa rounded to target precision
  nexponent = oexponent;
  nmantissa = TFPU_RoundMantissa(omantissa, TFPU_F64_M_SIZE, tmSize,
                                 TFPU_ROUND_EVEN, &oflow);

  // Did we overflow the mantissa?
  if (oflow) {
    nexponent++;
  }

  rresult = TFPU_F64FromFields(0, nexponent, nmantissa);

  if (rresult > qMax) {
    // IEEE 754-2008: 7.4
    if (nanoo) {
      return TFPEXCPT_OFLO | TFPEXCPT_INV;
    }
    return TFPEXCPT_OFLO;
  }

  return TFPEXCPT_NONE;
}
```

4.1.3.18 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.

```
bool TFPU_AACCResetValue(uint32_t &value)
{
  // Every $AACC register reset to zero
  value = 0;
  return true;
}
```

4.1.3.19 TFPU_AACCReadFlags

Check for exception conditions on accumulator read

Returns: TFPEXCPT_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

```
uint32_t TFPU_AACCReadFlags(std::vector<float> &aacc, TileFloatFormat_t format, bool nanoo)
{
  uint32_t fpExcpt = TFPEXCPT_NONE;

  for (auto iter = aacc.begin(); iter != aacc.end(); ++iter) {
    float value = *iter;

    if (std::isinf(value)) {
      // Assume that the infinity is the result of an arithmetic overflow
      // (which is otherwise invisible in Tile's AMP engine)
      fpExcpt |= TFPEXCPT_OFLO;

      if (TFPU_FP16 == format) {
        // For f16 results the infinity result is converted to qNaN
        // and Tile sets the INVALID_OPERATION flag
        fpExcpt |= TFPEXCPT_INV;
      }
    } else {
      fpExcpt |= TFPU_GenOFLOCheck(value, format, nanoo);
    }
  }

  return fpExcpt;
}
```

4.1.3.20 TFPU_F16DotProduct

Perform an inner-product on the two *f16* input vectors x and y as per the Tile AMP hardware.

```
float TFPU_F16DotProduct(const std::vector<float> &x, const std::vector<float> &y, int scale, TileFP16Fmt_t
fmt=TFPU_FP16FMT_HALF)
{
  return TFPU_HalfDotProduct(x, y, 0);
}
```

4.1.3.21 TFPU_F8DotProduct

Perform an inner-product on the two *quarter-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 quarter-precision floating-point values (represented as single-precision floats)
- y: an array of quarter-precision floating-point values (represented as single-precision floats). The number of elements in y must be at least that of x.

```
float TFPU_F8DotProduct(const std::vector<float> &x, const std::vector<float> &y, int xBias, int yBias, int scale)
{
}
```

4.1.3.22 TFPU_F32DotProductFull

Perform a full precision inner-product on the two *single-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 single-precision floating-point values
- y: an array of single-precision floating-point values. The number of elements in y must be at least that of x.

```
float TFPU_F32DotProductFull(const std::vector<float> &x, const std::vector<float> &y)
{
  float sum = -0.0;

  for (unsigned i = 0; i < x.size(); i ++) {
```

```
    float mulRes = TFPU_Mul(x[i], y[i], TFPU_FP32, TFPU_ROUND_EVEN);
    sum = TFPU_Add(sum, mulRes, TFPU_FP32);
  }

  return sum;
}
```

4.1.3.23 TFPU_F32DotProduct

Perform an inner-product on the two *single-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 single-precision floating-point values
- y: an array of single-precision floating-point values. The number of elements in y must be at least that of x.

```
float TFPU_F32DotProduct(const std::vector<float> &x, const std::vector<float> &y, TileFP32Prec_t
prec=TFPU_FP32PREC_F32)
{
  prec = prec;
  return TFPU_F32DotProductFull(x, y);
}
```

4.1.3.24 TFPU_DoAddPreExecute

```
bool TFPU_DoAddPreExecute(float op0, float op1, uint32_t *fpExcpt, float *result)
{
  bool specialCase = true;

  if (TFPU_F32_IsSNan(op0) || TFPU_F32_IsSNan(op1)) {
    // IEEE 754-2008: 6.2
    *fpExcpt |=  TFPEXCPT_INV;
    *result = TFPU_F32_QNan();

  } else if (TFPU_F32_IsQNan(op0) || TFPU_F32_IsQNan(op1)) {
    // IEEE 754-2008: 6.2.3
    *result = TFPU_F32_QNan();

  } else if (_InvCheckADD(op0, op1)) {
    *fpExcpt |= TFPEXCPT_INV;
    *result = TFPU_F32_QNan();

  } else if (std::isinf(op0)) {
    // IEEE 754-2008: 6.1
    *result = op0;

  } else if (std::isinf(op1)) {
    // IEEE 754-2008: 6.1
    *result = op1;

  } else {
    specialCase = false;
  }

  return specialCase;
}
```

4.1.3.25 TFPU_Add

```
float TFPU_Add(float op0, float op1, TileFloatFormat_t fmt, bool abs=false, bool subNotAdd=false)
{
  if (TFPU_F32_IsSNan(op0)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op0);

  } else if (TFPU_F32_IsQNan(op0)) {
    // 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: }
```

4.1.3.26 TFPU_DoAxpbyPreExecute

Floating-point exception and special-case checking for $a\mathbf{x} + b\mathbf{y}$ 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: }
```

4.1.3.27 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: }
```

4.1.3.28 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: }
```

4.1.3.29 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;
```

```
    *result = TFPU_F32_QNan();

  } else if (TFPU_F32_IsQNan(op0) || TFPU_F32_IsQNan(op1)) {
    // IEEE 754-2008: 6.2.3
    *result = TFPU_F32_QNan();

  } else if (std::isinf(op0)) {
    if (fabs(op1) == 0.0) {
      // IEEE 754-2008: 7.2
      *fpExcpt |= TFPEXCPT_INV;
      *result = TFPU_F32_QNan();

    } else {
      bool op2IsNeg = std::signbit(op1);
      if (op2IsNeg) {
        // IEEE 754-2008: 6.1
        *result = -op0;

      } else {
        // IEEE 754-2008: 6.1
        *result = op0;
      }
    }

  } else if (std::isinf(op1)) {
    if (fabs(op0) == 0.0) {
      // IEEE 754-2008: 7.2
      *fpExcpt |= TFPEXCPT_INV;
      *result = TFPU_F32_QNan();

    } else {
      bool op1IsNeg = std::signbit(op0);
      if (op1IsNeg) {
        // IEEE 754-2008: 6.1
        *result = -op1;

      } else {
        // IEEE 754-2008: 6.1
        *result = op1;
      }
    }

  } else {
    specialCase = false;
  }

  return specialCase;
}
```

4.1.3.30 TFPU_Mul

```
double TFPU_Mul(float op0, float op1,  TileFloatFormat_t fmt, TileRoundMode_t rmode)
{
  bool isNeg0 = std::signbit(op0);
  bool isNeg1 = std::signbit(op1);

  if (TFPU_F32_IsSNan(op0)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op0);

  } else if (TFPU_F32_IsQNan(op0)) {
    // IEEE 754-2008: 6.2.3
    return TFPU_F32_QuietenNan(op0);

  } else if (TFPU_F32_IsSNan(op1)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op1);

  } else if (TFPU_F32_IsQNan(op1)) {
    // IEEE 754-2008: 6.2.3
    return TFPU_F32_QuietenNan(op1);

  } else if (_InvCheckMUL(op0, op1)) {
    return TFPU_F32_QNan();

```

```
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: }
```

4.1.3.31 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: }
```

4.1.3.32 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: }
```

4.1.3.33 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: }
```

4.1.3.34 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: }
```

4.1.3.35 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: }
```

4.1.3.36 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: }
```

4.1.3.37 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: }
```

4.1.3.38 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 (std::isinf(op0)) {
15:     // IEEE 754-2008: 9.2.1
16:     bool op1IsNeg = std::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: }
```

4.1.3.39 TFPU_F32Sigmoid

```
1: float TFPU_F32Sigmoid(float op0, TileRoundMode_t rmode)
2: {
3:   return _FTU_F32sigm(op0, rmode);
4: }
```

4.1.3.40 TFPU_F16Sigmoid

```
1: float TFPU_F16Sigmoid(float op0, TileRoundMode_t rmode)
2: {
3:   return _FTU_F16sigm(op0, rmode);
4: }
```

4.1.3.41 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: }
```

4.1.3.42 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: }
```

4.1.3.43 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: }
```

4.1.3.44 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: }
```

4.1.3.45 TFPU_F32Tanh

```
1: float TFPU_F32Tanh(float op0, TileRoundMode_t rmode)
2: {
3:   return _FTU_F32tanh(op0, rmode);
4: }
```

4.1.3.46 TFPU_F16Tanh

```
1: float TFPU_F16Tanh(float op0, TileRoundMode_t rmode)
2: {
3:   return _MPFR_F16tanh(op0, rmode);
4: }
```

4.1.3.47 TFPU_F32Recip

```
1: float TFPU_F32Recip(float op0, TileRoundMode_t rmode)
2: {
3:   return _MPFR_F32recip(op0, rmode);
4: }
```

4.1.3.48 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: }
```

4.1.3.49 TFPU_F32RSqrt

```
1: float TFPU_F32RSqrt(float op0, TileRoundMode_t rmode)
2: {
3:   return _FTU_F32rsqrt(op0, rmode);
4: }
```

4.1.3.50 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*.

```
float TFPU_Min(float op0, float op1)
{
  bool isNeg0 = std::signbit(op0);
  bool isNeg1 = std::signbit(op1);

  if (TFPU_F32_IsSNan(op0)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op0);

  } else if (TFPU_F32_IsSNan(op1)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op1);

  } else if (TFPU_F32_IsQNan(op1)) {
    // IEEE 754-2008: 5.3.1
    return op0;

  } else if (TFPU_F32_IsQNan(op0)) {
    // IEEE 754-2008: 5.3.1
    return op1;

  } else if (std::isinf(op0)) {
    // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
    // Also min(a,a) = a
    if (isNeg0) {
      return op0;
    } else {
      return op1;
    }

  } else if (std::isinf(op1)) {
    // IEEE 754-2008: 6.1 (-inf < {every finite number} < +inf)
    if (isNeg1) {
      return op1;
    } else {
      return op0;
    }

  } else if (std::abs(op0) == std::abs(op1)) {
    if (isNeg1) {
      // if a is positive, min(-a,-a) = -a; min(a,-a) = -a (including a == 0.0)
      return op1;
    } else {
      return op0;
    }

  } else {
    return (op0 < op1 ? op0 : op1);
  }
}
```

4.1.3.51 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*.

```
float TFPU_Max(float op0, float op1)
{
  bool isNeg0 = std::signbit(op0);
  bool isNeg1 = std::signbit(op1);

  if (TFPU_F32_IsSNan(op0)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op0);

  } else if (TFPU_F32_IsSNan(op1)) {
    // IEEE 754-2008: 6.2
    return TFPU_F32_QuietenNan(op1);

  } 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: }
```

4.1.3.52 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: }
```

4.1.3.53 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: }
```

4.1.3.54 TFPU_GetNanooMode

```
1: HalfSaturationMode_t TFPU_GetNanooMode(bool fpCtlNanoo)
2: {
3:   if (fpCtlNanoo) {
4:     return TFPU_HSATURATE_NAN;
5:   } else {
6:     return TFPU_HSATURATE_MAX;
7:   }
8: }
```

4.1.3.55 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: }
```

4.1.3.56 TFPU_ApplyF16StochasticRound

```
1: void TFPU_ApplyF16StochasticRound(std::array<uint64_t, 2> &randm, std::vector<float> &values, TileFP16Fmt_t fmt)
2: {
3:   TFPU_ApplyStochasticRoundHalf(randm, values);
4:   fmt = fmt;
5: }
```

4.1.4 Exceptions

4.1.4.1 Imprecise Exceptions

The only imprecise exception raised by *IPU21* is overflow (*TFPEXCPT_OFLO*), detected during the execution of *f32div*.

4.1.4.2 Super-imprecise Exceptions

The only Super-imprecise exception raised by *IPU21* 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 Parameters

Table 4.7: Exceptions parameters

Parameter name	Value	Description
TEXCPT_ENUM_BITWIDTH	4	The total number of bits required to encode all supported exceptions

4.1.4.5 TileException

Tile exception identifiers.

Table 4.8: Enumeration: TileException

Identifier	Value	Description
TEXCPT_MEMERR	14	Memory error exception
TEXCPT_EXERR	13	Exchange error exception
TEXCPT_INVALID_INSTR	12	Invalid instruction exception
TEXCPT_DBRK	11	Data BREAK exception. This is a debug exception. Run mode returns to TRUNM_EXECUTING, with the Instruction-phase set to TPHASE_FETCH once the exception is cleared.
TEXCPT_INVALID_PC	10	Invalid $PC exception
TEXCPT_INVALID_OP	9	Invalid operand exception
TEXCPT_INVALID_ADDR	8	Load/store invalid *Tile Memory* address exception
TEXCPT_EXCONF	7	Invalid Exchange configuration exception
TEXCPT_CONFLICT	6	*Tile Memory* bank or port conflict exception
TEXCPT_FP	5	Malign floating-point exception
TEXCPT_BOS	4	BREAK-On-Sync exception. This is a debug exception. Run mode returns to TRUNM_EXECUTING, with the Instruction-phase set to TPHASE_FETCH once the exception cleared.

Continued on next page

Table 4.8 – continued from previous page

Identifier	Value	Description
TEXCPT_PBRK1	3	Patched BREAKPOINT/System call ID 1. This is a debug exception. Run mode returns to TRUNM_EXECUTING, with the Instruction-phase set to TPHASE_FETCH once exception is cleared.
TEXCPT_PBRK0	2	Patched BREAKPOINT/System call ID 0. This is a debug exception. Run mode returns to TRUNM_EXECUTING, with the Instruction-phase set to TPHASE_FETCH once exception is cleared.
TEXCPT_RBRK	1	Retirement BREAK exception. This is a debug exception. Run mode returns to TRUNM_EXECUTING, with the Instruction-phase set to TPHASE_FETCH once the exception is cleared.
TEXCPT_NONE	0	No exception

4.1.5 Contexts

4.1.5.1 Parameters

Table 4.9: Contexts parameters

Parameter name	Value	Description
CTXT_WORKERS	6	The total number of Worker contexts supported per Tile
CTXT_TOTAL_BITWIDTH	3	ceil(log2(CTXT_TOTAL))

4.1.5.2 TileCtxtStatus

Context status brief.

Table 4.10: Enumeration: TileCtxtStatus

Identifier	Value	Description
TCTXT_STATUS_INACTIVE	0	Worker context *run mode* is *Inactive*
TCTXT_STATUS_ACTIVE	1	Context *run mode* is not any of those covered by the other TCTXT_STATUS enumerations.
TCTXT_STATUS_EXCEPTED_DBG	2	Context *run mode* is TRUNM_EXCEPTED (having raised a debug exception)
TCTXT_STATUS_EXCEPTED_NDBG	3	Context *run mode* is TRUNM_EXCEPTED (having raised a non-debug exception)

4.1.6 Memory

4.1.6.1 Memory Regions

IPU21's implemented memory space is split into two *Memory Regions* of unequal size.

Table 4.11: *IPU21* 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 *IPU21* 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.

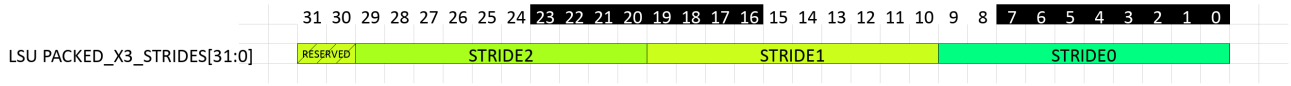


Fig. 4.1: **LSU_PACKED_X3_STRIDES** data-type format

Table 4.12: **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 4.13: Memory parameters

Parameter name	Value	Unit	Description
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 *Tile Memory* region 0.
TMEM_REGION1_SIZE	416/0x1a0	KiBytes	The logical address space of *Tile Memory* 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 *Tile Memory* region 0.
TMEM_BASE_ADDR	311296/0x4c000	bytes	The logical base address of *Tile Memory* (in bytes).
TMEM_BASE_ADDR_WORD	77824/0x13000	words	The logical base address of *Tile Memory*, in 32-bit words.
TMEM_FULL_ADDRESS_MASK	0x1fffff	Bits	Architectural (and therefore constant across all implementations) address space mask.
TMEM_BYTE_ADDRESS_WIDTH	20/0x14	Bits	Implementation byte address width in bits.
TMEM_WORD_ADDRESS_WIDTH	18/0x12	Bits	Word address width in bits.
TMEM_DWORD_ADDRESS_WIDTH	17/0x11	Bits	64-bit (double *word*) address width in bits.
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:
10:   } else {
11:     return   ((((address - regionBase) >> (TMEM_ELEM_OFFSET_SIZE + 1)) << 1)
12:            | ((address >> 3) & 1)) + (TMem_RegionSizeKBytes(0) / TMEM_ELEMSIZE);
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 Parameters

Table 4.14: Registers parameters

Parameter name	Value	Description
TREG_REPEAT_COUNT_WIDTH	16/0x10	Repeat down-counter width.

4.1.7.2 TileRFAccessSize

The width of a read or write access to a Tile register-file.

Table 4.15: 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

4.1.7.3 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:     if (size > TRF_ACCESS_SIZE_QUAD) {
32:       indices.push_back(baseIndex | 4);
33:       indices.push_back(baseIndex | 5);
34:       indices.push_back(baseIndex | 6);
35:       indices.push_back(baseIndex | 7);
36:     }
37:   }
38:  }
39: }
```

4.1.7.4 TReg_IsValidCCCS

```
bool TReg_IsValidCCCS(uint32_t index)
{
  if (index < (TREG_CCCS_WEIGHT_GROUP_SIZE * TREG_CCCS_NUM_WEIGHT_REG_GROUP)) {
    return true;
  } else {
    return false;
  }
}
```

4.1.7.5 TReg_WriteException

```
TileException_t TReg_WriteException(uint32_t index, bool supervisor)
{

  if (!supervisor) {
    switch (index) {
      case 0: /* $PC */
      case 1: /* $WSR */
      case 2: /* $VERTEX_BASE */
      case 3: /* $WORKER_BASE */
      case 4: /* $REPEAT_COUNT */
      case 5: /* $REPEAT_FIRST */
      case 6: /* $REPEAT_END */
      case 96: /* $COUNT_L */
      case 97: /* $COUNT_U */
      case 112: /* $DBG_DATA */
      case 113: /* $DBG_BRK_ID */
      case 256: /* $FP_STS */
      case 257: /* $FP_CLR */
      case 258: /* $FP_CTL */
      case 259: /* $PRNG_0_0 */
      case 260: /* $PRNG_0_1 */
      case 261: /* $PRNG_1_0 */
      case 262: /* $PRNG_1_1 */
      case 263: /* $PRNG_SEED */
      case 264: /* $TAS */
      case 265: /* $FP_NFMT */
      case 266: /* $FP_SCL */
        return TEXCPT_NONE;
    }
  }

  // Default for unknown registers is INVALID_OP exception
  return TEXCPT_INVALID_OP;
}
```

4.1.7.6 TReg_IsValidCSR

```
bool TReg_IsValidCSR(uint32_t index, bool supervisor)
{

  if (!supervisor) {
    switch (index) {
      case 0: /* $PC */
      case 1: /* $WSR */
      case 2: /* $VERTEX_BASE */
      case 3: /* $WORKER_BASE */
      case 4: /* $REPEAT_COUNT */
      case 5: /* $REPEAT_FIRST */
      case 6: /* $REPEAT_END */
      case 96: /* $COUNT_L */
      case 97: /* $COUNT_U */
      case 112: /* $DBG_DATA */
      case 113: /* $DBG_BRK_ID */
      case 256: /* $FP_STS */
      case 257: /* $FP_CLR */
      case 258: /* $FP_CTL */
      case 259: /* $PRNG_0_0 */
      case 260: /* $PRNG_0_1 */
      case 261: /* $PRNG_1_0 */
      case 262: /* $PRNG_1_1 */
      case 263: /* $PRNG_SEED */
```

```
25:       case 264: /* $TAS */
26:       case 265: /* $FP_NFMT */
27:       case 266: /* $FP_SCL */
28:         return true;
29:     }
30:   }
31:   return false;
32: }
```

4.1.7.7 MRF

4.1.7.7.1 Parameters

Table 4.16: MRF parameters

Parameter name	Value	Description
MRF_GP_REGISTERS	12/0xc	The total number of populated MRF general-purpose registers, per context

4.1.7.7.2 TReg_MRFResetValue

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_MRFResetValue(uint32_t &value)
2: {
3:   // Every populated MRF register reset to zero
4:   value = 0;
5:   return true;
6: }
```

4.1.7.8 ARF

4.1.7.8.1 Parameters

Table 4.17: ARF parameters

Parameter name	Value	Description
ARF_GP_REGISTERS	8	The total number of populated ARF general-purpose registers, per context

4.1.7.8.2 TReg_ARFResetValue

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_ARFResetValue(uint32_t &value)
2: {
3:   // Every populated ARF register reset to zero
4:   value = 0;
5:   return true;
6: }
```

BIBLIOGRAPHY

[IEEE754] IEEE Std 754™-2008 http://ieeexplore.ieee.org/document/4610935/

INDEX

Trademarks & Copyright