R600/R700/Evergreen

Assembly Language Format

1 Introduction

This document describes the shader assembly language for the R600-, R700-family, and Evergreen-family of devices. It is based on the microcode format that is defined in the ISA document. If you are not familiar with the shader architecture or the microcode format, first read the *ATI R700-Family Instruction Set Architecture* and the *ATI Evergreen-Family ISA Instructions and Microcode* documents.

2 Program Structure

A shader program consists of a control flow program and a set of clauses. The control flow program is the main program. The clauses are small blocks of instructions of a certain type, such as ALU, texture fetch, or vertex fetch instructions. Figure 1 shows a typical memory layout.

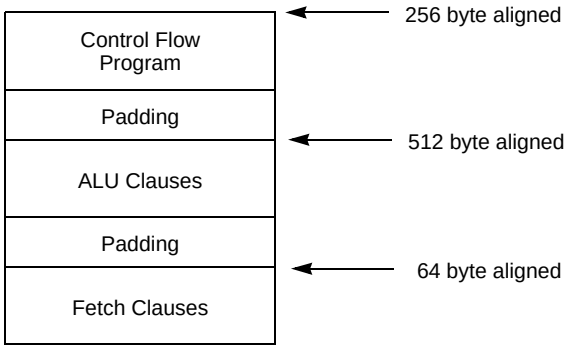

Figure 1 Typical Memory Layout

For the assembly language, the clauses are interleaved into the control flow program for enhanced readability. This improves the ability to follow the program flow and understand the dependencies.

Consider the following IL shader.

```
il_ps_2_0
dcl_cb cb0[2]
dcl_output_generic o0
mul o0, cb0[0], cb0[1]
end
```

For the shader program, the equivalent assembly is:

```
; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(4) KCACHE0(CB0:0-15)
      0  x: MUL         R0.x,  KC0[0].x,  KC0[1].x
         y: MUL         R0.y,  KC0[0].y,  KC0[1].y
      1  z: MUL         R0.z,  KC0[0].z,  KC0[1].z
         w: MUL         R0.w,  KC0[0].w,  KC0[1].w
01 EXP_DONE: PIX0, R0
END_OF_PROGRAM
```

The following sections detail the assembly syntax.

Comments in the assembly language are single lines starting with a comma. White space is ignored in the grammar.

Besides the instruction set architecture (ISA) code, an assembly program contains information used to set up the hardware control header. Certain directives to control the header must be placed before the ISA code; the remaining directives must be placed after.

3 Control Flow (CF) Program

The CF program constitutes the main program. It specifies how to execute each clause in the program; it also contains control flow statements (conditional jumps, looping, subroutines), memory allocation, and instructions to signal other blocks when VS/GS shaders have completed.

3.1 Clause Instructions

There are CF instructions to execute ALU, texture, or vfetch clauses. At minimum, these operations take a clause address and a count specifying the size of the clause.

The syntax form of these instructions typically is:

```
CFClauseInstruction
   ::= InstCount InstType ':' ClauseAddr ClauseCount ClausePropertiesOpt
```

The shader program in the previous section had an CF ALU instruction to execute an ALU clause at address 32 with a size of four 64-bit slots.

```
00 ALU: ADDR(32) CNT(4)
```

The address is a quadword address specified in the microcode; it is not a byte address.

If the ALU clause used constant waterfalling, there would be a clause property indicating this.

```
00 ALU: ADDR(32) CNT(4) USES_WATERFALL
```

Other examples of clause instructions are:

```
02 TEX: ADDR(415) CNT(1) BARRIER
02 VTX: ADDR(415) CNT(2)
02 ALU: ADDR(34) CNT(6) WHOLE_QUAD_MODE USES_WATERFALL
02 ALU: ADDR(34) CNT(6) KCACHE_MODE0(LOCK_0) KCACHE_BANK(0)
```

The END_OF_PROGRAM can be specified for all CF instructions except for ALU clause instructions. Unlike other CF instruction properties, the END_OF_PROGRAM property can be specified on the next line by itself.

```
02 TEX: ADDR(415) CNT(1) BARRIER
END_OF_PROGRAM
```

Here is an informal grammar for a clause instruction in the CF program.

```
CFClauseInstruction
    ::= InstCount InstType ':' ClauseAddress ClauseCount CFInstPropertiesOpt
InstCount           ::= Integer
InstType            ::= "ALU" | "ALU_PUSH_BEFORE"
                      | "ALU_POP_AFTER"
                      | "ALU_POP2_AFTER"
                      | "ALU_CONTINUE"
                      | "ALU_BREAK"
                      | "ALU_ELSE_AFTER"
                      | "TEX"
                      | "VTX"
                      | "VTX_TC"
ClauseAddress       ::= "ADDR(" Integer ")"
ClauseCount         ::= "CNT(" Integer ")"
CFInstPropertiesOpt ::= CFInstProperty*
CFInstProperty      ::= "USES_WATERFALL" | "WHOLE_QUAD_MODE" |
                              "NO_BARRIER" | “VARID_PIX” | ELEM_SIZE(int)
                            | BURSTCNT(int)
                          "END_OF_PROGRAM" | KCacheProperties
KCacheProperties    ::= "KCACHE0(" CB:start-end ")"
                      | "KCACHE1(" CB:start-end ")"
```

3.2 Export Instructions

Export instructions refer to external memory. There are two types of exports:

- those that refer to write-only memory, and
- those that refer to read/write memory.

The syntax of export instructions to write-only memory is:

```
CFExportInstruction
    ::= InstCount InstType ':' ExportTarget ',' SourceGPR CFInstPropertiesOpt
```

The normal export targets include Pixel, Position, and Parameter Cache. Here are some export examples with descriptions to the right.

```
01 EXP_DONE: POS0, R0   ; Last export to position with array_base = 60
02 EXP_DONE: POS1, R4   ; Last export to position with array_base = 61
01 EXP: PARAM0, R1      ; Export to parameter cache with array_base = 0
01 EXP: PIX0, R2        ; Export to pixel (RT) with array_base = 0
01 EXP_DONE: PIX1, R2   ; Last export to pixel (RT) with array_base = 1
```

If properties are specified, they are appended to the end of the instruction line.

```
01 EXP_DONE: PIX1, R2 WHOLE_QUAD_MODE BARRIER
01 EXP: PARAM1, R0 BARRIER
```

Here is an informal grammar for an export instruction in the CF program.

```
CFExportInstruction
   ::= InstCount InstType ':' ExportTarget ',' VecGPR CFInstPropertiesOpt
InstCount           ::= Integer
InstType            ::= "EXP" | "EXP_DONE"

ExportTarget        ::= Target ArrayBase
Target              ::= "PIX" | "POS" | "PARAM"
ArrayBase           ::= Integer
VecGPR              ::= "R" Integer FourCompSwizzleOpt?
FourCompSwizzleOpt  ::= '.' ['x'|'y'|'z'|'w']+
```

3.3 Mem Instructions

Mem instructions have two formats:

```
CF_MEM CMD : Dest, GPR_REG , PropListOpt
| CF_MEM CMD : GPR_REG , Source ',' PropListOpt
```

Where:

```
CF_MEM
MEM_SCRATCH | MEM_REDUCTION | MEM_RING | MEM_GLOBAL
| MEM_STREAM0 | MEM_STREAM1 | MEM_STREAM2 | MEM_STREAM3
```

CMD is one of:

```
_WRITE | _WRITE_IND | _WRITE_ACK | _WRITE_IND_ACK | _READ | _READ_IND
```

`Dest` or Source are:

```
VEC_PTR[int] | DWORD_PTR[int] | VEC_PTR[gpr + offset] | DWORD_PTR[gpr + offset]
```

The gpr can be indexed `R3[AL]`.

3.4 Miscellaneous CF Instructions

Miscellaneous CF instructions have the form:

```
CF_CMD  PropListOpt
```

Where the CF_CMD can be:

```
EMIT  |  CUT  +  EMIT_CUT  |  NOP  |  RET  |  PUSH  |  CALL_FS  |  POP
```

`POP` takes an additional argument: `int`.

3.5 Flow Control Instructions

Instead of showing a grammar, here are examples.

3.5.1 Looping Instructions

```
   ; Integer constant is 0, jump to CF inst 4 if condition fails.
01 LOOP AL, i0 FAIL_JUMP_ADDR(4)
02 ...
   ; Integer constant is 0, jump to CF inst 2 if condition passes.
03 ENDLOOP i0 PASS_JUMP_ADDR(2)
04 ... |
   ; Integer constant is 0, jump to CF inst 4 if condition fails.
01 REP i0 FAIL_JUMP_ADDR(4)
02 ...
   ; Integer constant is 0, jump to CF inst 2 if condition passes.
03 ENDREP i0 PASS_JUMP_ADDR(2)
04 ...
```

For the LOOP/REP instruction, the PASS_JUMP_ADDR option cannot be used; however, it can be used with ENDLOOP/ENDREP.

Loop opcode values are LOOP_NO_AL, LOOP_AL, ENDLOOP, and LOOP_DX10.

3.5.2 If/Else Instructions

If instructions on devices in the HD26XX through HD48XX series are in the form of an ALU instruction to evaluate the condition, and a sequence of PUSH/POPs to preserve the pixel state.

```
if ( R2.w == 0.0f )
{
   // if block
}
else
{
   // else block
}
```

```
03 PUSH             ; Push current pixel state for later.
04 ALU: ADDR(46) CNT(1)
   0 w: PRED_SETE ____, R2.w     UPDATE_EXEC_MASK UPDATE_PRED
06 ... ; if block
07 ELSE             ; Invert current pixel state.
08 ... ; else block
09 POP (1)          ; Preserve original pixel state.
```

The POP instruction has an argument that specifies the pop_count. The hardware allows a pop_count on most control flow instructions. To specify this, use the POP_CNT(%d) option.

```
03 PUSH POP_CNT(1)
07 ELSE POP_CNT(1)
09 POP POP_CNT(1)    ; Same as POP (1) where the option is implied due to
   opcode.
```

3.5.3 Break/Condition

Both the continue and break instruction take a jump address as its only argument.

```
06 BREAK 8       ; Break out to CF instruction 8.
06 CONTINUE 7    ; loop continue starting from CF instruction 7.
```

A typical scenario for using break is inside an if/else within a loop.

```
01 LOOP AL, i0 FAIL_JUMP_ADDR(10)
   02 ALU: ADDR(44) CNT(2)
      1  w: SETGE R2.w, (0.400000006f).x, R0.x
   03 PUSH
   04 ALU: ADDR(46) CNT(1)
      2  w: PRED_SETE ____, R2.w       UPDATE_EXEC_MASK UPDATE_PRED
   05 BREAK 9
   06 ELSE
   07 ALU: ADDR(47) CNT(4)
   3  x: ADD R0.x, R1.x, R0.x
      y: ADD R0.y, R1.y, R0.y
      z: ADD R0.z, R1.z, R0.z
      w: ADD R0.w, R1.w, R0.w
   08 POP (1)
09 ENDLOOP i0 PASS_JUMP_ADDR(2)
```

3.5.4 Call/Ret instruction

The `call` instruction takes a single jump address argument.

```
01 CALL 05       ; execute subroutine starting at 05.
02 ...
03 ...
04 ...
END_OF_PROGRAM
05 ...
06 ...
07 RET           ; return to calling address + 1
```

3.5.5 Jumps

Dynamic Jumps – Use the same format as miscellaneous opcodes with `JUMP` as the instruction and an additional argument with the jump destination: `CF_CONST(int)`.

Static Jumps – These require an additional two arguments: `CND(BOOL)` or `CMD(NOT_BOOL)`, followed by a Boolean register number.

4 ALU Clause

Devices in the HD26XX through HD48XX series are considered a scalar machines that execute up to five scalar operations per cycle. Instruction words are variable length. Each scalar operation is executed on a scalar functional unit.

The clauses are interleaved into the CF program, so the ALU clause instructions come immediately after the CF ALU instruction.

```
****************** change (0xfffff, float)
   00 ALU: ADDR(34) CNT(7)
      0  x: MUL    R0.x,  R0.x,  (1.0f).x
         y: MUL    R0.y,  R0.y,  (0.25f).y
         z: MUL    R0.z,  R0.z,  (0.5f).z
         w: MUL    R0.w,  R0.x,  (0.75f).w
         t: RECIPSQRT_IEEE R2.w, R0.w
```

4.1 Instruction Count

The instruction count is separate from the CF instruction count. Each group of scalar operations that make up an ALU instruction have a count. The following example contains two ALU instructions, with the count starting at 0.

```
0  x: MUL    R0.x,  R0.x,  R1.x
   y: MUL    R0.y,  R0.y,  R1.y
   z: MUL    R0.z,  R0.z,  R1.z
   w: MUL    R0.w,  R0.w,  R1.w
1  x: ADD    R0.x,  R0.x,  R1.x
   y: ADD    R0.y,  R0.y,  R1.y
   z: ADD    R0.z,  R0.z,  R1.z
   w: ADD    R0.w,  R0.w,  R1.w
```

4.2 Scalar Assignment

Each scalar operation in the ALU instruction is labeled with its scalar assignment, which can be assigned to x, y, z, w, or to the transcendental unit t. This information does not translate directly to any bits in the microcode; instead, it conveys information about how the scalar operation is assigned.

4.3 Opcode

All ALU opcode names in the disassembly format match the names used in the ISA document, except that unnecessary prefixes are stripped off. See Chapter 1 of the *AMD_Evergreen-Family_ISA_Instructions_and_Microcode*.

4.4 Output Modifiers

Output modifiers such as M2, M4, D2, and D4 are typically grouped with the instruction name. For example,

```
0  x: MUL*2  R0.x,  R0.x,  1.000000.x
   y: MUL/2  R0.y,  R0.y,  1.000000.x
```

modifiers include *2, *4, and /2.

4.5 Write Mask

When a write mask is not specified on a scalar operation, an underscore is used. The following implements a DOT3 using a DOT4 vector operation.

```
0  x: DOT4 ____,    R2.x,  R2.x
   y: DOT4 ____,    R2.y,  R2.y
   z: DOT4 ____,    R2.z,  R2.z
   w: DOT4 R1.w,    0.0f,  0.0f
```

4.6 Registers Types

r	general purpose register	{ R/W }
c	constant	{ R }
t	clause temp	{ R/W}
i	integer constant	{ R }
b	boolean constant	{ R }
KC0	constant cache	(R)
KC1	constant cache	(R)
CBn[m]	constant buffer	(R)

4.7 Relative Addressing

Relative addressing has a similar syntax to that of DX. Also, to maintain consistency with the DX language, we label the address register with A0 instead of AR, which is used frequently in hardware documentation.

Address Relative Example:

```
0  x: MUL    R0.x,  R0.x,  R2[A0.x].x
   y: MUL    R0.y,  R0.y,  R3[A0.y].x
```

Loop Relative Example:

```
0  x: MUL    R0.x,  R0.x,  R2[AL].x
   y: MUL    R0.y,  R0.y,  R3[AL].x
```

Valid indexes are AL, A0.x, A0.y, A0.z, A0.w, and Ga0.x (the last is used for R700-family shared register indexing).

4.8 Source Modifiers

The source modifiers negate and absolute value are specified as follows.

```
0  x: MUL    R0.x, |R0.x|,    R1.x      ; Absolute value
   y: MUL    R0.y, -|R0.y|,   R1.x      ; Absolute value and negate
```

4.9 Literals and Special Constants

There are two types of inlined floating point constants: literal constants, and special select constants. Literal constants consume extra slots per instruction. Special constants are selected through a special encoding of the source operand select field in the microcode.

Up to four 32-bit floating point literals can be stored per instruction. They are accessed by setting the source operand select to use the literal encoding, and setting the source channel. Literals in the assembly are enclosed in parenthesis and have a channel specified for them.

```
   (0x0d212, [-]float_comment).y     The hex value is used for the constant, the
floating comment can be used to document the value.
```

Float_comments can be any floating point number or `1.#QNANf` `1.#INFf` `1.#INDf` .

Special select constants are not enclosed in parenthesis and do not have a channel specification. Currently, 0.0f, 1.0f, 0.5f, and integer 1 are supported.

```
   0  x: MUL    R0.x, |R0.x|, 1.0f         ; Absolute value of R0.x
      y: MUL    R0.y, -|R0.y|, -|0.5f|     ; Absolute value and negate R0.y and
0.5f.
```

4.10 Previous Scalar and Previous Vector

If an instruction uses previous scalar or previous vector, then PS%d and PV%d syntax is used. The compiler uses the number to specify the ALU instruction where the previous vector or scalar was computed. While the assembler requires a number, it is ignored.

```
   2  x: ADD    R1.x,  R0.x,     C0.x
      y: ADD    R1.y,  R0.y,     C0.y 3
   3  x: MUL    R1.x,  R0.x,     C0.x
   4  x: MUL    R1.x,  PV3.x,    C0.x      ; PV.x came from instruction 3
      y: MUL    R1.y,  PV2.y,    C0.y      ; PV.y came from instruction 2
```

Its possible that PV can come from different ALU instruction if a previous instruction does not contain a scalar operation on a particular channel.

4.11 Properties

Properties such as clamp, and actions such as update_execute_mask and update_pred, can be specified along with each scalar operation. These properties usually are appended to the end of the instruction line.

```
   04 ALU: ADDR(67) CNT(3)
      10 w: CNDGE   R1.w,  -R1.w,    0.0f,         1.0f
      11 w: SETE    R4.w,  PV(10).w, -PV(10).w
      12 w: PRED_SETE ____, R4.w    UPDATE_EXEC_MASK UPDATE_PRED
```

Instruction 12 has action properties tagged to the end of the line.

ALU properties follow:

Bank_SWIZZLE can be any of:

```
VEC_012 | VEC_021 | VEC_120 | VEC_102 | VEC_201 | VEC_210
| SCL_210 | SCL_122 | SCL_212 | SCL_221
```

```
CLAMP | UPDATE_EXEC_MASK | UPDATE_PRED | FOG_MERGE
```

5 Tex Clause

The CF tex clause instruction specifies the clause address and count. This is followed immediately by the texture clause instructions.

```
   00 TEX: ADDR(214) CNT(1)
      0 SAMPLE R1, R1.xyxx, t0, s0 NORM(XYZW)
```

Currently, all texture instructions have the same form.

```
   TexInst ::= InstCount TexOpcode DestReg ',' SrcReg ',' ResourceId ','
SamplerId TexProperties
```

The special texture opcode GET_SAMPLE_INFO does not use a SrcReg.

5.1 Texture Opcode

```
GET_SAMPLE_INFO GET_COMP_TEX_LOD RESINFO_TEX LD SAMPLE PASS FETCH4 SAMPLE_L
SAMPLE_LB SET_CUBE_INDEX SAMPLE_LZ SAMPLE_G SAMPLE_G_L SAMPLE_G_LB SAMPLE_G_LZ
SAMPLE_C SAMPLE_C_L SAMPLE_C_LB SAMPLE_C_LZ SAMPLE_C_G SAMPLE_C_G_L
SAMPLE_C_G_LB SAMPLE_C_G_LZ SET_GRADIENTS_H SET_GRADIENTS_V SET_GRADIENTS_H
GET_GRADIENTS_V VTX_FETCH VTX_SEMANTIC
```

5.2 Destination Register

The destination register is a GPR, and a four component write mask is used.

5.3 Source Register

The source register is a GPR, and four component swizzle can be specified.

5.4 Resource Id

The Resource ID is specified as a t register, where the index specifies the id.

```
t5     ; Resource Id 5
```

5.5 Sampler Id

The sampler id is specified as an s register, where the index specifies the id.

```
s4     ; Sampler Id 4
```

5.6 Texture Properties

The texture properties are appended to the end of the line. The texture properties are:

```
WHOLE_QUAD       ; fetch whole quad
BC_FRAC_MODE
NORM(XYZW)       ; Treat XYZW as normalized coordinates
NORM(XY)         ; Treat XY as normalized coordinates
XOFFSET(3.5)     ; 3.1 signed fixed point
YOFFSET(1.5)
ZOFFSET(1.5)
LOD(1.25)        ; 3.4 signed fixed point
```

6 Vtx Clause

There are two kinds of fetches: semantic and non-semantic vertex fetches. The following program shows an example of a non-semantic fetch.

```
00 CALL 3
01 EXP_DONE: POS0, R0
02 EXP_DONE: PARAM0, R1
END_OF_PROGRAM
03 VTX: ADDR(26) CNT(2)
   0  VFETCH R0.xyz1, R0.x, f160 MEGAFETCH(11)
```

```
       FORMAT(0x39)
    1  VFETCH R1, R0.x, f160      MEGAFETCH(3)
       OFFSET(0xC) FORMAT(0x6)
04 RET
```

The syntax form for non-semantic fetch is.

```
VFetchInst ::= InstCount VtxOpcode Dest ',' SrcReg ',' BufferId VtxFetchProps
```

Otherwise the syntax form is.

```
VFetchInst ::= InstCount VtxOpcode SemanticId ',' SrcReg ',' BufferId
VtxFetchProps
```

Also, if a fetch constant is not used, the `BufferId` is left out.

6.1 Vtx Fetch Opcode

The opcode names are the same as the enumerations with the prefix stripped. There are only three: `VFETCH`, `VSEMATIC`, and `RESINFO_BUFFER`.

6.2 Dest and SemanticId

If the fetch is a non-semantic fetch, a destination GPR must be specified, and a four-component write mask can be specified that can include x, y, z, w, 0, and 1.

If a semantic fetch is used, “SemId”[0-9]+ is used in place of the destination GPR.

6.3 Source Register

This specifies the source GPR. Loop relative addressing can be used here.

```
R5[AL]
```

6.4 BufferId

The buffer id is specified as an f register with the number being the id. If this is not specified, `use_const_fields` = 0, and the data comes from the fetch properties.

6.5 Vertex Fetch Properties

The properties specify things such as whether the instruction uses a mega fetch, etc. If a fetch constant is not used, there are many more properties.

```
WHOLE_QUAD
MEGAFETCH(12)              ; Specifies that 12 bytes be fetched.
OFFSET(0xC)                ; Offset to 12 bytes
DX                         ; srf_mode_all SQ_SRC_MODE_ZERO_CLAMP_MINUS_ONE
OGL                        ; srf_mode_all = SQ_SRF_MODE_NO_ZERO
NUM_FORMAT_ALL(INT)        ; num_format_all = SQ_NUM_FORMAT_INT
NUM_FORMAT_ALL(SCALED)     ; num_format_all = SQ_NUM_FORMAT_SCALED
FORMAT_COMP_ALL(SIGNED)    ; format_comp_all = SQ_FORMAT_COMP_SIGNED
FORMAT_COMP_ALL(UNSIGNED_BIASED)
                           ; format_comp_all = SQ_FORMAT_COMP_UNSIGNED_BIASED
ENDIAN(8IN16)              ; endian_swap = SQ_ENDIAN_8IN16
```

```
ENDIAN(8IN32)              ; endian_swap = SQ_ENDIAN_8IN32
FETCHTYPE(INSTANCE_DATA)   ; fetch_type = SQ_VTX_FETCH_INSTANCE_DATA
FETCHTYPE(NO_INDEX_OFFSET) ; fetch_type = SQ_VTX_FETCH_NO_INDEX_OFFSET
FORMAT(0x32)               ; data_format = 0x32
}
```

7 Header Options That Must Precede the ISA Code

```
ShaderType = VALID_SHADER_TYPE
the il enum IL_SHADER_TYPE, list the possible values)
```

Valid shader are:

```
IL_SHADER_PIXEL = 1
IL_SHADER_COMPUTE = 3
```

```
TargetChip = Letter chip (p for HD2900, B for HD38XX)
```

Valid devices are:

p = HD2900
l = HD26xx/HD36xx
b = HD3850/HD3870
w = HD4850/HD4870
m = HD44xx/HD4650

Assignments to constant registers:

```
I2.x (set as int const) = INTEGER_LITERAL
I17.x (set to DX10 Const Buffer with FLT value) from cb1[9].y
I17.x (set to DX10 Const Buffer with INT value) from cb1[9].y
C3.x = type, src-reg src-component
```

8 Header Options That Must Follow the ISA Code

The semantic input to pixel shader is:

```
IN GPR = Usage Index V_REG Swiz L_DEFAULT_VAL PinPropListOpt
cbOutput<n>_enable            = INTEGER_LITERAL
ResourcesAffectAlphaOutput<n> = INTEGER_LITERAL
Field                         = INTEGER_LITERAL
```

Name	Function
Position	Usage
Pointsize	Usage
Color	Usage
Backcolor	Usage
Fog	Usage
Primcoord	Usage
Generic	Usage
Clipdistance	Usage

Name	Function
Culldistance	Usage
Primitiveid	Usage
Vertexid	Usage
Instanceid	Usage
Isfrontface	Usage
Lod	Usage
Coloring	Usage
node_coloring	Usage
Normal	Usage
rendertarget_array_index	Usage
viewport_array_index	Usage
sample_index	Usage
IL_pos	Usage
IL_pointsize	Usage
IL_color	Usage
IL_backcolor	Usage
IL_fog	Usage
IL_primcoordtype	Usage
IL_generic	Usage
IL_Unknown	Usage
NumIntrlFConstants	Before
NumIntrlIConstants	Before
NumIntrlBConstants	Before
NumClauseTemps	Before
SQ_PGM_RESOURCES:NUM_GPRS	After
CodeLen	After
SQ_PGM_END_CF	After
SQ_PGM_END_ALU	After
SQ_PGM_END_FETCH	After
SQ_PGM_RESOURCES:STACK_SIZE	After
SQ_PRM_RESOURCES:FETCH_CACHE_LINES	After
SQ_PRM_RESOURCES:PRIME_CACHE_ENABL	After
MaxScratchRegsNeeded	After
GprPoolSize	After
SQ_PGM_EXPORTS_PS:PS_EXPORT_MODE	After
SPI1:GEN_INDEX_PIX	After
SPI1:FIXED_PT_POSITION_ENA	After
SPI1:FIXED_PT_POSITION_ADDR	After
SPI1:FRONT_FACE_ENA	After

Name	Function
SPI1:FRONT_FACE_ADDR	After
SPI1:FRONT_FACE_CHAN	After
SPI1:FOG_ADDR	After
SPI1:GEN_INDEX_PIX_ADDR	After
TexCubeMaskBits	After
SPI0:NUM_INTERP	After
NumTexStages	After
SPI0:POSITION_ENA	After
SPI0:POSITION_CENTROID	After
SPI0:POSITION_ADDR	After
SPI0:PARAM_GEN	After
SPI0:PARAM_GEN_ADDR	After
SPI0:BARYC_SAMPLE_CNTL	After
SPI0:PERSP_GRADIENT_ENA	After
SPI0:LINEAR_GRADIENT_ENA	After
SPI0:POSITION_SAMPLE	After
SPI0:BARYC_SAMPLE_ENA	After
SPI:PROVIDE_Z_TO_SPI	After
DB:Z_EXPORT_ENABLE	After
DB:STENCIL_REF_EXPORT_ENABLE	After
DB:MASK_EXPORT_ENABLE	After
DB:ALPHA_TO_MASK_DISABLE	After
DB:Z_ORDER	After
DB:KILL_ENABLE	After
CB_SHADER_CONTROL:bitmap	After
MaxReductionBufferSize	After
SampleFreq	After
VGT_GS_OUT_PRIM_TYPE	After
MemExportVertexSize	After
MaxOutputVertexCount	After
UsesPrimId	After
StreamOutEnable	After
StreamOutDecls	After
VS_EXPORT_COUNT	After
VsOutSemanticMode	After
PA_CL_VS_OUT_CNTL	After
USE_VTX_POINT_SIZE	After
USE_VTX_EDGE_FLAG	After
USE_VTX_RENDER_TARGET_INDEX	After

Name	Function
USE_VTX_VIEWPORT_INDX	After
USE_VTX_KILL_FLAG	After
VS_OUT_MISC_VEC_ENA	After
VS_OUT_CCDIST0_VEC_ENA	After
VS_OUT_CCDIST1_VEC_ENA	After
MergeFlags	After
CLIP_DIST_ENA0	After
CLIP_DIST_ENA1	After
CLIP_DIST_ENA2	After
CLIP_DIST_ENA3	After
CLIP_DIST_ENA4	After
CLIP_DIST_ENA5	After
CLIP_DIST_ENA6	After
CLIP_DIST_ENA7	After
CULL_DIST_ENA0	After
CULL_DIST_ENA1	After
CULL_DIST_ENA2	After
CULL_DIST_ENA3	After
CULL_DIST_ENA4	After
CULL_DIST_ENA5	After
CULL_DIST_ENA6	After
CULL_DIST_ENA7	After
MemExportSize	After
GS_MODE	After

Normally, a geometry shader is followed by a copy shader.

An example input follows.

Source:

```
il_cs_2_0
dcl_cb cb0[1]
dcl_num_thread_per_group 128
dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
itof r0.z, vaTid.x
mul r0.y, r0.z, cb0[0].y
mod r0.x, r0.z, cb0[0].x
flr r0, r0
sample_resource(1)_sampler(0) g[vaTid.x], r0.xy
ret_dyn
end
```

API Call:

```
calclCompile...
calclLink...
calclDisassembleImage...
```

ISA:

```
ShaderType = 3
TargetChip = w
;SC Dep components
NumClauseTemps = 4

; --------  Disassembly --------------------
00 ALU: ADDR(32) CNT(19) KCACHE0(CB0:0-15)
      0  x: LSHL        R1.x,  R0.x,  (0x00000002, 2.802596929e-45f).x
         w: MOV         T0.w,  |KC0[0].x|
         t: I_TO_F      T0.y,  R0.x
      1  z: MOV         T0.z,  |PS0|
         w: MUL         ____,  PS0,  KC0[0].y
         t: RCP_e       ____,  |KC0[0].x|
      2  x: MUL         ____,  |T0.y|,  PS1
         y: FLOOR       R0.y,  PV1.w
      3  y: TRUNC       ____,  PV2.x
      4  x: MULADD      T0.x, -PV3.y,  T0.w,  T0.z
      5  y: ADD         ____, -|KC0[0].x|,  PV4.x
         w: SETGE       ____,  PV4.x,  |KC0[0].x|
      6  z: CNDE        T0.z,  PV5.w,  T0.x,  PV5.y
      7  x: ADD         ____,  |KC0[0].x|,  PV6.z
      8  y: CNDGT       R123.y, -T0.z,  PV7.x,  T0.z
      9  w: CNDGT       R123.w, -T0.y, -PV8.y,  PV8.y
     10  z: CNDE        R123.z,  KC0[0].x,  T0.y,  PV9.w
     11  x: FLOOR       R0.x,  PV10.z
01 TEX: ADDR(64) CNT(1)
     12  SAMPLE R0, R0.xyxx, t1, s0  UNNORM(XYZW)
02 MEM_EXPORT_WRITE_IND: DWORD_PTR[0+R1.x], R0, ELEM_SIZE(3)  VPM
END_OF_PROGRAM

; ----------------- CS Data -----------------------
; Input Semantic Mappings
;    No input mappings

GprPoolSize = 0
CodeLen                = 528;Bytes
PGM_END_CF             = 0; words(64 bit)
PGM_END_ALU            = 0; words(64 bit)
PGM_END_FETCH          = 0; words(64 bit)
MaxScratchRegsNeeded   = 0
; texResourceUsage[0]     = 0x00000000
; texResourceUsage[1]     = 0x00000000
; texResourceUsage[2]     = 0x00000000
; texResourceUsage[3]     = 0x00000000
; fetch4ResourceUsage[0]  = 0x00000000
; fetch4ResourceUsage[1]  = 0x00000000
; fetch4ResourceUsage[2]  = 0x00000000
; fetch4ResourceUsage[3]  = 0x00000000
; texSamplerUsage         = 0x00000000
; constBufUsage           = 0x00000000
ResourcesAffectAlphaOutput[0]  = 0x00000000
ResourcesAffectAlphaOutput[1]  = 0x00000000
ResourcesAffectAlphaOutput[2]  = 0x00000000
ResourcesAffectAlphaOutput[3]  = 0x00000000

;SQ_PGM_RESOURCES        = 0x30000002
SQ_PGM_RESOURCES:NUM_GPRS     = 2
SQ_PGM_RESOURCES:STACK_SIZE           = 0
SQ_PRM_RESOURCES:FETCH_CACHE_LINES    = 0
SQ_PRM_RESOURCES:PRIME_CACHE_ENABLE   = 1
CsSetupMode = Fast
NumThreadPerGroup = 128
NumWavefrontPerSIMD = 2
IsMaxNumWavePerSIMD = No
; SetBufferForNumGroup = false
```

Contact

Advanced Micro Devices, Inc.
One AMD Place
P.O. Box 3453
Sunnyvale, CA, 94088-3453
Phone: +1.408.749.4000

For Stream Computing:
URL: **www.amd.com/stream**
Questions: **streamcomputing@amd.com**
Developing: **ATI_Stream_SDK_Help_Request**
Forum: **www.amd.com/streamdevforum**