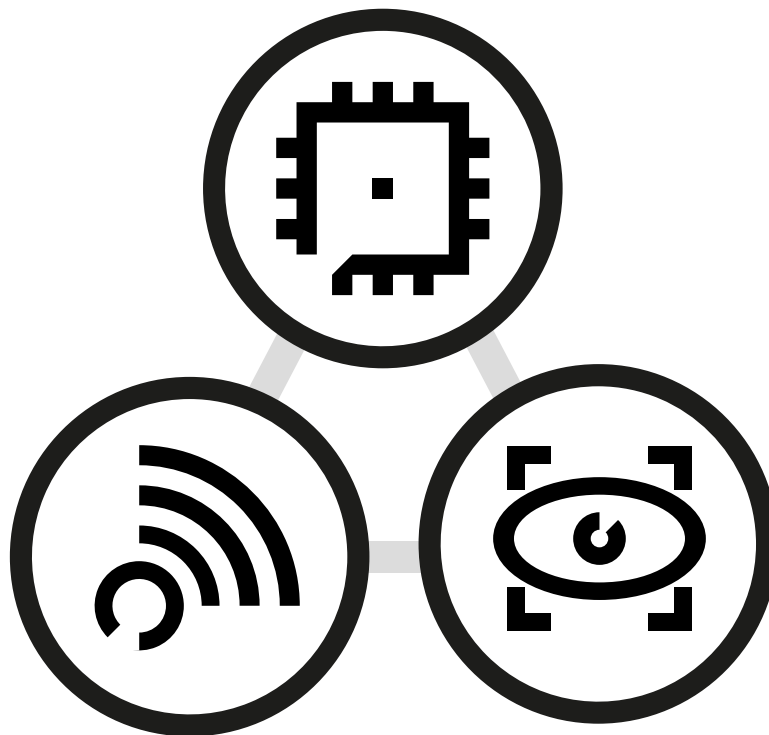


# PowerVR Instruction Set Reference

Revision: 1.0  
11/02/2020  
Public



Public. This publication contains proprietary information which is subject to change without notice and is supplied 'as is', without any warranty of any kind. Redistribution of this document is permitted with acknowledgement of the source.

Published: 11/02/2020-08:48

# Contents

<b>1. General Architecture Information.....</b>	<b>6</b>
<b>2. PowerVR USC Series 6 Core Overview.....</b>	<b>8</b>
USC.....	8
USC ALU.....	9
Vertex Processing Pipeline.....	10
Pixel Processing Pipeline.....	11
<b>3. Structure of ISR assembly code.....</b>	<b>13</b>
Main Instruction Group.....	13
Bitwise Instruction Group.....	15
<b>4. Types of Registers.....</b>	<b>16</b>
<b>5. Instruction Modifiers.....</b>	<b>20</b>
<b>6. Floating Point Instructions.....</b>	<b>23</b>
FMAD.....	23
FADD.....	23
FMUL.....	23
FRCP.....	24
FRSQ.....	24
FSQRT.....	25
FLOG.....	25
FEXP.....	26
GCMP.....	26
GEXP.....	26
F16SOP.....	27
SOPMOV.....	28
F16SOP.MAD.....	28
F16SOP.U8.....	29
SOPMOV.U8.....	29
F16SOP.U8MAD.....	29
SOPU8MADMOV.....	30
MBYP.....	30
FDSX.....	30
FDSY.....	31
FDSXF.....	31
FDSYF.....	31
CONVERTTOF64.....	32
CONVERTFROMF64.....	32
FSINC.....	32
FARCTANC.....	33

FRED.....	33
GTA.....	34
<b>7. Data Movement Instructions.....</b>	<b>35</b>
MOV.....	35
MOVC.....	35
PCK.....	35
UNPCK.....	36
<b>8. Integer Instructions.....</b>	<b>37</b>
UADD8.....	37
UMUL8.....	37
UMAD8.....	38
IADD8.....	38
IMUL8.....	38
IMAD8.....	39
UADD16.....	39
UMUL16.....	40
UMAD16.....	40
IADD16.....	41
IMUL16.....	41
IMAD16.....	42
ADD64.....	42
UADD6432.....	42
SADD6432.....	43
UMADD32.....	43
SMADD32.....	43
UMADD64.....	44
SMADD64.....	44
<b>9. Test Instructions.....</b>	<b>45</b>
TSTZ.....	45
TSTGZ.....	45
TSTGEZ.....	46
TSTC.....	46
TSTE.....	47
TSTG.....	47
TSTGE.....	48
TSTNE.....	48
TSTL.....	49
TSTLE.....	49
TSTMIN.....	50
TSTMAX.....	50
<b>10. Bitwise Instructions.....</b>	<b>51</b>
AND.....	51
OR.....	51
XOR.....	52
NAND.....	52

NOR.....	52
XNOR.....	53
SHFL.....	53
REV.....	53
LSL.....	53
CPS.....	54
SHR.....	54
ASR.....	54
ROL.....	55
TZ.....	55
TNZ.....	56
BYP.....	56
MSK.....	56
CBS.....	57
FTB.....	57
FTB_SHI.....	57
FTB_MSB.....	58
<b>11. Backend Instructions.....</b>	<b>59</b>
<b>12. Flow Control Instructions.....</b>	<b>61</b>
<b>13. Conditional Instructions.....</b>	<b>62</b>
CNDST.....	62
CNDEF.....	62
CNDSM.....	62
CNDLT.....	63
CNDEND.....	63
CNDSETL.....	64
CNDLPC.....	64
<b>14. Data Access Instructions.....</b>	<b>66</b>
WDF.....	66
ITRSM (ITRSM1D ITRSM2D ITRSM3D).....	66
SBO.....	66
DITR.....	67
<b>15. Appendix.....</b>	<b>68</b>
<b>16. Contact Details.....</b>	<b>71</b>

# 1. General Architecture Information

---

The Unified Shading Cluster (USC) assembly code described in this Instruction Set Reference (ISR) drives the USC component(s) of the PowerVR architecture.

The Unified Shading Cluster (USC) assembly code described in this Instruction Set Reference (ISR) drives the USC component(s) of the PowerVR architecture.

The information in this document enables developers to write shaders that use the hardware more efficiently.

Even though most of the information in this document applies to the PowerVR Series 6XT architecture, parts that may not be applicable on all PowerVR architectures have been labelled as **optional**.

Refer to the main PowerVR ISR document (NDA required) for precise information regarding feature availability.

Below is the hierarchical description of a USC with the important parts in bold.

A **USC** comprises:

- USC Common Store (CS or USCCS)
- **USC Pipeline Datapaths (USCPDs)**
- Iterators
- DMA Output
- F64 ALU-Pipeline datapath (optional)

Each **USCPD** comprises:

- Unified Store (US)
- Bypass FIFO
- One **ALU Pipeline**

The **Arithmetic Logic Unit (ALU) Pipeline** in each USCPD comprises:

- ALU Instances
- Sideband/Control Bypass Pipeline
- Texture Address Unit (TAU) (optional)

Each **ALU Instance** contains a set of ALUs (see 'ALU Partitioning') and comprises:

- ALU Source unit – selects sources, and swizzles
- ALU Phase 0 – arithmetic operations for phase 0
- ALU Phase 1 – arithmetic operations for phase 1
- ALU Phase 2 – arithmetic operations for phase 2
- ALU Move – final multiplexing of results
- F16 ALU-phases (optional)

The pipeline operates in Single Instruction Multiple Data (SIMD) mode across multiple parallel data instances that are processed at a rate of one scalar component per clock, in other words, it is not a vector pipeline.

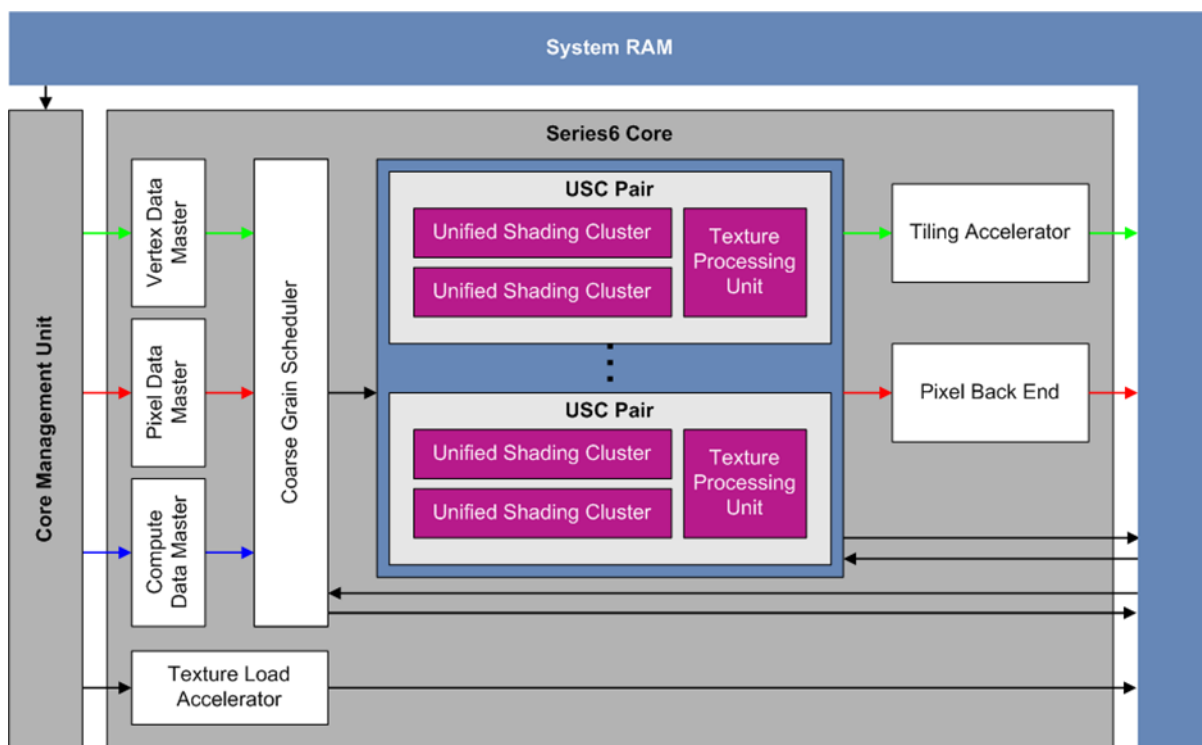
## 2. PowerVR USC Series 6 Core Overview

A high-level overview of the architecture of the PowerVR Series 6 GPU core

The image below shows a high level overview of the PowerVR Series 6 Graphics Processing Unit (GPU) core.

In this architecture the vertex and fragment stages share the same Unified Shading Cluster (USC) cores. These cores can either output their result to the Tiling Accelerator (TA) or the Pixel Back End (PBE). The USC cores are fed by 3 types of data masters and a scheduler. Each pair of USCs share a Texture Processing Unit (TPU).

The Texture Load Accelerator (TLA) handles converting texture data into optimal format and the acceleration of 2D surface operations such as blits.



### USC

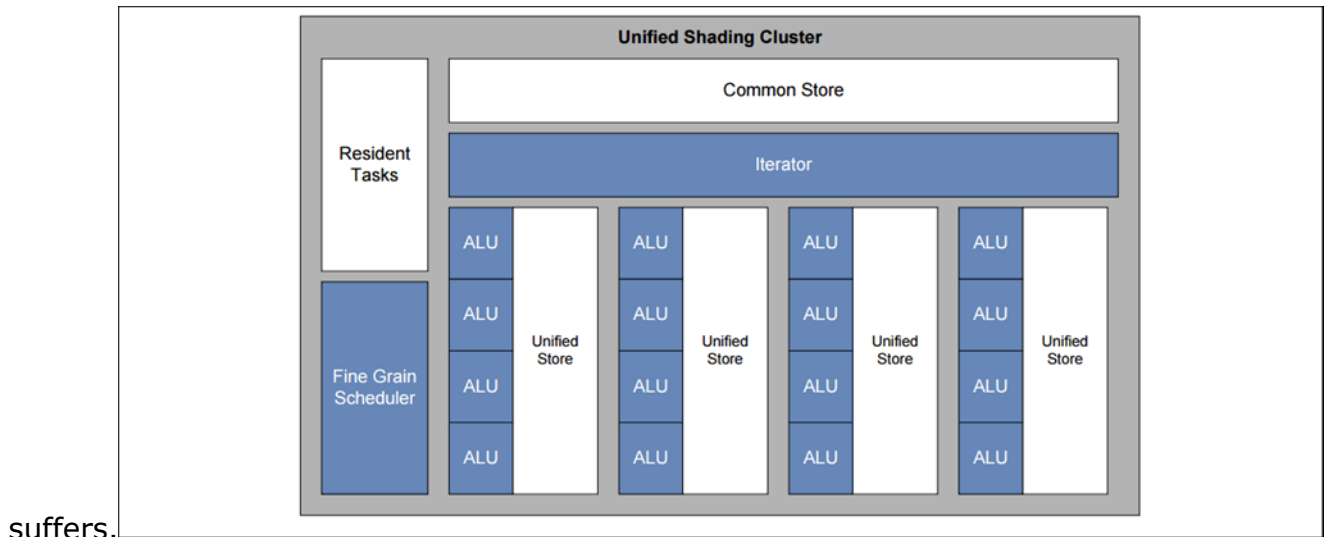
The layout of a single unified shading cluster

The image below shows the layout of a single USC. A USC uses the Common Store (USCCS) for fetching uniform data and the iterator for getting interpolated varying data. It is also fed by the fine grain scheduler that takes resident tasks to be executed.

The USC contains numerous ALU instances each working on a thread. These could be a vertex task, a pixel task, and so on. There are 16 instances in a group and the instances in a group execute the same instructions.



Each group of 1024 ALU instances share a Unified Store (US) that they can use to store data temporarily. Each ALU instance has space for about 24 floats in a shared US. Although it is possible to have more than 24 floats in the US for each ALU instance, it is not advised as in that case occupancy



suffers.

## USC ALU

The layout of a single ALU pipeline is shown below. Most of the USC instructions run on this pipeline, and it is best to utilise all the stages in a given path in this pipeline.

For example:

**Optional:** It is possible to execute...

- two F16 (Sum of Products) SOP instructions
- + the F32 <-> F16 conversions
- + the `mov/output/pack` instruction... all in one cycle

**Optional:** Also, on some of the PowerVR hardware there is the possibility of executing 4 SOPMAD instructions in one cycle.

Alternatively, it is possible to execute...

- an FP32 (Multiply-Add) MAD and an FP32/INT32 MAD/UNPACK instruction could be executed,
- plus a test (conditional) instruction
- plus the `mov/output/pack` instruction... in one cycle

This allows for performing a conditional `mov`.

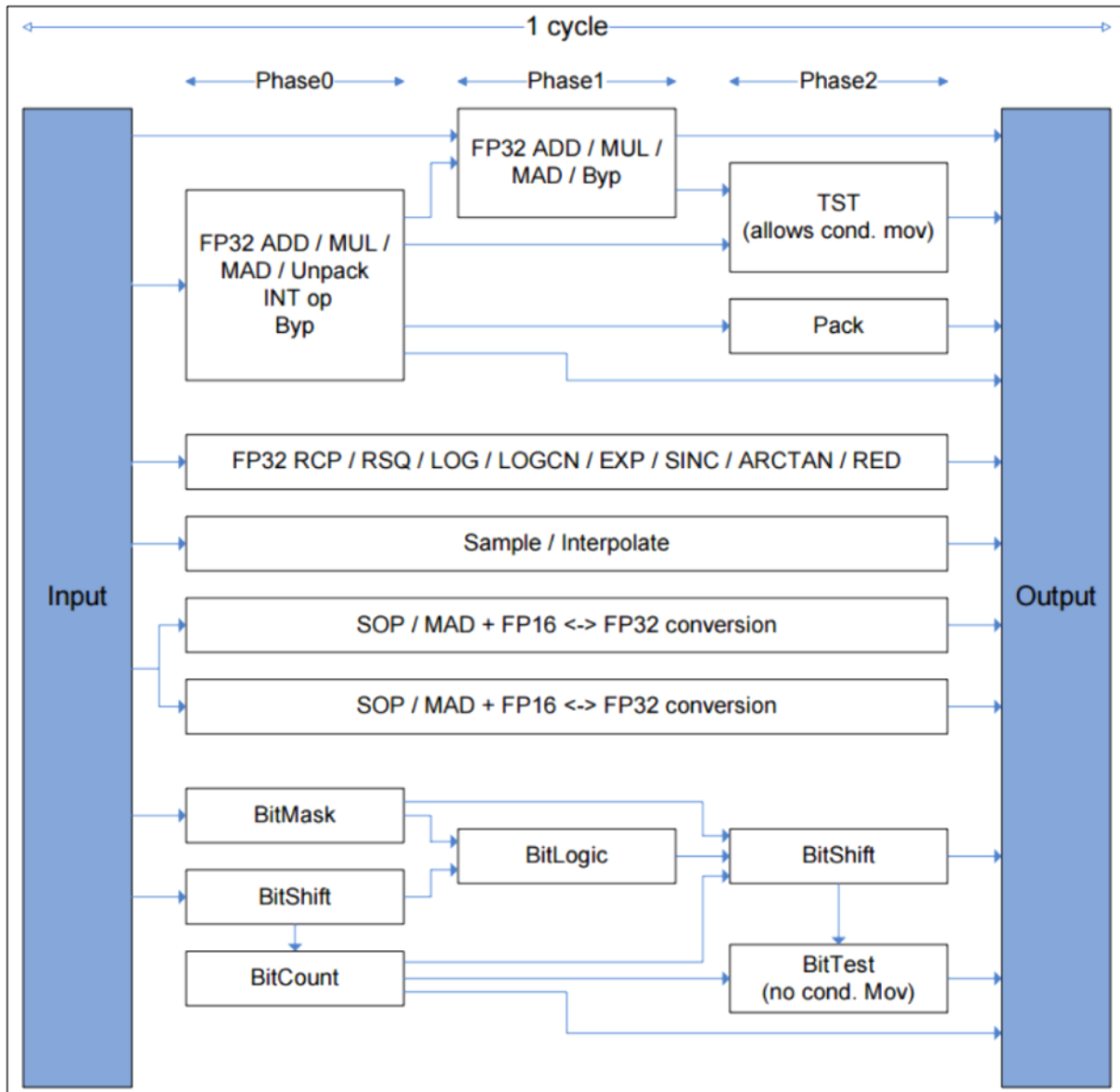
If there is bitwise work to be done, it is possible to issue...

- a bitwise MASK or SHIFT/COUNT,
- a bitwise logical operation,
- a bitwise shift,
- a test

- the mov/output/pack instructions... all in one cycle

It is also possible to execute a single complex operation (ie. `rcp`) and a `mov/output/pack` instruction in one cycle.

Lastly, an interpolate/sample instruction plus the usual `mov/output/pack` instruction can be executed in one cycle.

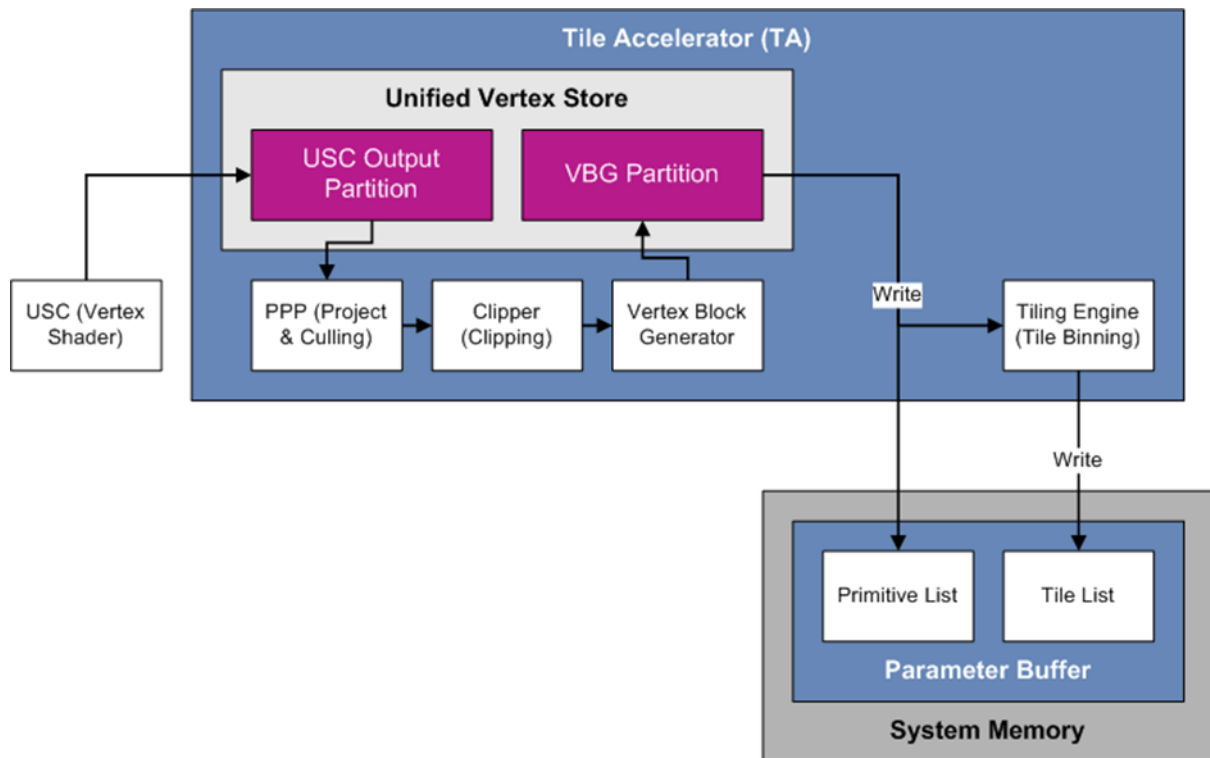


## Vertex Processing Pipeline

The vertex processing pipeline produces a list of primitives and tile list

The image below shows the various stages a vertex goes through after being processed by the vertex shader.

The end result of this process is a list of primitives that are projected and clipped. In addition, a list containing which primitive belongs to a given tile is also produced.

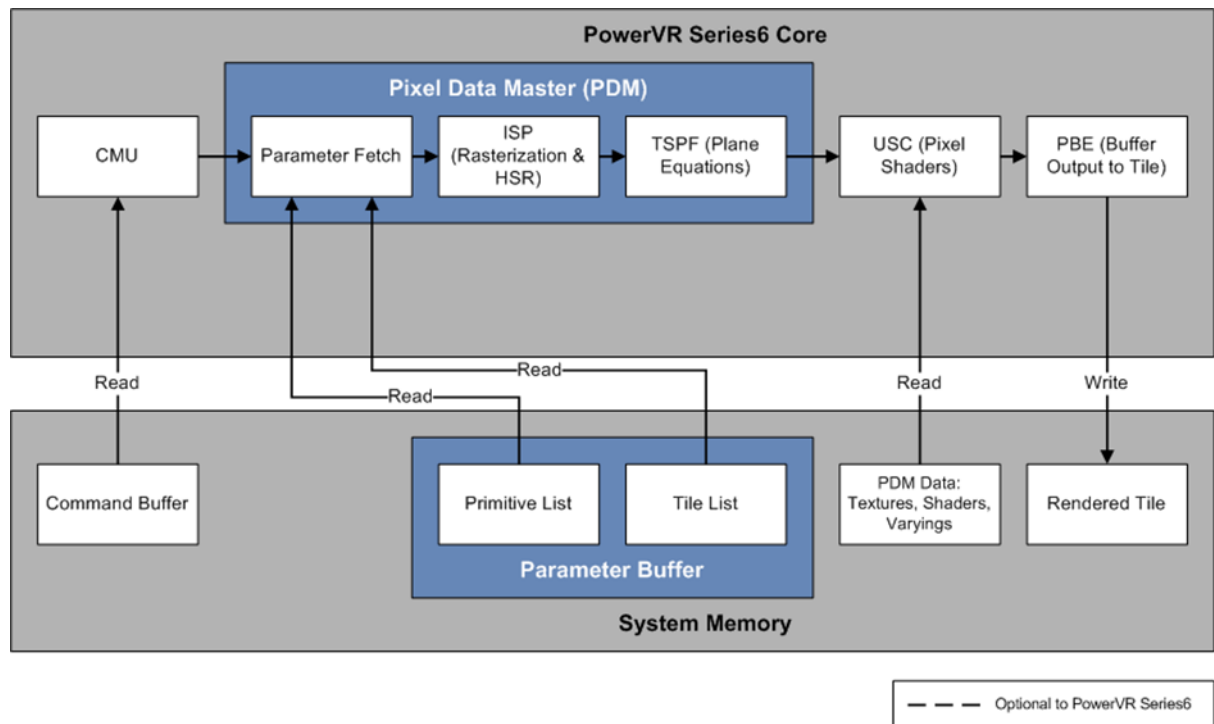


## Pixel Processing Pipeline

An overview of the pixel processing pipeline in the PowerVR Series6 core

After the tiling process is complete, tiles are processed at the pixel stage. Primitives are rasterized, hidden surfaces are removed and plane equations are calculated for interpolating vertex data.

Then, pixel shaders are executed and their result is written to the pixel buffer.



### 3. Structure of ISR assembly code

To aid programming, and support instruction co-issue, program instruction groups are used.

Instruction co-issue is where multiple instructions are issued, in a single clock cycle, to run in the multiple phases of the Unified Shading Cluster (USC) Pipeline Datapath (USCPD).

The Arithmetic Logic Unit (ALU) pipeline allows up to 6 instructions (1 per phase + backend) to be issued within a clock. These instructions are presented to the assembler as a "group" of instructions.

The general layout of an instruction group can be seen below:

```
[n] : [if (cond)] # n is group number (if is optional)
[Op 0] # First op
[Op 1] # Second op (optional)
..
[Op N] # Nth operation (optional)
```

**Note:** Some of the commands are removed in the interest of clarity. However, the main instructions are still there, so it is sufficient to optimise code based on the disassembly output.

#### Main Instruction Group

The Main ALU performs all floating point, integer, and packing/unpacking operations

The Main ALU performs all floating point, integer, and packing/unpacking operations. The operations are split into three phases that are fully exposed as separate entities where they are not being used for other functions.

The Main ALU may not be used at the same time as the Bitwise ALU.

#### Example

```
0 : if (p0)
mul ft0, s0, s1 # phase 0 instruction
mad ft1, s3, s4, s5 # ft1 drives w1 by default, second to last phase present
mul ft2, is0, ft0 # ft2 drives w0 by default, last phase present
tstz.f32 ftt, p0, is1 # test instruction always drives feedthrough ftt

# p0 write optional
movc w0,w1, ftt, ft0, ft1 # ft1 or ft0 output to w0 (overrides default)
uvsw w0 # backend instruction
```

#### Opcodes

Phase 0, phase 1 and backend can each take one instruction. Phase 2 can take PACK, TEST and/or MOV instructions.

Instructions should appear in the code in phase order (zero to two) and if there is a TEST instruction it should appear before the MOV instruction.

Instructions do not need to be present for all phases, but there are some restrictions on which combinations of phases can be present. Phase 1 can only be present if

phase 0 is also present. Backend instructions can be present on their own or in combination with phase 0 and phase 2.

#### Source Arguments

There are two groups of ALU inputs:

- S0, S1, S2
- S3, S4, S5

0, 1, 2 or 3 inputs may be used from each instruction group, but the lowest numbered inputs within each group must be used first.

Source arguments denote possible input registers in the reference section.

#### Internal Source

Internal feedthrough sources (FT0, FT1 and FT2) are generated by phase 0, 1 and 2 instructions respectively. These are shown as the destination (first argument) of the phase instructions.

Internal sources for phase 2 may need to be declared if phase two instructions are present. Phase 2 instructions can only choose from two pre-selected external sources in addition to FT0 and FT1 and these are named IS0 and IS1.

#### Destination Arguments

ALU outputs are:

- W0
- W1

Destination arguments denote possible output registers in the reference section.

#### Complex Instructions

Complex and trigonometric instructions use the entire Main ALU and have fixed sources and destinations.

**Optional:** However, when issued in Phase 1, Complex and Trigonometric instruction types can leave resources free in Phase 0 of the Main ALU to be coissued. They can also be coissued with F16SOP operations. FRED instructions can never be coissued and 32/64 bit integer instructions cannot be used in Phase 0.

#### Texture Address Unit

The texture address unit (TAU) is used to calculate texel memory addresses for a given set of texture coordinates. The texture address unit occupies all phases of the Main ALU, and is used via the GTA (Generate Texel Address) instruction. GTA has fixed sources and destinations.

The TAU processes four instances in parallel which allows common source data and calculations to be shared. For example, each instance reads only 32 bits of the 128-bit texture state.

## F16 Sum-of-Products Unit

The F16 Sum-of-Products (SOP) unit is an F16 precision vector ALU, which may be used for operations such as alpha blending. Any F32 inputs are converted to F16 prior to performing calculations. The F16 SOP unit uses the entire Main ALU.

## Bitwise Instruction Group

The Bitwise ALU performs all bitwise and logical operations

The Bitwise ALU performs all bitwise and logical operations. The Bitwise ALU may not be used at the same time as the Main ALU.

### Example

```
0 : lsl ft2, s2, s1 # Shift1 instruction
  ftb ft3, ft2 # Count instruction, ft3 implicitly drives w1
  and ft4, ft2, s3 # Logical instruction
  asr ft5, ft4, s1, ft3 # Shift2 instruction, ft5 implicitly drives w0
  tz p0, ft5 # Test instruction
```

### Opcodes

A bitwise instruction group consists of up to six instructions (grouped into three phases) which are associated with the following six functional units:

Phase 0:

- bitmask
- bit shift #1
- bit count

Phase 1:

- logical instruction

Phase 2:

- bit shift #2
- bit test

Different instructions are available for each functional unit. Not all six instructions have to be present, but those that are present must be arranged in the order shown above. The source declarations should be placed before the instructions and the destination declarations should be placed after the instructions.

In the code example above no instruction has been provided for the bitmask function in phase 0. So this logical unit is unused and the opcode for it will be set to a default value by the compiler.

In general, the compiler will fill in default encoding values for unused instructions in any phase that is used. The programmer need not worry about phases and can just assign opcodes to functional blocks as required.

**Note:** No bitwise instructions use multiple phases and there are no bitwise backend instructions.

# 4. Types of Registers

A list of the different types of registers available within PowerVR GPUs

## Temporary

**Code:** Rn[IDXi, D]

**Access:** Read/Write

**Max Available:** 248 (n)

**Description:** *Temporary Registers* are used for general purpose calculation and are uninitialised.

*Temporary Registers* are allocated from the *Unified Store* and may be indexed

(D is dimension can be [1, 2, 4, 8, 16], default value is 1 if omitted)

## Vertex Input

**Code:** Vin[IDXi, D]

**Access:** Read/Write

**Max Available:** 248 (n)

**Description:** *Vertex Input Registers* behave similarly to *Temporary Registers* except they contain pre-initialised inputs to each instance.

*Vertex Input Registers* are allocated from the *Unified Store* and may be indexed.

Table 1: Example:

<pre>void main() {     gl_Position = inVertex; }</pre>	<pre>0      : mov ft0, vi3       : mov ft0.e0.e1.e2.e3, ft0       : uvsw.write ft0, 3;</pre>
--	--

## Coefficient Registers (Normal/Alternate set)

**Code:** CFn[IDXi, D], CFAn[IDXi, D]

**Access:** Read/Write

**Max Available:** Architecture Dependent (n)

**Description:** *Coefficient Registers* (and the alternate set) contain pre-initialised inputs shared between multiple instances of the same thread. They are normally read-only but may be written to if multiple instances ensure they do not clash by writing the same registers – the USC does not perform hazard checking for this.

*Coefficient Registers* are allocated from the *Common Store* and may be indexed.

They are used for interpolating vertex shader outputs between vertices.



Table 2: Example:

<pre>void main() {     fragColor = vec4(textureCoordinate, 0.0, 1.); }</pre>	<pre>0      : fitr.pixel r0, drc0, cf4, cf0, 2;</pre>
--	---

**Shared Registers****Code:** SHn[IDX<sub>i</sub>, D]**Access:** Read/Write**Max Available:** 4096 (n)

**Description:** *Shared Registers* contain pre-initialised inputs shared by an entire thread. They are normally read-only but may be written to if multiple instances ensure they do not clash by writing the same registers – the USC does not perform hazard checking for this.

*Shared Registers* are allocated from the *Common Store* and may be indexed.

Table 3: Example:

<pre>uniform vec4 t;  void main() {     fragColor = t; }</pre>	<pre>0      : mov ft0, sh0         mov r0.e0.e1.e2.e3, ft0         mov r1, sh1;</pre>
--	---

**Index Registers****Code:** IDX<sub>i</sub>**Access:** Read/Write**Max Available:** 2 (i)

**Description:** The *Index Registers* are used to index other register banks and are uninitialised.

**Pixel Output Registers****Code:** On[IDX<sub>i</sub>, D]**Access:** Read/Write**Max Available:** Architecture Dependent (n)

**Description:** *Pixel Output Registers* are used by each instance of a pixel shader to output data to the PBE module.

If there are tiles per USC *Pixel Output Registers* are allocated from the *Partition Store* and may be indexed.

If this is not the case, *Pixel Output Registers* are allocated from the *Common Store* and may be indexed.

## Special Constant

**Code:** SC<sub>n</sub> / SR<sub>n</sub>

**Access:** SC: read-only. SR: some are Read/Write

**Max Available:** 240 (n)

**Description:** *Special Constants* contain fixed values that may be useful to an instance. These are always present for each thread (they do not have to be allocated from either store) and are described in the tables below.

The *Special Constants* logical memory space, SC, contains "Special Constants". These are values which are truly constant for a particular instance of a program.

To help with using shorter encodings the special constants are interleaved, so the most commonly used are available with 6-bit offsets. They are interleaved 32 values each.

8 offset registers are provided, registers 36-43.

**Note:** Although offsets 0 to 7 are provided for *Internal Registers*, the actual number of *Internal Registers* and offsets depends on the configuration and will not exceed 8.

**Note:** Although offsets 0 to 7 are provided for *Slot Registers*, the actual number of *Slot Registers* and offsets depends on the configuration and will not exceed 8. The total number of *Slot Registers* and *Internal Registers* will not exceed 8.

You can find a list of these registers in the [appendix](#).

**Table 4: Example:**

<pre>void main() {     fragColor = vec4(1.0); }</pre>	<pre>0      : mov ft0, c64 #c64 = 1.0         mov r0.e0.e1.e2.e3, ft0         mov r1, c64;</pre>
---	--

## Vertex Output Registers

**Code:** V<sub>on</sub>

**Access:** Write-only

**Max Available:** 256 (n)

**Description:** *Vertex Output Registers* are used by each instance to output data to the UVS module. They exist outside of the USC and as such are write-only using the UVSW instruction.

*Vertex Output Registers* may be indexed. They may only be written using the UVSW instruction.

## Dynamic Constant Registers (optional)

**Code:** DC<sub>n</sub> [IDX<sub>i</sub>, D]

**Access:** Read-only

**Max Available:** 16384 (n)

**Description:** *Dynamic Constant Registers* are large arrays of constants that exist in main memory to which the USC manages windows of 16 read only registers per instance.

*Dynamic Constant Registers* must be indexed.

#### Internal Registers (optional)

**Code:** `In`

**Access:** Read/Write

**Max Available:** Architecture Dependent (n)

**Description:** Internal Registers used for general purpose calculation and are uninitialised.

They are similar to *Temporary Registers* but are not allocated from the *Unified Store*, instead there is a dedicated set per instance. As such they cannot be used for DMA or FITR operations.

Internal Registers may not be indexed.

The number of *Internal Registers* available is dependent on the configuration of Rogue and may range from not being available at all to 8 registers.

#### Slot Registers (optional)

**Code:** `SIn`

**Access:** Read/Write

**Max Available:** Architecture Dependent (n)

**Description:** Slot registers are used for general purpose calculations that are not dependent on data, such as loop counters. They are similar to a shared registers allocation that is not initialised in that only one instance will write into this register per instruction, however the *Slot Registers* are not visible to other threads.

The number of *Slot Registers* available is dependent on the configuration of Rogue and may range from not being available at all to 8 registers.

## 5. Instruction Modifiers

A list of instruction modifiers which can be on PowerVR GPUs

Modifier Name	Description		
.ABS	Take the absolute value  <b>Table 5: Example:</b>  <table> <tr> <td> <pre>void main() {     fragmentColor = abs(a); }</pre> </td><td> <pre>0 : mov ft0, sh0.abs    mov ft1, sh1.abs    mov r0, ft0;    mov r1, ft1;</pre> </td></tr> </table>	<pre>void main() {     fragmentColor = abs(a); }</pre>	<pre>0 : mov ft0, sh0.abs    mov ft1, sh1.abs    mov r0, ft0;    mov r1, ft1;</pre>
<pre>void main() {     fragmentColor = abs(a); }</pre>	<pre>0 : mov ft0, sh0.abs    mov ft1, sh1.abs    mov r0, ft0;    mov r1, ft1;</pre>		
.ARRAY	Enable texture arrays  <b>Table 6: Example:</b>  <table> <tr> <td> <pre>uniform highp sampler2DArray sampler;  void main() {     fragmentColor =     texture(sampler,     vec3(textureCoordinate, 0.0)); }</pre> </td><td> <pre>3 : smp2d.fcnorm.array drc0,    sh4, r1, sh0, _, r0, 4;</pre> </td></tr> </table>	<pre>uniform highp sampler2DArray sampler;  void main() {     fragmentColor =     texture(sampler,     vec3(textureCoordinate, 0.0)); }</pre>	<pre>3 : smp2d.fcnorm.array drc0,    sh4, r1, sh0, _, r0, 4;</pre>
<pre>uniform highp sampler2DArray sampler;  void main() {     fragmentColor =     texture(sampler,     vec3(textureCoordinate, 0.0)); }</pre>	<pre>3 : smp2d.fcnorm.array drc0,    sh4, r1, sh0, _, r0, 4;</pre>		
.CLAMP	Clamp is applied after absolute, but before negation. Clamps value to range [+0,1]		
.COMPARISON	Enable comparison filtering in TPU		
.DIRECT	Direct DMA instruction, bypassing the main ALU pipeline		
.E0	Element selector for operations that operate on types narrower than 32-bit. This will normally be preceded by the data type (f16 etc.). For source, selects the part of the argument that forms an element For destination, broadcasts the result to these elements of destination. Reads from: bits 0-7		
.E1	Reads from: bits 8-15		
.E2	Reads from: bits 16-23		
.E3	Reads from: bits 24-31		
.F16	Return packed F16 data		
.F32	Return packed F32 data		
.FCNORM	Fixed point texture data, converted to floating point when returned to USC  <b>Table 7: Example:</b>  <table> <tr> <td> <pre>void main() {     fragmentColor =     texture(sampler,     ntextureCoordinate); }</pre> </td><td> <pre>0 : (ignorepe) {     itrsm2d.pixel.fcnorm.schedwdf     r0, 1, drc0, cf4, sh4, sh0, 4,     cf0, }</pre> </td></tr> </table>	<pre>void main() {     fragmentColor =     texture(sampler,     ntextureCoordinate); }</pre>	<pre>0 : (ignorepe) {     itrsm2d.pixel.fcnorm.schedwdf     r0, 1, drc0, cf4, sh4, sh0, 4,     cf0, }</pre>
<pre>void main() {     fragmentColor =     texture(sampler,     ntextureCoordinate); }</pre>	<pre>0 : (ignorepe) {     itrsm2d.pixel.fcnorm.schedwdf     r0, 1, drc0, cf4, sh4, sh0, 4,     cf0, }</pre>		

Modifier Name	Description		
.FLR	<p>Take the floor part (before absolute/negate), NaN's, +/-inf and +/-0.0f are all preserved</p> <p><b>Table 8: Example:</b></p> <table> <tr> <td> <pre>void main() {     fragmentColor =         floor(a); }</pre> </td><td> <pre>0    : fadd ft0, sh0.flr, c0       fadd ft1, sh1.flr, c0       mov r0, ft0;       mov r1, ft1;</pre> </td></tr> </table>	<pre>void main() {     fragmentColor =         floor(a); }</pre>	<pre>0    : fadd ft0, sh0.flr, c0       fadd ft1, sh1.flr, c0       mov r0, ft0;       mov r1, ft1;</pre>
<pre>void main() {     fragmentColor =         floor(a); }</pre>	<pre>0    : fadd ft0, sh0.flr, c0       fadd ft1, sh1.flr, c0       mov r0, ft0;       mov r1, ft1;</pre>		
.INTEGER	U, [V] , [S], [T] and [Q] <i>Sample Data</i> are treated as integers		
[.LODM]	<p>LOD Mode possible values:</p> <ul style="list-style-type: none"> <li>-.BIAS</li> <li>-.REPLACE</li> <li>-.GRADIENT</li> </ul>		
.LP	Low Precision		
.NEG	<p>Negate</p> <p><b>Table 9: Example:</b></p> <table> <tr> <td> <pre>void main() {     fragmentColor = -a; }</pre> </td><td> <pre>0    : mov ft0, sh0.neg       mov ft1, sh1.neg       mov r0, ft0;       mov r1, ft1;</pre> </td></tr> </table>	<pre>void main() {     fragmentColor = -a; }</pre>	<pre>0    : mov ft0, sh0.neg       mov ft1, sh1.neg       mov r0, ft0;       mov r1, ft1;</pre>
<pre>void main() {     fragmentColor = -a; }</pre>	<pre>0    : mov ft0, sh0.neg       mov ft1, sh1.neg       mov r0, ft0;       mov r1, ft1;</pre>		
.NNCOORDS	Non Normalised Coordinates		
.ONEMINUS	$x = 1 - x$		
.PPLOD	Per Pixel LOD is enabled – only valid when LODM = Bias or Replace		
.PROJ	Per Pixel LOD is enabled – only valid when LODM = Bias or Replace		
.ROUNDZERO	Round value to zero		
.SAT	<p>Saturate iterated coordinates to 0.0..1.0</p> <p><b>Table 10: Example:</b></p> <table> <tr> <td> <pre>void main() {     fragmentColor =         clamp(a, 0.0, 1.0); }</pre> </td><td> <pre>0    : fadd.sat ft0, sh0, c0       fadd.sat ft1, sh1, c0       mov r0, ft0;       mov r1, ft1;</pre> </td></tr> </table>	<pre>void main() {     fragmentColor =         clamp(a, 0.0, 1.0); }</pre>	<pre>0    : fadd.sat ft0, sh0, c0       fadd.sat ft1, sh1, c0       mov r0, ft0;       mov r1, ft1;</pre>
<pre>void main() {     fragmentColor =         clamp(a, 0.0, 1.0); }</pre>	<pre>0    : fadd.sat ft0, sh0, c0       fadd.sat ft1, sh1, c0       mov r0, ft0;       mov r1, ft1;</pre>		
[.SBMode]	<p><i>Sample Bypass Mode</i> possible values:</p> <ul style="list-style-type: none"> <li>-.DATA</li> <li>-.INFO</li> <li>-.BOTH</li> </ul> <p><b>Table 11: Example:</b></p> <table> <tr> <td> <pre>uniform highp sampler2DShadow sampler;  void main() {     fragmentColor =         vec4(texture(sampler,             vec3(textureCoordinate,                 0.0))); }</pre> </td><td> <pre>11   : smp2d.fcnorm.both drc0,       sh12, r1, sh4, _, r0, 1;</pre> </td></tr> </table>	<pre>uniform highp sampler2DShadow sampler;  void main() {     fragmentColor =         vec4(texture(sampler,             vec3(textureCoordinate,                 0.0))); }</pre>	<pre>11   : smp2d.fcnorm.both drc0,       sh12, r1, sh4, _, r0, 1;</pre>
<pre>uniform highp sampler2DShadow sampler;  void main() {     fragmentColor =         vec4(texture(sampler,             vec3(textureCoordinate,                 0.0))); }</pre>	<pre>11   : smp2d.fcnorm.both drc0,       sh12, r1, sh4, _, r0, 1;</pre>		

Modifier Name	Description		
.SNO	<i>Sample Number</i> is supplied		
.SOO	<p><i>Sample Offset</i> is supplied</p> <p><b>Table 12: Example:</b></p> <table> <tr> <td> <pre> void main() {     fragmentColor =         textureOffset(sampler,             textureCoordinate, ivec2(1,                 1)); } </pre> </td><td> <pre> 9      : if(!p0) {     smp2d.fcnorm.soo drc0,         sh8, r32, sh0, _, r0, 4; } </pre> </td></tr> </table>	<pre> void main() {     fragmentColor =         textureOffset(sampler,             textureCoordinate, ivec2(1,                 1)); } </pre>	<pre> 9      : if(!p0) {     smp2d.fcnorm.soo drc0,         sh8, r32, sh0, _, r0, 4; } </pre>
<pre> void main() {     fragmentColor =         textureOffset(sampler,             textureCoordinate, ivec2(1,                 1)); } </pre>	<pre> 9      : if(!p0) {     smp2d.fcnorm.soo drc0,         sh8, r32, sh0, _, r0, 4; } </pre>		
.TAO	<i>Texture Address Override</i>		
[.type]	<p>Data type possible values:</p> <ul style="list-style-type: none"> <li>-.F32</li> <li>-.U16</li> <li>-.S16</li> <li>-.U8</li> <li>-.S8</li> <li>-.U32</li> <li>-.S32</li> </ul>		
.ZABS	Absolute modifier for the toF16 operand of the Z term		
.ZCLAMP	Clamp bit for the toF16 operand of the Z term		

## 6. Floating Point Instructions

### FMAD

**Format:** FMAD dest, source1, source2, source3.

**Phase0:** FMAD{.LP}{.SAT} FT0, S0{.ABS}{.NEG}, S1{.ABS}{.NEG}, S2{.ABS}{.NEG}{.FLR}

**Phase1:** FMAD{.LP}{.SAT} FT1, S3{.ABS}{.NEG}, S4{.ABS}{.NEG}, S5{.ABS}{.NEG}{.FLR}

**Phase2:** -

**Description:**

```
FT0 = S0 * S1 + S2
```

```
FT1 = S3 * S4 + S5
```

```
void main()
{
    fragColor = a * b + c;
}
```

```
2    : fmad ft0, sh5, i3, sh9
      fmad ft1, sh4, i1, i0
      mov r0, ft1;
      mov r1, ft0;
```

### FADD

**Format:** FADD dest, source1, source2.

**Phase0:** FADD{.SAT} FT0, S0{.ABS}{.NEG}{.FLR}, S1{.ABS}

**Phase1:** FADD{.SAT} FT1, S3{.ABS}{.NEG}{.FLR}, S4{.ABS}

**Phase2:** FADD{.SAT} FT2, IS3{.ABS}{.NEG}, FTE{.ABS}

**Description:**

```
FT0 = S0 + S1
```

```
FT1 = S3 + S4
```

```
FT2 = IS3 + FTE
```

**Table 13: Example:**

```
void main()
{
    fragColor = a + b;
}
```

```
1    : fmad ft0, sh4, c64, sh0
      fadd ft1, sh5, i0
      mov r0, ft0;
      mov r1, ft1;
```

### FMUL

**Format:** FMUL dest, source1, source2.

**Phase0:** FMUL{.SAT} FT0, S0{.ABS}{.NEG}{.FLR}, S1{.ABS}

**Phase1:** FMUL{.SAT} FT1, S3{.ABS}{.NEG}{.FLR}, S4{.ABS}

**Phase2:** FMUL{.SAT} FT2, IS3{.ABS}{.NEG}, FTE{.ABS}

**Description:**

```
FT0 = S0 * S1
```

```
FT1 = S3 * S4
```

```
FT2 = IS3 * FTE
```

**Table 14: Example:**

<pre>void main() {     fragColor = a * b; }</pre>	<pre>1    : fmul ft0, sh4, i1       fmul ft1, sh5, i0       mov r0, ft0;       mov r1, ft1;</pre>
---	---

## FRCP

**Format:** FRCP dest, source.

**Phase0:** FRCP W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:**

```
W0 = 1 / S0
```

Special cases:

```
FRCP(NaN) = +NaN
FRCP(+/-INF) = +/-0
FRCP(+/-0) = +/-INF
FRCP(+/-1.0) = +/-1.0 exactly
```

**Table 15: Example:**

<pre>void main() {     fragColor = 1.0 / a; }</pre>	<pre>0    : frcp r0, sh0</pre>
---	--------------------------------

## FRSQ

**Format:** FRSQ dest, source.

**Phase0:** FRSQ W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:**

```
W0 = 1 / sqrt(S0)
```



Special cases:

```
FRSQ (Any NaN) = +NaN
FRSQ (+INF) = +0
FRSQ (+/-0) = +/-INF
FRSQ (-X) = +NaN
FRSQ(1.0) = 1.0 exactly
```

**Table 16: Example:**

<pre>void main() {     fragColor = 1.0 / sqrt(a); }</pre>	<pre>0    : frsq r0, sh0</pre>
---	--------------------------------

## FSQRT

**Format:** FRSQT dest, source.

**Phase0:** FSQRT W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:**

```
W0 = sqrt(S0)
```

**Table 17: Example:**

<pre>void main() {     fragColor = sqrt(a); }</pre>	<pre>0    : fsqrt r0, sh0</pre>
---	---------------------------------

## FLOG

**Format:** FLOG dest, source.

**Phase0:** FLOG W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:**

```
W0 = log2(S0)
```

Special cases:

```
log2 (Any NaN) = +NaN
log2 (+INF) = +INF
log2 (+/-0) = -INF
log2 (-X) = +NaN
log2(1.0) = +0.0
```

Table 18: Example:

<pre>void main() {     fragColor = log2(a); }</pre>	<pre>0    : flog r0, sh0</pre>
---	--------------------------------

## FEXP

**Format:** FEXP dest, source.

**Phase0:** FEXP W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:**

W0 = exp2(S0)

Special cases:

exp2(Any NaN) = +NaN  
exp2(+INF) = +INF  
exp2(+/-0) = +1.0

Table 19: Example:

<pre>void main() {     fragColor = exp2(a); }</pre>	<pre>0    : fexp r0, sh0</pre>
---	--------------------------------

## GCMP

**Format:** GCMP dest, source.

**Phase0:** GCMP W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:** Gamma compression

W0 = gcmp(S0)

## GEXP

**Format:** GEXP dest, source.

**Phase0:** GEXP W0{.F16.E0.E1}, S0{.F16.E0|.F16.E1}{.ABS}{.NEG}

**Phase1:** -

**Phase2:** -

**Description:** Gamma expansion

```
W0 = gexp(S0)
```

## F16SOP

**Format:** F16SOP jDest, kDest, movDest, jArgumentA, jArgumentE, jOp, jArgumentB, jArgumentF, kArgumentC, kArgumentG, kOp, kArgumentD, kArgumentH, movSource

**Phase0:** F16SOP W0{.E0|.E1}.jOut, (W0|W1){.E0|.E1}.kOut, (W1|\_), Sn{.E0|.E1}{.NEG}, (Sn|0){.E0|.E1}{.ONEMINUS}, jOp, Sn{.E0|.E1}{.NEG}, (Sn|0){.E0|.E1}{.ONEMINUS}, Sn{.E0|.E1}{.NEG}, (Sn|0){.E0|.E1}{.ONEMINUS}, kOp, Sn{.E0|.E1}{.NEG}, (Sn|0){.E0|.E1}{.ONEMINUS}, (S3|\_)

**Phase1:** -

**Phase2:** -

**Description:** 16-bit floating point sum of products, one source.

```
a, b, c, d = {S0, S1, S2, S3, S4, S5}.
z = min(s1, 1 - s0)
e, f, g, h = {S0, S1, S2, S3, S4, S5} or z.
v, w, x, y = {S0, S1, S2, S3, S4, S5}.
jop = any of {add, sub, min, max, rsub, mad}
kop = any of {add, sub, min, max, rsub}

if (jop == mad)
{
    W0.e0 = a*e+v
    W1.e0 = b*f+x
    W0.e1 = c*g+w
    W1.e1 = d*h+y
}
else
{
    j = (a * e) jop (b * f)
    k = (c * g) kop (d * h)

    if (rfmt(1) = 1)
    {
        w1 = toF32([k])
        w0 = toF32([j])
    }
    else if (rfmt(0) = 1) then
    {
        w0[31:16] = one of {j, a, b}
        w0[15:0]  = one of {k, c, d}
    }
    else
    {
        w0[31:16] = one of {k, c, d}
        w0[15:0]  = one of {j, a, b}
    }
}
```

**Table 20: Example:**

<pre> <b>void</b> main() {     mediump vec4 a16 = a;     mediump vec4 b16 = b;     mediump vec4 c16 = c;     fragColor = a16 * b16 + c16; } </pre>	<pre> 2      : sop r0.joutj, sh4, i3, add, sh8,       0.oneminus         sop r1.koutk, sh5, i1, add, i0,       0.oneminus </pre>
--	--

# SOPMOV

**Format:** SOPMOV movDest, movSource.

**Phase0:** SOPMOV.U8{.ZABS}{.ZCLAMP} W0, W1, S3

### Phase1: -

**Phase2:** SOPMOV (W1|\_), (S3|\_)

**Description:** MOV part of the F16SOP operation.

# F16SOP.MAD

**Format:** F16SOP.MAD dest1, dest2, dest3, dest4, argumentA1, argumentE1, argumentV1, argumentC2, argumentG2, argumentX2, argumentB3, argumentF3, argumentW3, argumentD4, argumentH4, argumentY4

**Phase0:** F16SOP.MAD W0{.F16}{.E0}{.CLAMP}, W0{.F16}{.E1}{.CLAMP}, W1{.F16}{.E0}{.CLAMP}, W1{.F16}{.E1}{.CLAMP}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}

### Phase1: -

## Phase2: -

**Description:** Multiply-add 16-bit floating point sum of products, one source

It is possible to do 4 SOPMADs in one cycle.

Table 21: Example:

<pre>void main() {     mediump vec4 a16 = a;     mediump vec4 b16 = b;     mediump vec4 c16 = c;     fragColor = a16 * b16 + c16; }</pre>	<pre>2      : sop r0.joutj, sh4, i3, add, sh8,       0.oneminus       sop r1.koutk, sh5, i1, add, i0,       0.oneminus</pre>
---	--

## F16SOP.U8

**Format:** F16SOP.U8 jDest, kDest, movDest, jArgumentA, jArgumentE, jOp, jArgumentB, jArgumentF, kArgumentC, kArgumentG, kOp, kArgumentD, kArgumentH, movSource

**Phase0:** F16SOP.U8{.ZABS}{.ZCLAMP} W0{.U8}{.E0|.E1|.E2|.E3}.JOUTJC, W0{.U8}{.E0|.E1|.E2|.E3}.KOUTKC, W1, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, jOp, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, kOp, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, S3

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 8-bit floating point sum of products, one source

Output elements can be in F16 or U8 format, depending on the mode of operation.

## SOPMOV.U8

**Format:** SOPMOV.U8 dest, movDest, movSource

**Phase0:** SOPMOV.U8{.ZABS}{.ZCLAMP} W0, W1, S3

**Phase1:** -

**Phase2:** -

**Description:** MOV part of unsigned 8-bit floating point sum of products, one source.

## F16SOP.U8MAD

**Format:** F16SOP.U8MAD dest1, dest2, dest3, dest4, movDest, argumentA1, argumentE1, argumentV1, argumentC2, argumentG2, argumentX2, argumentB3, argumentF3, argumentW3, argumentD4, argumentH4, argumentY4, movSource

**Phase0:** F16SOP.U8MAD W0{.U8}{.E0}{.CLAMP}, W0{.U8}{.E1}{.CLAMP}, W0{.U8}{.E2}{.CLAMP}, W0{.U8}{.E3}{.CLAMP}, W1, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}

{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}  
 {.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR},  
 (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0)  
 {.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}  
 {.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}  
 {.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}  
 {.CLAMP}{.ABS}{.FLR}, Sn{.U8|.F16}{.E0|.E1|.E2|.E3}{.NEG}{.ABS}{.FLR}, (Sn|  
 0){.U8|.F16}{.E0|.E1|.E2|.E3}{.ONEMINUS}{.CLAMP}{.ABS}, (Sn|0){.U8|.F16}  
 {.E0|.E1|.E2|.E3}{.NEG}{.CLAMP}{.ABS}{.FLR}, S0

**Phase1:** -

**Phase2:** -

**Description:** Multiply-add unsigned 8-bit floating point sum of products, one source.

## SOPU8MADMOV

**Format:** SOPU8MADMOV movDest, movSource.

**Phase0:** -

**Phase1:** -

**Phase2:** SOPU8MADMOV W1, S0

**Description:** MOV part of Multiply-add unsigned 8-bit floating point sum of products, one source.

## MBYP

**Format:** MBYP dest, source.

**Phase0:** MBYP FT0, S0{.NEG}{.ABS}

**Phase1:** MBYP FT1, S3{.NEG}{.ABS}

**Phase2:** -

**Description:**

FT0 = S0

FT1 = S3

## FDSX

**Format:** FDSX dest, source.

**Phase0:** FDSX FT0, S0{.NEG}{.ABS}

**Phase1:** FDSX FT1, S3{.NEG}{.ABS}-

**Phase2:** -

**Description:** Gradient in x direction, one source

```
D = Pix1 - Pix0
```

Table 22: Example:

<pre>void main() {     vec4 data = texture(texture0,         textureCoordinate);     fragColor = dFdx(data); }</pre>	<pre>4 : fdsx ft0, r2    fdsx ft1, r3    mov r0, ft0;    mov r1, ft1;</pre>
--	---

## FDSY

**Format:** FDSY dest, source.

**Phase0:** FDSY FT0, S0{.NEG}{.ABS}

**Phase1:** FDSY FT1, S3{.NEG}{.ABS}-

**Phase2:** -

**Description:** Gradient in y direction, one source

```
D = Pix2 - Pix0
```

Table 23: Example:

<pre>void main() {     vec4 data = texture(texture0,         textureCoordinate);     fragColor = dFdy(data); }</pre>	<pre>4 : fdsy ft0, r2    fdsy ft1, r3    mov r0, ft0;    mov r1, ft1;</pre>
--	---

## FDSXF

**Format:** FDSXF dest, source.

**Phase0:** FDSXF FT0, S0{.NEG}{.ABS}

**Phase1:** FDSXF FT1, S3{.NEG}{.ABS}

**Phase2:** -

**Description:** Gradient in x direction, one source

```
If (Pix0, Pix1) D = Pix1 - Pix0
```

```
If (Pix2, Pix3) D = Pix3 - Pix2
```

(DX 11 version)

## FDSYF

**Format:** FDSYF dest, source.

**Phase0:** FDSYF FT0, S0{.NEG}{.ABS}

**Phase1:** FDSYF FT1, S3{.NEG}{.ABS}

**Phase2:** -

**Description:** Gradient in y direction, one source

```
If (Pix0, Pix2) D = Pix2 - Pix0
```

```
If (Pix1, Pix3) D = Pix3 - Pix1
```

(DX 11 version)

## CONVERTTOF64

**Format:** CONVERTTOF64.format dest1, dest2, source.

**Phase0:** -

**Phase1:** -

**Phase2:** CONVERTTOF64.(F32|S32|U32) FT0, FT2, IS3

**Description:** Convert (Pack) from 32-bit to 64-bit by data format.

Dest FT0 contains 32-bit LSBs

Dest FT2 contains 32-bit MSBs

## CONVERTFROMF64

**Format:** CONVERTFROMF64.format dest, source1, source2

**Phase0:** CONVERTFROMF64.(F32|S32|U32) FT0, S0, S2

**Phase1:** -

**Phase2:** -

**Description:** Convert (Unpack) from 64-bit to 32-bit by data format.

Source S0 contains 32-bit LSBs.

Source S2 contains 32-bit MSBs.

## FSINC

**Format:** FSINC dest, destPred, source.

**Phase0:** FSINC W0, P0, S0.

**Phase1:** -

**Phase2:** -

**Description:** Trigonometric '(sine x) / x', one source.

Used for calculating sin(x).



```
W0 = sinc(S0)
```

P = perform final multiply

1. Behaviour is not defined for input outside of  $[-1,1]$  range
2.  $\text{sinc}(\text{Any NaN}) = +\text{NaN}$
3. Any input with an exponent  $< (103)$  produces  $\pi/2$

**Table 24: Example:**

<pre>void main() {     fragColor = sin(a); }</pre>	<pre>2    : fsinc r0, p0, i0</pre>
--	------------------------------------

## FARCTANC

**Format:** FARCTANC dest, source.

**Phase0:** FARCTANC W0, S0.

**Phase1:** -

**Phase2:** -

**Description:** Trigonometric  $'(\arctan x)/x'$ , one source.

Used for calculating  $\text{atan}(x)$ .

```
W0 = arctanc(S0)
```

1. Behaviour is not defined for input outside of  $[-1,1]$  range
2.  $\text{FARCTAN}(\text{Any NaN}) = +\text{NaN}$
3. Any input  $< 0x39C00000$  produces 1.0

**Table 25: Example:**

<pre>void main() {     fragColor = atan(a); }</pre>	<pre>5    : if(p0) {     farctanc i0, i1 }</pre>
---	--

## FRED

**Format:** FRED.SIN dest1, dest2, destPred, iterationCount, source1, source2

FRED.COS dest1, dest2, predicate, iterationCount, source1, source2.

**Phase0:** FRED.SIN{.part} W0, W1, (P0|\_), iterationCount, S0{.NEG}{.ABS}, (S3|\_)

FRED.COS{.part} W0, W1, (P0|\_), iterationCount, S0{.NEG}{.ABS}, (S3|\_)

**Phase1:** -

**Phase2:** -

**Description:** Trigonometric range reduction, sine/cosine.

Used for making the result of sin/cos/etc. operations more precise.

If iteration = 0 and PARTA:

```
W0 = Range Reduce(S0)
```

```
W1 = Range Reduce(S0) (data for PARTB)
```

For all other cases:

```
W0 = Range Reduce(S0, S3) (result - PARTB only)
```

```
W1 = Range Reduce(S0, S3) (data for PARTB or next itr)
```

Where S0 is the original input value and S3 is the data from the previous iteration.

```
If P0 is set then P = Perform further iteration
```

This instruction happens in two parts, both parts are required (see example usage below)

W0 does not contain any useful data in part A and may be discarded.

W1 may be discarded on the last iteration.

**Table 26: Example:**

<pre>void main() {     fragColor = sin(a); }</pre>	<pre>0    : fred.sin i1, i0, _, 0, sh0, 1    : fred.sin.partb i0, ft0, _, 0, sh0, i0 2    : fsinc r0, p0, i0</pre>
--	--

## GTA

**Format:** GTA dest1, dest2, destPredOutOfBounds, sourceLookUp, sourceTextureState, sourceUCoord, sourceVCoord, sourceSTQ

**Phase0:** -

**Phase1:** -

**Phase2:** GTA{.ARRAY} W0, W1, (P0|\_), S3, S0, S1, S4, S5

**Description:** Generate Texel Address:

BurstLength, DataSize, Address.

P0 is set if out of bounds.

## 7. Data Movement Instructions

---

### MOV

**Format:** MOV dest, sourceMovW0

**Phase0:** -

**Phase1:** -

**Phase2:** MOV W0{.E0}{.E1}{.E2}{.E3}, (FT0|FT1|FT2|FTE)

**Description:** Data movement.

Modifies the multiplexer assignments for W0 allowing emulation of masked writes.

The source selected from MovW0 (FT0 | FT1 | FT2 | FTE) will be used as W0 output for all bytes for which a MaskW0 element (E0, E1, E2, E3) has been set.

For those bytes where the MaskW0 element has not been set, the W0 output comes from the source selected by IS4.

### MOVC

**Format:** MOVC dest1, dest2, sourceAW, sourceMovW0, sourceMovW1.

**Phase0:** -

**Phase1:** -

**Phase2:** MOVC W0{.E0}{.E1}{.E2}{.E3}, W1, FTT, (FT0|FT1|FT2|FTE), (FT0|FT1|FT2|FTE)

**Description:** Conditional data movement.

A test is mandatory in the instruction group for MOVC.

If the test result is true, then the W0 assignments are modified as for MOV, and W1 is assigned the source selected from MovW1 (the second FT0 | FT1 | FT2 | FTE).

If the test result is false, W0 and W1 are assigned as per IS4 and IS5.

If MaskW0 is not specified, the default value (all elements set/bits are "1111") is assumed.

### PCK

**Format:** PCK.format dest, source.

**Phase0:** -

**Phase1:** -

**Phase2:** PCK.format{.SCALE}{.ROUNDZERO} FT2, IS3

**Description:** Pack by data format

Packed formats are defined as comprising a number of scalars, R0 to Rn. A PCK in an instruction group with a REPEAT instruction group modifier of n converts values from n sequential register locations into a single result.

```
FT2 = pack(IS3)
```

**Data formats supported:** U8888, S8888, S8888OGL, O8888, U1616, S1616, S1616OGL, O1616, F16F16, U32, S32, F32, F32MASK, U1010102, S1010102, 2F102F10F10, U111110, S111110, F111110, SE9995, U565U565, COVERAGE MASK (PCK only), S8D24, D24S8, CONST0 (PCK only), CONST1 (PCK only)

**Table 27: Example:**

<pre>void main() {     fragColor = ec4(float(int(a))); }</pre>	<pre>0 : pck.s32.rndzero ft2, sh0    mov i0, ft2;  1 : unpck.s32 ft0, i0.e0    mov r0.e0.e1.e2.e3, ft0    mov r1, ft0;</pre>
--	--

## UNPCK

**Format:** UNPCK.format dest, source.

**Phase0:** UNPCK.format{.SCALE}{.ROUNDZERO} FT0, S0{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Unpack by data format.

Similarly UPCK with a REPEAT of n produce results in n sequential register locations.

```
FT0 = unpack(S0)
```

**Data formats supported:** U8888, S8888, S8888OGL, O8888, U1616, S1616, S1616OGL, O1616, F16F16, U32, S32, F32, F32MASK, U1010102, S1010102, 2F102F10F10, U111110, S111110, F111110, SE9995, U565U565, S8D24, D24S8

**Table 28: Example:**

<pre>void main() {     fragColor = vec4(float(i)); }</pre>	<pre>0 : unpck.s1616 ft0, sh0.e0    mov r0.e0.e1.e2.e3, ft0    mov r1, ft0;</pre>
--	---

## 8. Integer Instructions

### UADD8

**Format:** UADD8 dest, source1, source2.

**Phase0:** UADD8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 8-bit add.

```
FT0 = S0 + S1
```

Saturate 0..255

**Table 29: Example:**

<pre>uniform lowp uint a; uniform lowp uint b; uniform lowp uint c;  void main() {     fragColor = vec4(a + b); }</pre>	<pre>1 : uadd8 ft0, sh1.e0, i0.e0    mov i0, ft0;</pre>
---	---

### UMUL8

**Format:** UMUL8 dest, source1, source2.

**Phase0:** UMUL8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 8-bit multiply.

```
FT0 = S0 * S1
```

**Table 30: Example:**

<pre>uniform lowp uint a; uniform lowp uint b; uniform lowp uint c;  void main() {     fragColor = vec4(a * b); }</pre>	<pre>1 : umul8 ft0, sh1.e0, i0.e0    mov i0, ft0;</pre>
---	---

## UMAD8

**Format:** UMAD8 dest, source1, source2, source3.

**Phase0:** UMAD8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}, S2{.NEG}{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 8-bit multiply and add.

```
FT0 = S0 * S1 + S2
```

**Table 31: Example:**

<pre>uniform lowp uint a; uniform lowp uint b; uniform lowp uint c;  void main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1    : umad8 ft0, sh1.e0, i0.e0, sh2.e0       mov i0, ft0;</pre>
---	---

## IADD8

**Format:** IADD8 dest, source1, source2.

**Phase0:** IADD8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 8-bit add.

```
FT0 = S0 + S1
```

Saturate -128..127

**Table 32: Example:**

<pre>uniform lowp int a; uniform lowp int b; uniform lowp int c;  void main() {     fragColor = vec4(a + b); }</pre>	<pre>1    : iadd8 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
--	---

## IMUL8

**Format:** IMUL8 dest, source1, source2.

**Phase0:** IMUL8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 8-bit multiply.

```
FT0 = S0 * S1
```

**Table 33: Example:**

<pre>uniform lowp int a; uniform lowp int b; uniform lowp int c;  void main() {     fragColor = vec4(a * b); }</pre>	<pre>1    : imul8 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
--	---

## IMAD8

**Format:** IMAD8 dest, source1, source2, source3.

**Phase0:** IMAD8{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1|.E2|.E3}, S1{.ABS}{.E0|.E1|.E2|.E3}, S2{.NEG}{.ABS}{.E0|.E1|.E2|.E3}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 8-bit multiply and add.

```
FT0 = S0 * S1 + S2
```

**Table 34: Example:**

<pre>uniform lowp int a; uniform lowp int b; uniform lowp int c;  void main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1    : imad8 ft0, sh1.e0, i0.e0, sh2.e0       mov i0, ft0;</pre>
--	---

## UADD16

**Format:** UADD16 dest, source1, source2.

**Phase0:** UADD16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 16-bit add.

```
FT0 = S0 + S1
```

Saturate 0..65535.

**Table 35: Example:**

<pre>uniform mediump uint a; uniform mediump uint b; uniform mediump uint c;  void main() {     fragColor = vec4(a + b); }</pre>	<pre>1    : uadd16 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
--	--

## UMUL16

**Format:** UMUL16 dest, source1, source2.

**Phase0:** UMUL16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 16-bit multiply.

```
FT0 = S0 * S1
```

**Table 36: Example:**

<pre>uniform mediump uint a; uniform mediump uint b; uniform mediump uint c;  void main() {     fragColor = vec4(a * b); }</pre>	<pre>1    : umul16 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
--	--

## UMAD16

**Format:** UMAD16 dest, source1, source2, source3.

**Phase0:** UMAD16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}, S2{.NEG}{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 16-bit multiply and add.

```
FT0 = S0 * S1 + S2
```



Table 37: Example:

<pre>uniform mediump uint a; uniform mediump uint b; uniform mediump uint c;  void main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1    : umad16 ft0, sh1.e0, i0.e0, sh2.e0       mov i0, ft0;</pre>
--	--

## IADD16

**Format:** IADD16 dest, source1, source2.

**Phase0:** IADD16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 16-bit add.

```
FT0 = S0 + S1
```

Saturate -32768..32767

Table 38: Example:

<pre>uniform mediump int a; uniform mediump int b; uniform mediump int c;  void main() {     fragColor = vec4(a + b); }</pre>	<pre>1    : iadd16 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
---	--

## IMUL16

**Format:** IMUL16 dest, source1, source2.

**Phase0:** IMUL16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 16-bit multiply.

```
FT0 = S0 * S1
```

Table 39: Example:

<pre>uniform mediump <b>int</b> a; uniform mediump <b>int</b> b; uniform mediump <b>int</b> c;  <b>void</b> main() {     fragColor = vec4(a * b); }</pre>	<pre>1    : imul16 ft0, sh1.e0, i0.e0       mov i0, ft0;</pre>
---	--

## IMAD16

**Format:** IMAD16 dest, source1, source2, source3.

**Phase0:** IMAD16{.SAT} FT0, S0{.NEG}{.ABS}{.E0|.E1}, S1{.ABS}{.E0|.E1}, S2{.NEG}{.ABS}{.E0|.E1}

**Phase1:** -

**Phase2:** -

**Description:** Signed (integer) 16-bit multiply and add.

$$FT0 = S0 * S1 + S2$$

Table 40: Example:

<pre>uniform mediump <b>int</b> a; uniform mediump <b>int</b> b; uniform mediump <b>int</b> c;  <b>void</b> main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1    : imad16 ft0, sh1.e0, i0.e0, sh2.e0       mov i0, ft0;</pre>
---	--

## ADD64

**Format:** ADD64 dest, source1RHS, source1LHS, source2RHS, source2LHS, sourcePredCarryIn.

**Phase0:** ADD64 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, IS0{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** 64-bit add with carry-in and carry-out bits.

$$COUT, FT0 = (S1 \ll 32 \mid S0) + (IS0 \ll 32 \mid S2) [+P0]$$

## UADD6432

**Format:** UADD6432 dest, source1RHS, source1LHS, source2, sourcePredCarryIn.

**Phase0:** UADD6432 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 64-bit and 32-bit add with carry-in bit.

```
FT0 = (S1<<32 | S0) + S2 [+P0]
```

## SADD6432

**Format:** SADD6432 dest, source1RHS, source1LHS, source2, sourcePredCarryIn.

**Phase0:** SADD6432 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Signed 64-bit and 32-bit add with carry-in bit.

```
FT0 = (S1<<32 | S0) + S2 [+P0]
```

## UMADD32

**Format:** UMADD32 dest, source1, source2, source3, sourcePredCarryIn.

**Phase0:** UMADD32 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 32-bit multiply and add with carry-in bit.

```
FT0 = S0 * S1 + S2 [+P0]
```

**Table 41: Example:**

<pre>uniform highp uint a; uniform highp uint b; uniform highp uint c;  void main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1 : umadd32 ft0, sh1, i0, sh2,    mov i0, ft0;</pre>
--	---

## SMADD32

**Format:** SMADD32 dest, source1, source2, source3, sourcePredCarryIn.

**Phase0:** SMADD32 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Signed 32-bit multiply and add with carry-in bit.

$$FT0 = S0 * S1 + S2 [+P0]$$

**Table 42: Example:**

<pre>uniform highp int a; uniform highp int b; uniform highp int c;  void main() {     fragColor = vec4(a * b + c); }</pre>	<pre>1    : smadd32 ft0, sh1, i0, sh2,       mov i0, ft0;</pre>
---	---

## UMADD64

**Format:** UMADD64 dest, source1, source2, source3RHS, source3LHS, sourcePredCarryIn

**Phase0:** UMADD64 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, IS0{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Unsigned 64-bit multiply and add with carry-in bit.

$$FT0 = S0 * S1 + (IS0 << 32 | S2) [+P0]$$

## SMADD64

**Format:** SMADD64 dest, source1, source2, source3RHS, source3LHS, sourcePredCarryIn

**Phase0:** SMADD64 FT0, S0{.NEG}{.ABS}, S1{.NEG}{.ABS}, S2{.NEG}{.ABS}, IS0{.NEG}{.ABS}, (P0|\_)

**Phase1:** -

**Phase2:** -

**Description:** Signed 64-bit multiply and add with carry-in bit.

$$FT0 = S0 * S1 + (IS0 << 32 | S2) [+P0]$$

## 9. Test Instructions

### TSTZ

**Format:** TSTZ dest, destPredWrite, source.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTZ{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test zero.

**Table 43: Example:**

<pre> uniform highp int a; uniform highp int b; uniform highp int c;  void main() {     highp int res;      if( a == 0 )     {         res = b;     }     else     {         res = c;     }      fragColor = vec4(res); } </pre>	<pre> 0      : mov ft0, sh1         mov ft1, sh2         tstz.s32 ftt, sh0         mov i0.e0.e1.e2.e3, ft1, ftt, ft0, ft1 </pre>
--	--

### TSTGZ

**Format:** TSTGZ dest, destPredWrite, source.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTGZ{.type} FTT, (P0|\_), IS1{.E0|.E1|.E2|.E3}

**Description:** Test greater than zero.

Table 44: Example:

<pre> uniform highp <b>int</b> a; uniform highp <b>int</b> b; uniform highp <b>int</b> c;  <b>void</b> main() {     highp <b>int</b> res;      <b>if</b>( a &gt; 0 )     {         res = b;     }     <b>else</b>     {         res = c;     }      fragColor = vec4(res); } </pre>	<pre> 0      : mov ft0, sh1         mov ft1, sh2         tstgz.s32 ftt, _, sh0         mov i0.e0.e1.e2.e3, ft1, ftt, ft0, ft1 </pre>
---	--

## TSTGEZ

**Format:** TSTGEZ dest, destPredWrite, source.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTGEZ{.type} FTT, (P0|\_), IS1{.E0|.E1|.E2|.E3}

**Description:** Test greater than or equal to zero.

Table 45: Example:

<pre> uniform highp <b>int</b> a; uniform highp <b>int</b> b; uniform highp <b>int</b> c;  <b>void</b> main() {     highp <b>int</b> res;      <b>if</b>( a &gt;= 0 )     {         res = b;     }     <b>else</b>     {         res = c;     }      fragColor = vec4(res); } </pre>	<pre> 0      : mov ft0, sh1         mov ft1, sh2         tstgez.s32 ftt, _, sh0         mov i0.e0.e1.e2.e3, ft1, ftt, ft0, ft1 </pre>
--	---

## TSTC

**Format:** TSTC dest, destPredWrite.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTC{.type} FTT, (P0|\_)

**Description:** Test integer carry out.

## TSTE

**Format:** TSTE dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTE{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test equal.

```
LHS == RHS
```

**Table 46: Example:**

<pre>uniform highp int a; uniform highp int b; uniform highp int c;  void main() {     highp int res;      if( a == b )     {         res = b;     }     else     {         res = c;     }      fragColor = vec4(res); }</pre>	<pre>0 : mov ft0, sh2    mov ft1, sh0    tste.s32 ftt, sh1, ft1    mov i0.e0.e1.e2.e3, ft0, ftt, ft1, ft0</pre>
--	---

## TSTG

**Format:** TSTG dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTG{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test greater than.

```
LHS > RHS
```

Table 47: Example:

<pre> uniform highp int a; uniform highp int b; uniform highp int c;  void main() {     highp int res;      if( a &gt; b )     {         res = b;     }     else     {         res = c;     }      fragColor = vec4(res); } </pre>	<pre> 0      : mov ft0, sh2         mov ft1, sh0         tstg.s32 ftt, _, sh1, ft1         mov i0.e0.e1.e2.e3, ft0, ftt, ft1, ft0 </pre>
--	--

## TSTGE

**Format:** TSTGE dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTGE{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test greater than or equal.

```
LHS >= RHS
```

## TSTNE

**Format:** TSTNE dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTNE{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test not equal.

```
LHS != RHS
```



Table 48: Example:

<pre> uniform highp int a; uniform highp int b;  void main() {     highp int res;      if( a != b )     {         res = a;     }     else     {         res = b;     }      fragColor = vec4(res); } </pre>	<pre> 0    : mov ft0, sh0       mov ft1, sh1       tstne.s32 ftt, _, ft0, ft1       mov i0.e0.e1.e2.e3, ft1, ftt, ft0, ft1 </pre>
---	---

## TSTL

**Format:** TSTL dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTL{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test less than.

LHS < RHS

Table 49: Example:

<pre> uniform highp int a; uniform highp int b;  void main() {     highp int res;      if( a &lt; b )     {         res = a;     }     else     {         res = b;     }      fragColor = vec4(res); } </pre>	<pre> 0    : mov ft0, sh0       mov ft1, sh1       tstl.s32 ftt, _, ft0, ft1       mov i0.e0.e1.e2.e3, ft1, ftt, ft0, ft1 </pre>
---	--

## TSTLE

**Format:** TSTLE dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTLE{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test less than or equal.

LHS <= RHS

## TSTMIN

**Format:** TSTMIN dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTMIN{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test minimum.

LHS < RHS

## TSTMAX

**Format:** TSTMAX dest, destPredWrite, sourceLHS, sourceRHS.

**Phase0:** -

**Phase1:** -

**Phase2:** TSTMAX{.type} FTT, (P0|\_), IS1{.F16}{.E0|.E1|.E2|.E3}, IS2{.F16}{.E0|.E1|.E2|.E3}

**Description:** Test maximum.

LHS >= RHS

## 10. Bitwise Instructions

### AND

**Format:** AND dest, source1, source2, source3, source4.

**Phase0:** -

**Phase1:** AND FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise AND

FT1 is bit mask

```
FT4 = {FT1 &} FT2 AND {~FT1 &} S3
```

**Table 50: Example:**

<pre>uniform highp int a; uniform highp int b;  void main() {     fragColor = vec4(a &amp; b); }</pre>	<pre>0    : mov ft0, ft1, c0, c0       mov ft2, sh1       cbs ft3, sh1       and ft4, _, ft2, _, sh0       lsl ft5, ft4, c0, sh0       mov i0, ft5;</pre>
--	---

### OR

**Format:** OR dest, source1, source2, source3, source4.

**Phase0:** -

**Phase1:** OR FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise OR

FT1 is bit mask

```
FT4 = {FT1 &} FT2 OR {~FT1 &} S3
```

**Table 51: Example:**

<pre>uniform highp int a; uniform highp int b;  void main() {     fragColor = vec4(a   b); }</pre>	<pre>0    : mov ft0, ft1, c0, c0       mov ft2, sh1       cbs ft3, sh1       or ft4, _, ft2, _, sh0       lsl ft5, ft4, c0       mov i0, ft5;</pre>
--	---

## XOR

**Format:** XOR dest, source1, source2, source3, source4.

**Phase0:** -

**Phase1:** XOR FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise XOR

FT1 is bit mask

```
FT4 = {FT1 &} FT2 XOR {~FT1 &} S3
```

**Table 52: Example:**

<pre>uniform highp int a; uniform highp int b;  void main() {     fragColor = vec4(a ^ b); }</pre>	<pre>0    : mov ft0, ft1, c0, c0       mov ft2, sh1       cbs ft3, sh1       xor ft4, _, ft2, _, sh0       lsl ft5, ft4, c0 -, sh0       mov i0, ft5;</pre>
--	---

## NAND

**Format:** NAND dest, source1, source2, source3, source4

**Phase0:** -

**Phase1:** NAND FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise NAND

FT1 is bit mask

```
FT4 = {FT1 &} FT2 NAND {~FT1 &} S3
```

## NOR

**Format:** NOR dest, source1, source2, source3, source4.

**Phase0:** -

**Phase1:** NOR FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise NOR

FT1 is bit mask

```
FT4 = {FT1 &} FT2 NOR {~FT1 &} S3
```

## XNOR

**Format:** XNOR dest, source1, source2, source3, source4.

**Phase0:** -

**Phase1:** XNOR FT4, (FT1|\_), FT2, (FT1.INVERT|\_), S3

**Phase2:** -

**Description:** Bitwise XNOR

FT1 is bit mask

```
FT4 = {FT1 &} FT2 XNOR {~FT1 &} S3
```

## SHFL

**Format:** SHFL dest, source1, source2.

**Phase0:** SHFL FT2, S2, S1

**Phase1:** -

**Phase2:** -

**Description:** Bitwise interleave of least significant 16 bits of S1 and S2

```
S1 = -----ABCDEFGHIJKLMN
```

```
S2 = -----abcdefghijklmn
```

```
FT2 = AaBbCcDdEeFfGgHhIiJjKkLlMmNnOoPp
```

## REV

**Format:** REV dest, source1

**Phase0:** REV FT2, S2

**Phase1:** -

**Phase2:** -

**Description:** Bitwise Reverse of S2

```
S2 = ABCDEFGHIJKLMNOPabcdefghijklmn
```

```
FT2 = ponmlkjihgfedcbaPONMLKJIHGFEDCBA
```

## LSL

**Format:** LSL dest, source1, source2

**Phase0:** LSL FT2, S2, S1

**Phase1:** -**Phase2:** LSL FT5, FT4, S4**Description:** Left Shift (Phase0)

```
FT2 = S2 << S1
```

**Note:** If S1 is greater than 32, then shift it by the modulo width.

Left shift (Phase2)

```
FT5 = FT4 << S4
```

**Table 53: Example:**

```
uniform highp int a;
uniform highp int b;

void main()
{
    fragColor = vec4(a << b);
}
```

```
1 : mov ft0, ft1, c0, c0
   mov ft2, sh0
   cbs ft3, sh0
   or ft4, _, ft2, _, c0
   lsl ft5, ft4, i0
   mov i0, ft5;
```

## CPS

**Format:** CPS dest, sourcePred, source2, source3.**Phase0:** -**Phase1:** -**Phase2:** CPS FT5, P0, FT4, S4**Description:** Clear, Predicated Set

```
FT5 = (S4 & ~FT4) | (P0 ? FT4 : 0)
```

## SHR

**Format:** SHR dest, source1, source2**Phase0:** -**Phase1:** -**Phase2:** SHR FT5, FT4, S4**Description:** Shift Right

```
FT5 = (unsigned) FT4 >> S4
```

**Note:** If S4 is greater than 32, then shift it by the modulo width

## ASR

**Format:** ASR.signPos dest, source1, source2

**Phase0:** -

**Phase1:** -

**Phase2:** ASR.signPos FT5, FT4, S4

**Description:** Arithmetic Shift Right

FT5 = arithmetic shift right of FT4 by S4 bits, where modifier SignPos indicates position of sign bit;

```
switch (SignPos)
{
  case TWB:
    sb=31;
    break; // Top Word
  case PWB:
    sb=15;
    break; // Partial Word
  case MTB:
    sb=FT0&0x1F;
    break; // Mask Top
  case FTB:
    sb=FT3&0x1F;
    break; // Find Top
}
FT5=(signed) (FT4<<(31-sb))>>(31+S4-sb)
```

**Table 54: Example:**

```
uniform highp int a;
uniform highp int b;

void main()
{
  fragColor = vec4(a >> b);
}
```

```
1 : mov ft0, ft1, c0, c0
   mov ft2, sh0
   cbs ft3, sh0
   or ft4, _, ft2, _, c0
   asr.twb ft5, ft4, i0
   mov i0, ft5;
```

## ROL

**Format:** ROL dest, source1, source2.

**Phase0:** -

**Phase1:** -

**Phase2:** ROL FT5, FT4, S4

**Description:** Rotate Left

FT5 = rotate left of FT4 by S4

## TZ

**Format:** TZ destPred, source.

**Phase0:** -

**Phase1:** -

**Phase2:** TZ P0, (FT3|FT5)

**Description:** Test Zero.

## TNZ

**Format:** TNZ destPred, source.

**Phase0:** -

**Phase1:** -

**Phase2:** TNZ P0, (FT3|FT5)

**Description:** Test Not Zero.

## BYP

**Format:** BYP dest, destBM, sourceBM1, sourceBM2

BYP dest, sourceShift1

BYP dest, sourceCount

BYP dest, sourcePhase1

**Phase0:** (BM) BYP FT0, FT1, S0, (S1|immediate)

(Shift1) BYP FT2, S2

(Count) BYP FT3, (S2|FT2)

**Phase1:** BYP FT4, S1

**Phase2:** -

**Description:** Bypass (Phase0, BM)

FT0 = S0

FT1 = S1 or immediate value

Bypass (Phase0, Shift1)

FT2 = S2

Bypass (Phase1)

FT4 = FT1

Bypass (Phase0, Count)

FT3 = S2 or FT2

## MSK

**Format:** MSK dest1, dest2, source1, source2.

**Phase0:** MSK FT0, FT1, S0, S1



**Phase1:****Phase2:** -**Description:** Make mask

S0 - Width of mask in bits (6 bits)

S1 - Bit Position of mask LSB (6 bits)

```
FT0 = (S0+S1)-1 # Result with 6 bits
```

```
FT1 = ((1 << S0) - 1) << S1
```

## CBS

**Format:** CBS dest, source**Phase0:** CBS FT3, (S2|FT2)**Phase1:** -**Phase2:** -**Description:** Count bits set**Table 55: Example:**

```
c=0;
for (i=0; i<32; i++)
    c += (S0>>i) & 1;
FT3=c;
```

## FTB

**Format:** FTB dest, source**Phase0:** FTB FT3, (S2|FT2)**Phase1:** -**Phase2:** -**Description:** Find top bit set, starting from the MSB – Where result is MSB=31 down to LSB=0

Return -1 if no bits set.

**Table 56: Example:**

```
c = 0;
while ((unsigned) S2>>c) ++c;
FT3 = c-1;
```

## FTB\_SHI

**Format:** FTB\_SHI dest, source.

**Phase0:** FTB\_SHI FT3, (S2)

**Phase1:** -

**Phase2:** -

**Description:** Find top bit with signal

Find the first 0 from the MSB if the number is negative, else the first 1 from the MSB. Return -1 if no bits set.

**Table 57: Example:**

```
c=0;
while ((S2[31] ^ S2)>>c) ++c;
FT3=c-1;
```

## FTB\_MSB

**Format:** FTB\_MSB dest, source.

**Phase0:** FTB\_MSB FT3, (FT2)

**Phase1:** -

**Phase2:** -

**Description:** Find top bit set, starting from the MSB – Where result is MSB=0 down to LSB=31

Return -1 if no bits set.

**Table 58: Example:**

```
If ((unsigned) ft2 ==0)
    FT3 = -1
else
    c=31;
    while ((unsigned) ft2>>32-c) --c;
    FT3=c;
```

# 11. Backend Instructions

A summary of the available backend instructions on PowerVR GPUs

Name	Format	Backend Phase	Description
UVSW	UVSW dest, source	UVSW.writeOp (W0 W1), (S0 S1 S2 S3 S4 S5 _ 0...255)	UVS data write <pre>void main() {     gl_Position = inVertex; }  0 : mov ft0, vi3     mov ft0.e0.e1.e2.e3,     ft0     uvsw.write ft0, 3;</pre>
TESSW	TESSW dest, source	TESSW.tessOp (W0 W1), immediateAddress	Tessellator data write.
ATST	ATST dest, sourceRef, sourceData, sourceStateW	ATST{.IFB} (P0 _), S0, S1, S2	Alpha Test
DEPTH	DEPTHF dest	DEPTHF W0	Depth Feedback
FITR	FITR dest, sourceDRC, sourceCoeffPt sourceCount	FITR.mode{.SAT} S3, (DRC0 DRC1), S0, (1...maxItrCount)	Iterate value(s) <pre>in vec2 textureCoordinate; out vec4 fragColor;  void main() {     fragColor = vec4(textureCoordinate, 0.0,     1.0); }  0 : fitr.pixel r0,     drc0, cf4,     cf0, 2;</pre>
FITRP	FITRP dest, sourceDRC, sourceCoeffPt sourceWCoeff sourceCount	FITRP.mode{.SAT} S3, (DRC0 DRC1), S0, S2, (1...maxItrCount)	All values should be multiplied by 1/W
IDF	IDF sourceDRC, sourceSelect	IDF{.DIRECT} (DRC0 DRC1), (S0 S1 S2 S3 S4 S5)	Issue data fence through memory subsystem. The fence is issued for the first valid instance in a task, in the following order; {16, ..., 31, 0, ..., 15}
LD	LD dest, sourceDRC, sourceBurstLe sourceAddress	LD{.DIRECT} S3, (DRC0 DRC1), (S0 S1 S2 S3 S4 S5 1...16), (S0 S1 S2 S3 S4 S5)	Loads data from memory into supplied destination
ST	ST sourceData, sourceDataSiz sourceDRC, sourceBurstLe sourceAddress sourceCovera ST.TEXELMOD sourceData, source, sourceDRC	ST{.TILED} {.DIRECT} (S0 S1 S2 S3 S4 S5), (0...2), (DRC0 DRC1), (S0 S1 S2 S3 S4 S5 1...16), (S0 S1 S2 S3 S4 S5), (S0 S1 S2 S3 S4 S5 _ _) ST.TEXELMODE S2, W1, ST.TEXELMOD (DRC0 DRC1)	Stores data from supplied source to memory. Stores data to memory, texel mode

Name	Format	Backend Phase	Description
SMP (SMP1D SMP2D SMP3D	SMP1D sourceDRC, sourceTexture sourceData, sourceSample sourceShared sourceDestPtr chan	SMP1D{.PROJ}{.FCNORM} {.NNCOORDS}{.LODM} {.PPLD}{.TAO}{.SOO} {.SNO} {.WRT}{.SBMode}{.ARRAY} {.INTEGER}{.COMPARISON} {.SCHEDSWAP}{.DIRECT} (DRC0 DRC1), S0, S1, S2, (S3 _), S4, chan	Sample Texture. One, two or three dimensions  <pre> in vec2 textureCoordinate; out vec4 fragColor;  void main() {     fragColor =     texelFetch(sampler, ivec2(gl_FragCoord.xy), 0); }  3      : smp2d.fcnorm.replace.integer drc0, sh4,       r1, sh0, sh11, r0, 4; </pre>
ATOM	ATOM.opCode sourceDestSe sourceDRC, sourceSelect	ATOM.opCode {.DIRECT} (S0  S1 S2 S3 S4 S5), (DRC0  DRC1), (S0 S1 S2 S3 S4 S5)	Loads data from memory which is operated on with supplied data and operation type is written back to memory and supplied destination

## 12. Flow Control Instructions

A summary of the available flow control instructions for PowerVR GPUs

Name	Format	Construction	Description
BA	BA source	BA{.ALLINST .ANYINST} immediateAddressOrOffset	Branch absolute to Addr, optionally using the modifiers.
BAL	BAL source	BAL{.ALLINST .ANYINST} immediateAddressOrOffset	Save link pointer then branch absolute to Addr, optionally using the modifiers.
BR	BR source	BR{.ALLINST .ANYINST} immediateAddressOrOffset	Branch relative by Offset, optionally using the modifiers.
BRL	BRL source	BRL{.ALLINST .ANYINST} immediateAddressOrOffset	Save link pointer then branch relative by Offset, optionally using the modifiers.
BPRET	BPRET	BPRET	Branch absolute to saved Breakpoint Return address. The predicate condition code must be set to "always".
LAPC	LAPC	LAPC	Link address to program counter (=RET).
SAVL	SAVL source	SAVL W0	Save (move) link address (Dest W0 is in terms of 16-bit, not bytes)

## 13. Conditional Instructions

---

### CNDST

**Format:** CNDST sourcePCND, dest, source, sourceAdjust.

**Construction:** CNDST (0|1|2|3), W0, S0, (0...2)

**Description:** Conditional Start

W0 is destination

```
sourceAdjust = 1 or 2
if ([[!]]sourcePCND|true|false] && S0 == 0)
{
    W0 = 0
    Pe = 1
}
else
{
    W0 = S0 + sourceAdjust
    Pe = 0
}
```

### CNDEF

**Format:** CNDEF sourcePCND, dest, source, sourceAdjust.

**Construction:** CNDEF (0|1|2|3), W0, S0, (0...2).

**Description:** Conditional ElseIf

W0 is destination

```
sourceAdjust = 0, 1 or 2
if (S0 == 0)
{
    W0 = S0 + sourceAdjust
    Pe = (sourceAdjust == 0)
}
elseif ([[!]]sourcePCND|true|false] && S0 == 1)
{
    W0 = 0
    Pe = 1
}
else
{
    W0 = S0
    Pe = 0
}
```

### CNDSTM

**Format:** CNDSTM sourcePCND, dest, source1, source2.

**Construction:** CNDSTM (0|1|2|3), W0, S0, S2.

**Description:** Conditional Set Mask

W0 is destination

```
if ([!]sourcePCND|true|false] && S0 == 0)
{
    W0 = S2
    Pe = (S2 == 0)
}
else
{
    W0 = S0
    Pe = (S0 == 0)
}
```

## CNDLT

**Format:** CNDLT *sourcePCND*, *dest*, *destPred*, *source*, *sourceAdjust*.

**Construction:** CNDLT (0|1|2|3), W0, P0, S0, (1...2).

**Description:** Conditional Loop Test

W0 is destination

```
sourceAdjust = 1 or 2
# if there are no running instances for which loop test passes, exit loop
if (or_reduce(all_instances([!]sourcePCND|true|false]) & all_instances(Pe)) == 0)
{
    P0 = 0
    if (S0 > sourceAdjust)
    {
        W0 = S0 - sourceAdjust
        Pe = 0
    }
    else
    {
        W0 = 0
        Pe = 1
    }
}
# if current instance is running and the test fails, increment mask count
elseif ([!]sourcePCND|true|false] && S0 == 0)
{
    P0 = 1
    W0 = sourceAdjust
    Pe = 0
}
# otherwise if current instance is running and the test fails or isn't
# running, leave the mask count alone and set Pe appropriately
else
{
    P0 = 1
    W0 = S0
    Pe = (S0 == 0)
}
```

## CNDEND

**Format:** CNDEND *dest*, *source*, *sourceAdjust*.

**Construction:** CNDEND W0, S0, (1...2).

**Description:** Conditional End.

W0 is destination.

```
sourceAdjust = 1 or 2
if (S0 > sourceAdjust)
```

```

{
    W0 = S0 - sourceAdjust
    Pe = 0
}
else
{
    W0 = 0
    Pe = 1
}

```

## CNDSETL

**Format:** CNDSETL.A dest, source1, source2

CNDSETL.B dest, source1, source2

**Construction:** CNDSETL.A W0, S0, S2

CNDSETL.B W0, S0, S2

**Description:** Conditional Set Link Address for Call Loop.

W0 is destination.

This instruction is split into two parts, A and B. Keep all arguments the same for both parts.

Part A does not update Dest or Pe, performing only internal operations.

PartB behaves as below:

```

find first instance where Pe = 1
link address = addr[first instance 's S2]
{
    # if call loop is skipped entirely or not processing chosen address
    # then leave mask count unchanged, otherwise mark for execution
    if S0 == 0 and current instance S2 /= first instance 's S2
    {
        W0 = 1
        Pe = 0
    }
    else
    {
        W0 = S0
        Pe = (S0 == 0)
    }
}

```

## CNDLPC

**Format:** CNDLPC dest, source

**Construction:** CNDLPC W0, S0

**Description:** Conditional Loop for Call

W0 is destination

```

{
    # if running, mark as done by setting mask count to 2
    if S0 = 0
    {
        W0 = 2
        Pe = 0
    }
    # else if this was not the chosen address, set mask count to 0 to

```



```
# enable execution on next iteration
elseif (S0 == 1)
{
    W0 = 0
    Pe = 1
}
# otherwise this instance skipped the call loop entirely so leave
# mask count unchanged
else
{
    W0 = S0
    Pe = 0
}
}
```

## 14. Data Access Instructions

### WDF

**Format:** WDF sourceDRC.

**Construction:** WDF (DRC0|DRC1)

**Description:** Wait until specified data fence is returned from memory sub-system.

**Table 59: Example:**

<pre>in vec2 textureCoordinate; out vec4 fragColor;  void main() {     fragColor = texelFetch(sampler,                           ivec2(gl_FragCoord.xy), 0); }</pre>	<pre>3    : smp2d.fcnorm.replace.integer drc0,     sh4,         r1, sh0, sh11, r0, 4;  4    : wdf drc0</pre>
--	--

### ITRSMP (ITRSMP1D ITRSMP2D ITRSMP3D)

**Format:** ITRSMP1D dest, sourcePerspectiveControl, sourceDRC, sourceCoeff, sourceTextureState, sourceSamplerState, chan, sourceCount, sourceWCoeff, sourceRasteriserState, sourceForcedSampleCount.

**Construction:** ITRSMP1D.mode{.SAT}{.PROJ}{.FCNORM}{.NNCOORDS}{.SCHEDSWAP|.SCHEDWDF}{.COMPARISON} (R0...R251|0){.F16}, (0...3), (DRC0|DRC1), (CF0...CF255), (SH0...SH255), (SH0...SH255), (1...4), (0...16), (CF0...CF255), (SH0...SH255), (SH0...SH255)

**Description:** Iterate coordinates from supplied coefficients and generate texture lookup request using those coefficients.

**Table 60: Example:**

<pre>void main() {     fragColor =     texture(sampler, textureCoordinate); }</pre>	<pre>0    : (ignorepe) {     itrsm2d.pixel.fcnorm.schedwdf     r0, 1, drc0, cf4,     sh4, sh0, 4, cf0, }</pre>
---	--

### SBO

**Format:** SBO.COEFF sourceBaseOffset

SBO.SHARED sourceBaseOffset

**Construction:** SBO.COEFF (0...255)

SBO.SHARED (0...255)

**Description:** Modify the base offset of shared or coefficient base addresses.

## DITR

**Format:** DITR dest, sourceDRC, sourceCoeff, sourceCount, sourcePerspectiveControl, sourceWCoeff, sourceRasteriserState, sourceForcedSampleCount.

**Construction:** DITR.(mode){.SAT}{.SCHEDSWAP|.SCHEDWDF} (R0...R251), (DRC0|DRC1), (CF0...CF255), (0...16), (0...3), (CF0...CF255)

**Description:** Iterate coordinates from supplied coefficients.

## 15. Appendix

### Special Constants

Index	Value	Definition	Type
0	0x00000000	0 (INT32) / 0.0 (Float)	SC
1	0x00000001	1 (INT32)	SC
2	0x00000002	2 (INT32)	SC
3	0x00000003	3 (INT32)	SC
4	0x00000004	4 (INT32)	SC
5	0x00000005	5 (INT32)	SC
6	0x00000006	6 (INT32)	SC
7	0x00000007	7 (INT32)	SC
8	0x00000008	8 (INT32)	SC
9	0x00000009	9 (INT32)	SC
10	0x0000000A	10 (INT32)	SC
11	0x0000000B	11 (INT32)	SC
12	0x0000000C	12 (INT32)	SC
13	0x0000000D	13 (INT32)	SC
14	0x0000000E	14 (INT32)	SC
15	0x0000000F	15 (INT32)	SC
16	0x00000010	16 (INT32)	SC
17	0x00000011	17 (INT32)	SC
18	0x00000012	18 (INT32)	SC
19	0x00000013	19 (INT32)	SC
20	0x00000014	20 (INT32)	SC
21	0x00000015	21 (INT32)	SC
22	0x00000016	22 (INT32)	SC
23	0x00000017	23 (INT32)	SC
24	0x00000018	24 (INT32)	SC
25	0x00000019	25 (INT32)	SC
26	0x0000001A	26 (INT32)	SC
27	0x0000001B	27 (INT32)	SC
28	0x0000001C	28 (INT32)	SC
29	0x0000001D	29 (INT32)	SC
30	0x0000001E	30 (INT32)	SC
31	0x0000001F	31 (INT32)	SC
64	0x3F800000	1.0f	SC

Index	Value	Definition	Type
65	0x40000000	float(21)	SC
66	0x40800000	float(22)	SC
67	0x41000000	float(23)	SC
68	0x41800000	float(24)	SC
69	0x42000000	float(25)	SC
70	0x42800000	float(26)	SC
71	0x43000000	float(27)	SC
72	0x43800000	float(28)	SC
73	0x44000000	float(29)	SC
74	0x44800000	float(210)	SC
75	0x3F000000	float(2-1)	SC
76	0x3E800000	float(2-2)	SC
77	0x3E000000	float(2-3)	SC
78	0x3D800000	float(2-4)	SC
79	0x3D000000	float(2-5)	SC
80	0x3C800000	float(2-6)	SC
81	0x3C000000	float(2-7)	SC
82	0x3B800000	float(2-8)	SC
83	0x3B000000	float(2-9)	SC
84	0x3A800000	float(2-10)	SC
85	0x3A000000	float(2-11)	SC
86	0x39800000	float(2-12)	SC
87	0x39000000	float(2-13)	SC
88	0x38800000	float(2-14)	SC
89	0x402DF854	e	SC
90	0x3EBC5AB2	1/e	SC
91	0x3FB504F3	Float SQRT(2)	SC
92	0x3F3504F3	Float 1/SQRT(2)	SC
93	0x3F490FDB	Float PI/4	SC
94	0x3FC90FDB	Float PI/2	SC
95	0x40490FDB	Float PI	SC
128	0x3EA2F983	Float 1/PI	SC
129	0x3F22F983	Float 2/PI	SC
130	0x3FA2F983	Float 4/PI	SC
131	0x40C90FDB	Float 2*PI	SC
132	0x41490FDB	Float 4*PI	SC
133	0x41C90FDB	Float 8*PI	SC
134	0x37800000	1.0f/65536f	SC

## 15. Appendix — Revision 1.0

Index	Value	Definition	Type
135	0x38000000	1.0f/32768f	SC
136	0x3B4D2E1C	0.0031308f	SC
137	0x414EB852	12.92f	SC
138	0x3ED55555	1.0f/2.4f	SC
139	0x3F870A3D	1.055f	SC
140	0x3D6147AE	0.055f	SC
141	0x80000000	-0.0f	SC
142	0x7F800000	Infinity	SC
143	0xFFFFFFFF		SC
144	0x7FFF7FFF		SC
145	0x3E9A209B	Log <sub>10</sub> (2)	SC
146	0x3F317218	Log <sub>e</sub> (2)	SC
147	0x0000007F	127 (INT32)	SC
148	0x7F7FFFFF	Max Float	SC
149	0x4B000000	2 <sup>23</sup>	SC
150	0x4B800000	2 <sup>24</sup>	SC
151	0x3F860A92	Pi/3	SC
152	0x3EAAAAAB	1/3	SC
153	0x3E2AAAAAB	1/6	SC
154	0x40549A78	Log <sub>2</sub> (10)	SC
155	0x3FB8AA3B	Log <sub>2</sub> (e)	SC
156	0x3D25AEE6	0.04045f	SC
157	0x3D9E8391	1.0f/12.92f	SC
158	0x3F72A76F	1.0f/1.055f	SC
159	0x4019999A	2.4f	SC

## 16. Contact Details

---

For further support, visit our forum:

<http://forum.imgtec.com>

Or file a ticket in our support system:

<https://pvrsupport.imgtec.com>

For general enquiries, please visit our website:

<http://imgtec.com/corporate/contactus.asp>