

# Vertex Shader Reference Manual

Version 2.8

Digital Media Professionals Inc.

**PROVISIONAL TRANSLATION**

**The content of this document is highly confidential  
and should be handled accordingly.**

### **Confidential**

These coded instructions, statements, and computer programs contain proprietary information of Nintendo and/or its licensed developers and are protected by national and international copyright laws. They may not be disclosed to third parties or copied or duplicated in any form, in whole or in part, without the prior written consent of Nintendo.

## Table of Contents

1	About This Document .....	16
2	Overview .....	17
3	Operating Environment .....	18
4	Main Objects and Reference Objects .....	19
5	How to Use the Assembler Tools .....	20
5.1	ctr_VertexShaderAssembler32 (Assembler) .....	20
5.1.1	Options .....	20
5.1.2	Example .....	21
5.2	ctr_VertexShaderLinker32 (Linker) .....	21
5.2.1	Input files .....	21
5.2.2	Options .....	21
5.2.3	Example .....	22
6	Vertex Shader Resources .....	23
6.1	Program RAM .....	23
6.2	Registers .....	23
6.2.1	Per-Register Resources .....	23
6.2.2	Precision of Floating-Point Registers .....	24
6.2.3	Input Registers .....	24
6.2.4	Temporary Registers .....	25
6.2.5	Floating-Point Constant Registers .....	25
6.2.6	Address Register .....	25
6.2.7	Boolean Registers .....	25
6.2.8	Integer Registers .....	25
6.2.9	Loop-Counter Register .....	25
6.2.10	Output Registers .....	26
6.2.11	Status Registers .....	26
7	Assembly Language Grammar Reference .....	27
7.1	Entering Assembly Instructions .....	27
7.1.1	Operation .....	27
7.1.2	Operand .....	27
7.1.3	Comment .....	27
7.2	Masking Output Components .....	27
7.3	Rearranging Input Components (Swizzling) .....	28

7.4	Adding a Negative Sign to Input Components.....	28
7.5	Using Register Index Offsets for Input Operands.....	29
7.6	Labels .....	29
7.6.1	The main Label .....	29
7.6.2	Label Name Collisions .....	30
7.7	Reserved Words .....	30
7.7.1	Register Names .....	30
7.7.2	Assembly Language Instructions.....	30
7.7.3	Preprocessor.....	30
8	Preprocessor Pseudo-Instructions .....	31
8.1	#include.....	31
8.2	#define .....	31
8.3	#undef.....	31
8.4	#ifdef, #ifndef, #if, #else, #elif, #endif .....	31
8.5	#error .....	32
8.6	#pragma.....	32
8.6.1	bind_symbol ( symbol_name , start_index [, end_index] ) .....	32
8.6.2	output_map ( data_name , mapped_register ) .....	34
8.7	#line .....	36
9	Assembly Language Instruction Reference .....	37
9.1	Define Instructions .....	37
9.1.1	def : Define Floating-Point Constants .....	37
9.1.2	defb : Define Boolean Constant .....	37
9.1.3	defi : Define Integer Constants .....	38
9.2	Arithmetic Instructions .....	38
9.2.1	add : Add .....	38
9.2.2	dp3: Three-Component Dot Product .....	39
9.2.3	dp4: Four-Component Dot Product .....	40
9.2.4	dph: Homogeneous Dot Product.....	40
9.2.5	dst: Distance Vector .....	41
9.2.6	exp: Exponential Base 2 .....	42
9.2.7	flr: Floor .....	42
9.2.8	litp: Light Coefficients.....	43
9.2.9	log: Logarithm Base 2 .....	43
9.2.10	mad: Multiply and Add .....	44
9.2.11	max: Maximum .....	44
9.2.12	min: Minimum.....	45
9.2.13	mov: Move.....	45
9.2.14	movs: Move to Address Register .....	46

9.2.15	mul: Multiply.....	46
9.2.16	nop: No Operation .....	47
9.2.17	rcp: Reciprocal .....	47
9.2.18	rsq: Reciprocal Square Root .....	48
9.2.19	sge: Set on Greater Than or Equal.....	48
9.2.20	slt: Set on Less Than.....	49
9.3	Macro Instructions .....	49
9.3.1	sub: Subtract .....	49
9.3.2	abs : Absolute .....	50
9.3.3	crs: Cross Product .....	51
9.3.4	frc: Fraction .....	51
9.3.5	lrp: Linear Interpolation .....	52
9.3.6	m3x2: 3x2 Multiply .....	53
9.3.7	m3x3: 3x3 Multiply .....	54
9.3.8	m3x4: 3x4 Multiply .....	55
9.3.9	m4x3: 4x3 Multiply .....	56
9.3.10	m4x4: 4x4 Multiply .....	57
9.3.11	nrm: Normalize.....	58
9.3.12	pow : Power.....	59
9.3.13	sgn: Sign.....	59
9.3.14	sincos: Sine and Cosine.....	60
9.4	Flow Control Instructions.....	61
9.4.1	call: Call Subroutine .....	62
9.4.2	callb: Boolean Call .....	62
9.4.3	callc: Condition Call .....	63
9.4.4	jpb: Boolean Jump .....	64
9.4.5	jpc: Condition Jump .....	65
9.4.6	ret: Return from Subroutine .....	66
9.4.7	ifb: Start if Block by Boolean .....	67
9.4.8	ifc: Start if Block by Condition.....	68
9.4.9	else: Start else Block.....	69
9.4.10	endif: End if Block.....	70
9.4.11	Loop: Start Loop Statement.....	70
9.4.12	endloop: End Loop Statement .....	71
9.4.13	breakc: Break from Loop Statement by Condition .....	72
9.4.14	cmp: Compare .....	73
9.4.15	end: End Process .....	75
10	Debug Build.....	76
11	Map Files.....	77
11.1	Overview .....	77

11.2	Loading Objects Order .....	77
11.3	Image Sizes .....	77
11.4	Program Code Information .....	77
11.5	Object Information .....	77
11.6	Swizzle Pattern Data .....	77
12	Precautions and Restrictions .....	78
12.1	Starting and Ending a Shader .....	78
12.2	Step Count .....	78
12.3	Pattern Counts for Swizzling and Masking .....	78
12.4	Control Instruction Limitations .....	79
12.5	Instructions That Cannot Be Called Consecutively .....	79
12.6	Registers That Cannot Be Used Simultaneously .....	80
12.7	Instruction Latency .....	80
12.7.1	Arithmetic and cmp Instruction Latency .....	82
12.7.2	Branch Instruction Latency .....	82
12.7.3	Output Order of Calculation Results .....	82
12.7.4	Stalls Due to Conflicts When Outputting Calculation Results .....	82
12.7.5	Stalls Due to Conflicts Among Arithmetic Units .....	83
12.7.6	Stalls Due to Instruction Dependencies .....	83
12.8	Results of Exceptional Operations .....	83
12.9	Limitations Related to Invalid Data Output .....	84
12.10	Shader Implementations That Cause Invalid Operations .....	84
12.10.1	Invalid Operation due to a <code>mov<sub>a</sub></code> Instruction .....	84
12.10.2	Invalid Operation due to a Specific Order of Instructions .....	86
13	Error Messages for the Assembler and Linker .....	89
13.1	Overview .....	89
13.2	Assembler Error Messages .....	89
13.2.1	80010001 .....	89
13.2.2	80010003 .....	89
13.2.3	80010004 .....	89
13.2.4	80010005 .....	89
13.2.5	80010006 .....	90
13.2.6	80010007 .....	90
13.2.7	80010008 .....	90
13.2.8	8001000b .....	90
13.2.9	80030001 .....	90
13.2.10	80030002 .....	90
13.2.11	80030003 .....	90

13.2.12	80030004	91
13.2.13	80030005	91
13.2.14	80030006	91
13.2.15	80030007	91
13.2.16	80030008	91
13.2.17	8003000b	91
13.2.18	8003000c	91
13.2.19	8003000d	92
13.2.20	8003000e	92
13.2.21	8003000f	92
13.2.22	80030010	92
13.2.23	80030011	92
13.2.24	80030012	92
13.2.25	80030013	92
13.2.26	80030014	93
13.2.27	80030015	93
13.2.28	80030017	93
13.2.29	80030018	93
13.2.30	80030019	93
13.2.31	8003001a	93
13.2.32	8003001b	93
13.2.33	8003001c	94
13.2.34	8003001d	94
13.2.35	8003001e	94
13.2.36	8003001f	94
13.2.37	80030020	94
13.2.38	80030021	94
13.2.39	80030022	94
13.2.40	80030023	95
13.2.41	80030024	95
13.2.42	80030025	95
13.2.43	8003002c	95
13.2.44	8003002d	95
13.2.45	80030033	95
13.2.46	80030034	95
13.2.47	80040001	96
13.2.48	80040005	96
13.2.49	80040007	96
13.2.50	80040009	96
13.2.51	8004000c	96
13.2.52	8004000d	96
13.2.53	8004000e	96

13.2.54	8004000f.....	96
13.2.55	80040010.....	97
13.2.56	80040011.....	97
13.2.57	80040012.....	97
13.2.58	80040015.....	97
13.2.59	80040016.....	97
13.2.60	80040017.....	97
13.2.61	8004001a.....	97
13.2.62	8004001b.....	97
13.2.63	8004001d.....	98
13.2.64	8004001f.....	98
13.2.65	80040021.....	98
13.2.66	80040022.....	98
13.2.67	80040023.....	98
13.2.68	80040024.....	98
13.2.69	80040025.....	98
13.2.70	80040026.....	98
13.2.71	80040027.....	99
13.2.72	8004002a.....	99
13.2.73	8004002b.....	99
13.2.74	8004002c.....	99
13.2.75	80040031.....	99
13.2.76	80040032.....	99
13.2.77	80040035.....	99
13.2.78	80040039.....	100
13.2.79	8004003a.....	100
13.2.80	8004003b.....	100
13.2.81	8004003c.....	100
13.2.82	8004003d.....	100
13.2.83	8004003e.....	100
13.2.84	8004003f.....	100
13.2.85	80040040.....	101
13.2.86	80040041.....	101
13.2.87	80050001.....	101
13.2.88	80050003.....	101
13.2.89	80050005.....	101
13.2.90	80050007.....	101
13.2.91	8005000a.....	101
13.2.92	8005000b.....	102
13.2.93	80060004.....	102
13.2.94	80060005.....	102
13.2.95	80060006.....	102



13.2.96	80060007 .....	102
13.2.97	80060009 .....	102
13.2.98	8006000b .....	102
13.2.99	8006000c .....	102
13.2.100	8006000d .....	103
13.2.101	8006000e .....	103
13.2.102	8006000f .....	103
13.2.103	80060010 .....	103
13.2.104	80060011 .....	103
13.2.105	80060012 .....	103
13.2.106	80060016 .....	103
13.2.107	80060017 .....	103
13.2.108	80060018 .....	104
13.2.109	80060019 .....	104
13.2.110	8006001b .....	104
13.2.111	8006001f .....	104
13.2.112	80060020 .....	104
13.2.113	80060021 .....	104
13.2.114	80060022 .....	104
13.2.115	80060023 .....	104
13.2.116	80060024 .....	104
13.2.117	80060025 .....	105
13.2.118	80060026 .....	105
13.2.119	80060028 .....	105
13.2.120	80060029 .....	105
13.2.121	8006002a .....	105
13.2.122	8006002c .....	105
13.2.123	8006002d .....	105
13.2.124	8006002e .....	105
13.2.125	8006002f .....	106
13.2.126	80060030 .....	106
13.2.127	80060031 .....	106
13.2.128	80060032 .....	106
13.2.129	80060033 .....	106
13.2.130	80060036 .....	106
13.2.131	80060037 .....	106
13.2.132	80060038 .....	106
13.2.133	8006003b .....	107
13.2.134	8006003c .....	107
13.2.135	8006003d .....	107
13.2.136	80060040 .....	107
13.2.137	40070001 .....	107

13.2.138	40070003.....	107
13.2.139	40070004.....	108
13.2.140	400a0001.....	108
13.2.141	400a0002.....	108
13.2.142	400a0003.....	108
13.2.143	400a0004.....	108
13.2.144	400a0005.....	108
13.3	Linker Error Messages.....	109
13.3.1	80080001.....	109
13.3.2	80080005.....	109
13.3.3	80080006.....	109
13.3.4	80080007.....	109
13.3.5	8008000f.....	109
13.3.6	80080012.....	109
13.3.7	80080014.....	109
13.3.8	80080015.....	110
13.3.9	8008001d.....	110
13.3.10	800800f.....	110
13.3.11	80080020.....	110
13.3.12	80080022.....	110
13.3.13	80080024.....	110
13.3.14	80080025.....	110
13.3.15	8008002a.....	111
13.3.16	8008002b.....	111
13.3.17	8008002c.....	111
13.3.18	8008002d.....	111
13.3.19	8008002e.....	111
13.3.20	8008002f.....	111
13.3.21	80080030.....	111
13.3.22	80080031.....	112
13.3.23	80080032.....	112
13.3.24	80080033.....	112
13.3.25	40090001.....	112
13.3.26	40090002.....	112
13.3.27	40090003.....	112
13.3.28	40090004.....	112
13.3.29	40090005.....	112
13.3.30	40090006.....	113
13.3.31	4009000.....	113
13.3.32	40090008.....	113
13.3.33	40090009.....	113
13.3.34	4009000a.....	113

13.3.35	4009000b .....	113
13.3.36	4009000c.....	113
14	File Format .....	114
14.1	Intermediate Object Files .....	114
14.1.1	Overview .....	114
14.1.2	File Header.....	114
14.1.3	Setup Information .....	117
14.1.4	Label Information.....	117
14.1.5	Program Code Information.....	118
14.1.6	Swizzle Data Information .....	118
14.1.7	Line Information.....	119
14.1.8	Relocation Information .....	119
14.1.9	Outmap Information .....	120
14.1.10	Bind Symbol Information .....	120
14.1.11	String Data .....	121
14.2	Executable Binary Files.....	121
14.2.1	Overview .....	121
14.2.2	Binary File Header.....	122
14.2.3	Package Information .....	122
14.2.4	Executable Image Information .....	124
15	Shader Checking Feature .....	127
15.1	Consistency Checker Feature.....	127
15.1.1	end Instruction Execution Check.....	127
15.1.2	Input Register Read Check .....	128
15.1.3	Output Register Write Check .....	128
15.2	Performance Checker Feature.....	128
15.2.1	Detectable Causes of Stalls.....	129
15.2.2	When There Are Multiple Stall Causes .....	130
15.2.3	Outputting the Results.....	131

## Code

Code 5-1 ctr_VertexShaderAssembler Example .....	21
Code 5-2 ctr_VertexShaderLinker Example .....	22
Code 7-1 Assembly Instruction Example .....	27
Code 7-2 Component Masking Example .....	27
Code 7-3 Swizzling Example 1 .....	28
Code 7-4 Swizzling Example 2 .....	28
Code 7-5 Negative Sign Example.....	28
Code 7-6 Index Offset Example.....	29

Code 7-7 Label Example .....	29
Code 8-1 #include Example .....	31
Code 8-2 #define Example .....	31
Code 8-3 #undef Example .....	31
Code 8-4 #ifdef, #endif Example .....	31
Code 8-5 #if, #endif Example .....	32
Code 8-6 #error Example .....	32
Code 8-7 bind_symbol Example, in Assembly Code and Application Code .....	33
Code 8-8 Running an Instruction After Writing to the Output Registers .....	34
Code 8-9 Writing to All Specified Registers .....	35
Code 8-10 Running Instructions Illegally After Writing to the Output Registers .....	35
Code 8-11 Writing to a Register Multiple Times .....	36
Code 8-12 Packing Multiple Attributes into a Single Output Register .....	36
Code 8-13 #line Example .....	36
Code 12-1 Pattern Count Example 1 .....	78
Code 12-2 Pattern Count Example 2 .....	79
Code 12-3 Instructions That Cannot Be Called Consecutively .....	79
Code 12-4 Registers That Cannot Be Used Simultaneously .....	80
Code 12-5 Output Order of Calculation Results .....	82
Code 12-6 Simultaneous Instruction Completion .....	82
Code 12-7 Register Use Causing a Stall .....	83
Code 12-8 Using Different Register Components Avoids a Stall .....	83
Code 12-9 Vertex Shader Causing Invalid Operations .....	84
Code 12-10 Vertex Shader Not Causing Invalid Operations .....	84
Code 12-11 Geometry Shader Causing Invalid Operations .....	85
Code 12-12 else-endif Clause Causing Invalid Operations .....	85
Code 12-13 call-ret Clause Causing Invalid Operations .....	85
Code 12-14 loop-endloop Clause Causing Invalid Operations .....	86
Code 12-15 mova Followed by a Branch Instruction Causing Invalid Operations .....	86
Code 12-16 Instruction Ordering that Causes Invalid Operations .....	87
Code 13-1 Example for Error 80060040 .....	107
Code 14-1 File Header Structure .....	115
Code 14-2 Setup Information Structure .....	117
Code 14-3 Label Information Structure .....	117
Code 14-4 Swizzle Data Information Structure .....	118
Code 14-5 Line Information Structure .....	119
Code 14-6 Relocation Information Structure .....	119
Code 14-7 Outmap Information Structure .....	120
Code 14-8 Bind Symbol Information Structure .....	121
Code 14-9 Binary File Header Structure .....	122
Code 14-10 Package Information Header Structure .....	123
Code 14-11 Executable Image Information Header Structure .....	125

Code 15-1 Dependency Stall Example.....	129
Code 15-2 Multiple Dependency Stall Example 1 .....	130
Code 15-3 Multiple Dependency Stall Example 2 .....	130
Code 15-4 Performance Checker Output .....	131

## Tables

Table 3-1 Operating Environment Specifications.....	18
Table 6-1 Register Types .....	23
Table 12-1 Instruction Latency.....	80
Table 14-1 File Header Fields.....	115
Table 14-2 Setup Information Fields .....	117
Table 14-3 Label Information Fields.....	118
Table 14-4 Swizzle Data Information Fields .....	118
Table 14-5 Line Information Fields.....	119
Table 14-6 Relocation Information Fields .....	119
Table 14-7 Outmap Information Fields.....	120
Table 14-8 Bind Symbol Information Fields .....	121
Table 14-9 Binary File Header Fields.....	122
Table 14-10 Package Information Header Fields .....	123
Table 14-11 Executable Image Information Header Fields.....	125

## Figures

Figure 2-1 Vertex Shader Overview.....	17
Figure 6-1 Floating-Point Number .....	24
Figure 14-1 Intermediate Object File Structure .....	114
Figure 14-2 Executable Binary File Structure .....	122
Figure 14-3 Package Information Structure.....	123
Figure 14-4 Executable Image Information Structure .....	124

## Revision History

Version	Revision Date	Description
2.8	2011/01/31	<ul style="list-style-type: none"> <li>• Made corrections in the code sample for loop instructions (section 7.3).</li> <li>• Revised the maximum number of swizzling and masking patterns for mad instructions from 64 to 32.</li> <li>• Made corrections in section 12.5 Instructions That Cannot Be Called Consecutively.</li> <li>• Added section 12.10 Shader Implementations That Cause Invalid Operations.</li> <li>• Added descriptions about swizzling.</li> </ul>
2.7	2010/11/17	<ul style="list-style-type: none"> <li>• Added text about the performance checker feature</li> <li>• Revised invalid error codes</li> </ul>
2.6	2010/11/05	<ul style="list-style-type: none"> <li>• Added Chapter 15 Shader Checking Feature.</li> </ul>
2.5	2010/09/30	<ul style="list-style-type: none"> <li>• Revised the reserved words for integer registers.</li> </ul>
2.4	2010/09/14	<ul style="list-style-type: none"> <li>• Revised information on the preprocessor pseudo-instruction <code>#pragma output_map</code>.</li> <li>• Added supplementary information to section 12.7.3 Output Order of Calculation Results.</li> <li>• Added supplementary information to section 12.7.5 Stalls Due To Arithmetic Unit Race Conditions.</li> <li>• Added section 12.7.6 Stalls Due To Instruction Dependencies.</li> </ul>
2.3	2010/08/20	<ul style="list-style-type: none"> <li>• Added support for UTF8 with a byte order mark as an assembler source format.</li> </ul>
2.2	2010/07/07	<ul style="list-style-type: none"> <li>• Deleted the TEG2 Limitations section.</li> </ul>
2.1	2010/06/04	<ul style="list-style-type: none"> <li>• Added Chapter 14 File Format.</li> <li>• Added preprocessor support for evaluating <code>#if</code> and <code>#elif</code>.</li> <li>• Added the preprocessor statement <code>#line</code>.</li> <li>• (English version only) Fixed typos, standardized terminology, and revised throughout for readability.</li> </ul>
2.0	2010/05/11	<ul style="list-style-type: none"> <li>• Added section 12.8 Results of Exceptional Operations.</li> <li>• Moved conditions for generating NaN output from section 12.9 Limitations Related to Invalid Data Output to section 12.8 Results of Exceptional Operations.</li> </ul>
1.9	2010/04/23	<ul style="list-style-type: none"> <li>• Added section 12.9 TEG2 Limitations.</li> <li>• Revised the format of the definition in section 8.5.1 <code>bind_symbol ( symbol_name , start_index [, end_index] )</code> to support an unspecified <code>end_index</code>.</li> <li>• Added information to Chapter 11 Map Files.</li> <li>• Added section 12.8 Limitations Related to Invalid Data Output.</li> </ul>
1.8	2010/04/02	<ul style="list-style-type: none"> <li>• Added description of limitations on writing to output registers to sections 8.6.2 <code>output_map ( data_name , mapped_register )</code> and 12.1 Starting and Ending a Shader.</li> </ul>
1.7	2010/03/12	<ul style="list-style-type: none"> <li>• Revised descriptions of the range of values for the address</li> </ul>

Version	Revision Date	Description
		<p>register.</p> <ul style="list-style-type: none"> <li>Added section 12.7.3 Output Order for Calculation Results.</li> <li>Added section 12.7.4 Stalls Due To Race Conditions When Outputting Calculation Results.</li> <li>Added section 12.7.5 Stalls Due To Arithmetic Unit Race Conditions.</li> <li>Added a specification that prohibits output registers from being overwritten.</li> <li>Revised the description of the <code>end</code> instruction.</li> <li>Added a restriction that requires input registers to be loaded.</li> </ul>
1.6	2010/02/15	<ul style="list-style-type: none"> <li>Added support for <code>#elif</code> and <code>#if defined</code>.</li> <li>Added support for comments delimited by <code>/*</code> and <code>*/</code>.</li> <li>Added information on specifications concerning file paths during debug builds.</li> </ul>
1.5	2009/12/25	<ul style="list-style-type: none"> <li>Fixed typos.</li> </ul>
1.4	2009/11/30	<ul style="list-style-type: none"> <li>Revised <code>bind_symbol</code> specifications for input registers.</li> <li>Removed <code>texture3</code> from the <code>output_map</code> settings.</li> <li>Revised error messages.</li> <li>Renamed tools.</li> </ul>
1.3	2009/10/30	<ul style="list-style-type: none"> <li>Explained a limitation that prohibits consecutive calls to the <code>movb</code> instruction.</li> <li>Added a note on jump limitations to the <code>ret</code> instruction from <code>jp</code>-related instructions.</li> <li>Revised an error in the description of the <code>abs</code> instruction.</li> </ul>
1.2	2009/09/10	<ul style="list-style-type: none"> <li>Explained limitations on the number of registers with <code>#pragma output_map</code>.</li> <li>Explained limitations of <code>call</code>-related instructions and control instructions.</li> <li>Explained limitations of the <code>loop</code> instruction.</li> <li>Removed output registers from the valid <code>dest</code> operands for the <code>crs</code> instruction.</li> </ul>
1.1	2009/06/25	<ul style="list-style-type: none"> <li>Added a note on the <code>end</code> instruction for vertex shaders.</li> </ul>
1.0	2009/04/30	<ul style="list-style-type: none"> <li>Initial version.</li> </ul>

# 1 About This Document

This documentation describes vertex shaders that run on the CTR-SDK.

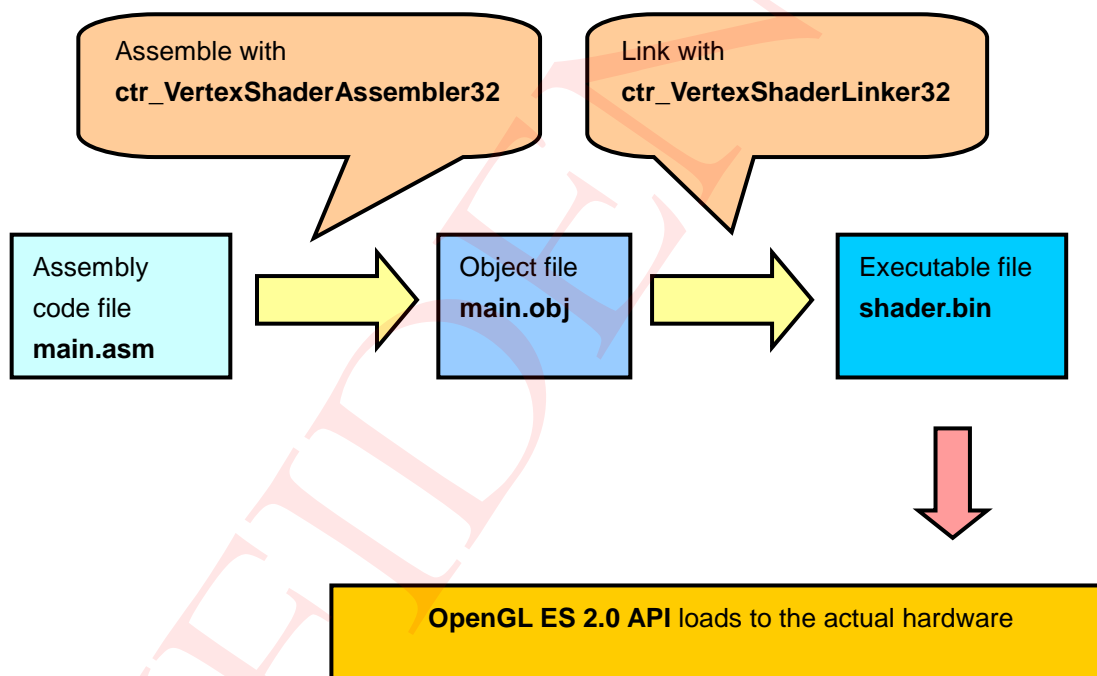


## 2 Overview

Vertex shaders are the only programmable shaders implemented by the CTR-SDK. Vertex shaders are written in PICA-specific assembly language. To generate executable files, the `ctr_VertexShaderAssembler32.exe` and `ctr_VertexShaderLinker32.exe` programs assemble and link vertex shaders written in this assembly language.

The OpenGL ES 2.0 API loads the executable files and then runs the shaders. The OpenGL ES 2.0 API can load only executable files that have been assembled and linked in this way. You cannot load shader files written in GLSL.

**Figure 2-1 Vertex Shader Overview**



### 3 Operating Environment

The `ctr_VertexShaderAssembler32.exe` and `ctr_VertexShaderLinker32.exe` programs have been confirmed to run in the following operating environment.

**Table 3-1 Operating Environment Specifications**

Component	System Requirements
CPU	Pentium 4 3.06 GHz
Memory	1 GB
OS	Windows XP
Development Environment	Microsoft Visual Studio .NET 2003

## 4 Main Objects and Reference Objects

Vertex shaders are run from the `main` function. A *main object* is an object file that can be assembled from an assembly code file that has a `main` function. A *reference object* is an object file that can be assembled from an assembly code file that does not have a `main` function.

To put a `main` function in an assembly code file, set the `main` label at the location where shader execution starts and set the `endmain` label after the last instruction at the end of the `main` function (this is the last instruction in the `main` function, not the last instruction in the assembly code file).

A reference object only has subroutines and is referenced by a main object to resolve unresolved labels. When the objects are linked to create an executable file, the executable file will not include reference objects that are not referenced by any main objects.

If multiple main and reference objects are linked, a single executable file is generated that includes more than one main object. Specify the executable file with the `glShaderBinary` function to load it. The number of shader objects you specify to the `glShaderBinary` function at that time specifies the number of main objects to load.

Main objects do not reference each other for unresolved labels when they are linked. Also note that if multiple reference objects with the same label name are specified as link targets, an error will occur while linking.

You cannot link only reference objects to create an executable file; you must link at least one main object.

## 5 How to Use the Assembler Tools

### 5.1 ctr\_VertexShaderAssembler32 (Assembler)

`ctr_VertexShaderAssembler32.exe` assembles assembly code files written in a PICA-specific assembly language and outputs object files.

The assembler is run from the command prompt as follows.

```
ctr_VertexShaderAssembler32 <input filename> [options]
```

Specify the input assembly code file in `<input filename>`. You can specify the options in section 5.1.1 for `[options]`.

This command has the following characteristics and requirements.

- An input file must be specified.
- The options can be omitted.
- Help is displayed when no arguments are given.
- The Shift-JIS and UTF-8 (with a byte order mark) file encodings are supported.
- The CR+LF newline code is supported.
- Use a filename that does not contain any spaces, is composed of no more than 128 single-byte alphanumeric characters, and does not use the symbols `\ / : * ? " < > |`.

#### 5.1.1 Options

- `-O <filename>`

Specifies the output filename (this is `<input filename>.obj` if it is left unspecified).

- `-I <file path>`

Specifies the input file path. You can use this option to specify the path to both the assembly code file and the files that it includes. First the current directory and then the directory specified by this option are searched for the input file and include files.

- `-D <key> [= <value>]`

Defines a macro. Specify the macro name with `<key>` and its value with `<value>`. The `<value>` can be omitted.

- `-debug`

Generates an object file with debugging information. When this option is not specified, the output object file does not include the full input file path.

- `-nowarning`

Suppresses warning message output.

- `-?` or `-help`

Displays Help.

---

### 5.1.2 Example

#### Code 5-1 ctr\_VertexShaderAssembler Example

```
ctr_VertexShaderAssembler32 main0.asm -IC:/sample/src -IC:/sample/inc -DDEBUG=1
```

In this example, `main0.asm` is the name of the input file; `C:/sample/src` and `C:/sample/inc` configure the input file paths. The `DEBUG` macro is also defined with a value of 1.

When this input file is successfully assembled, the file `main0.obj` is generated.

---

## 5.2 ctr\_VertexShaderLinker32 (Linker)

`ctr_VertexShaderLinker32.exe` links object files output by `ctr_VertexShaderAssembler32.exe` and then outputs an executable file.

The linker is run from the command prompt as follows.

```
ctr_VertexShaderLinker32 <input files> [options]
```

Specify the input object files in `<input files>`. The input files must be object files that together include at least one `main` function. You can specify (or omit) the options in section 5.2.2 for `[options]`. Help is displayed if no arguments are given. Use filenames that do not contain any spaces, are composed of no more than 128 -single-byte alphanumeric characters, and do not use the symbols `\ / : * ? " < > |`.

If there is more than one main object in the input files, the application must specify the same number of shader objects to the `glShaderBinary` function. At that time, each shader object references a main object in the order the main objects were specified in the argument to `ctr_VertexShaderLinker32.exe`.

---

### 5.2.1 Input files

- `<input filenames> [input files]`

The input files must be object files that together include at least one `main` function.

---

### 5.2.2 Options

- `-I <file path>`

Specifies the input file path. First the current directory and then the directory specified by this option are searched for the input files.

- `-O <filename>`

Specifies the name of the output file. This is `shader.bin` by default.

- -M

Outputs a map file. This file has the same name as the executable file, but uses the .map extension. For details on map files, see Chapter 11 Map Files.

- -debug

Links all linked object files with debugging information.

- -nodebug

Links all linked object files without debugging information.

- -check\_consistency

Performs a consistency check of all linked main objects. See section 15.1 Consistency Checker Feature for details on the consistency check.

- - check\_performance

Conducts a performance check on all linked main objects. For details, see section 15.2 Performance Checker Feature.

- -? or -help

Displays Help.

### 5.2.3 Example

#### Code 5-2 ctr\_VertexShaderLinker Example

```
ctr_VertexShaderLinker32 main0.obj main1.obj subr0.obj subr1.obj -Oshader0.bin -M
```

This example links the main objects main0.obj and main1.obj, and the reference objects subr0.obj and subr1.obj.

## 6 Vertex Shader Resources

Vertex shaders have the following resources.

### 6.1 Program RAM

Program RAM is the region that stores assembly language instruction codes. It can store 512 instructions. If an assembly code file uses more than 512 assembly instructions, it will cause an error when it is assembled or linked.

### 6.2 Registers

#### 6.2.1 Per-Register Resources

The following register types are used for calculations and flow control.

**Table 6-1 Register Types**

Name	ID	Components	Number	R/W	Index	Bit Width
Input registers	v#	4	12	R	-	24
Temporary registers	r#	4	16	RW	-	24
Floating-point constant registers	c#	4	96	R	a0/aL	24
Address register	a0	2	1	RW	-	8
Boolean registers	b#	1	16*	R	-	1
Integer registers	i#	1	4	R	-	24
Loop-counter register	aL	1	1	R	-	8
Output registers	o#	4	16	W	-	24
Status registers	-	1	2	RW	-	1

- ID

This identifier is used when entering assembly instructions. "#" indicates the register number. Specify a number from 0 to (number of registers - 1).

- **Name**  
This is the name of the register.
- **Number**  
This is the number of registers.
- **Components**  
This is the amount of data in a single register. When there are four components, a single register contains the x, y, z, and w components.
- **R/W**  
This indicates whether reads and writes are allowed. "R" indicates that a register can be specified only as an input operand. "W" indicates that a register can be specified only as an output operand. "RW" indicates that a register can be specified as either an input or output operand.
- **Index**  
It is possible to specify the register numbers of these registers using the content of other registers in Table 6-1. See section 7.5 Using Register Index Offsets for Input Operands.
- **Bit Width**  
This indicates the bit width of each register.

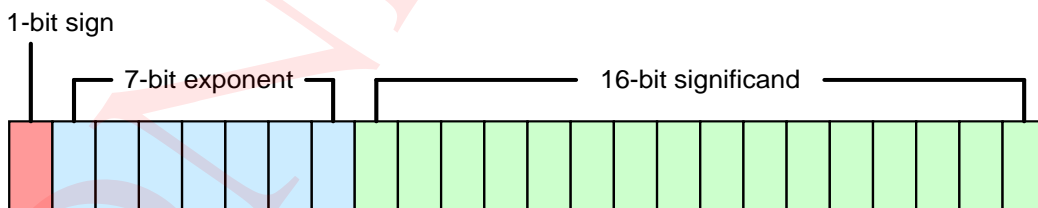
## 6.2.2 Precision of Floating-Point Registers

The input registers, temporary registers, and floating-point constant registers are all floating-point number registers. Floating-point numbers use 1 sign bit, 7 exponent bits, and 16 significand bits for a total of 24 bits. A sign bit of 0 is positive and 1 is negative. The exponent bits are in base 2 and have a bias of 63. The significand bits represent a value that is one less than the actual significand.

The actual value of a floating-point number is:

$$(-1)^{(\text{sign})} \times 2^{(\text{exponent}-63)} \times (1 + \text{significand bits})$$

**Figure 6-1 Floating-Point Number**



## 6.2.3 Input Registers

Input registers are floating-point registers that store vertex attribute data ("attributes" in OpenGL ES 2.0 applications).



### 6.2.4 Temporary Registers

---

Temporary registers are floating-point registers that temporarily maintain calculation results to be reused later. Their values are preserved until they are overwritten.

### 6.2.5 Floating-Point Constant Registers

---

Floating-point constant registers are floating-point registers that store constants to use for calculations. Uniforms for OpenGL ES 2.0 applications are stored here.

### 6.2.6 Address Register

---

An address register is an integer register that accepts values in the range [-95, 95]. If you assign the value from a floating-point register to the address register, only the integer part is assigned. Behavior is undefined if you assign values that are not in the range [-95, 95].

You can use this register's value to specify another register's number. See section 7.5 Using Register Index Offsets for Input Operands.

### 6.2.7 Boolean Registers

---

Boolean registers hold Boolean values. They are used for branches and jumps.

Uniforms for OpenGL ES 2.0 applications are also stored here. Boolean register 15 (b15) is reserved for geometry shaders.

### 6.2.8 Integer Registers

---

Integer registers hold integer values and are used to control `loop` instructions.

These registers store loop counts, the initial values for the loop-counter register, and the amounts by which to increment the loop-counter register. They are 24 bits wide: bits 0-7 specify the loop count; bits 8-15 specify the initial value for the loop-counter register; and bits 16-23 specify the amount by which to increment the loop-counter register.

When control enters a `loop` instruction, the loop-counter register is first initialized with its initial value from one of the integer registers. Then the assembly code instructions from the `loop` to the `endloop` instructions are run repeatedly. The number of loop iterations is the loop count plus one (meaning that the instructions are run only once when the loop count in this register is 0). At each iteration of the loop, the loop-counter register is incremented by the amount given by this register.

Uniforms for OpenGL ES 2.0 applications are also stored here.

### 6.2.9 Loop-Counter Register

---

This register stores the counter value for loop instructions. Its value is in the range [0, 255].

You can use the value of this register to specify another register's number, just as you can with the address register. See section 7.5 Using Register Index Offsets for Input Operands.

### 6.2.10 Output Registers

---

These registers output data that has been processed by vertex shaders into a later stage of the graphics pipeline.

### 6.2.11 Status Registers

---

These registers have their values set by comparison instructions and are used for branch conditions.

## 7 Assembly Language Grammar Reference

### 7.1 Entering Assembly Instructions

Assembly instructions are entered in the following format. The comment can be omitted.

```
<Operation> [Operand] [Comment]
```

#### Code 7-1 Assembly Instruction Example

```
add    r0, r1, r2    // r0 = r1 + r2
mul    r3, c0, v0    // r3 = c0 * v0
```

Assembly instructions are entered using ASCII characters, in compliance with the following rules.

#### 7.1.1 Operation

For the operation, denote the assembly instruction to run. You can only code one operation per line.

#### 7.1.2 Operand

```
Operand name[, Operand]
```

Denote the name of the operand, such as the register or the direct value targeted by the operation. The operation is followed by at least one space or tab before the operand name. Use commas to delimit multiple operands. You can denote one or more spaces or tabs between operands.

#### 7.1.3 Comment

Denoting two forward-slashes ("/") causes the rest of the line to be treated as a comment. Text between the delimiters /\* and \*/ is also treated as a comment.

### 7.2 Masking Output Components

When you output calculation results to registers that have more than one component, you can specify which components to output. Values are updated only for the specified components. If nothing is specified, all components are updated. Specify the components in x, y, z, w order (you cannot, for example, denote "wzyx").

#### Code 7-2 Component Masking Example

```
add    r0.x, r1, r2    // The x component is updated.
                        // The y, z, and w components are not updated.
mov    r0.yz, r1       // The y and z components are updated.
                        // The x and w components are not updated.
dp3    r0, r1, r2      // The x, y, z, and w components are all updated.
```

## 7.3 Rearranging Input Components (Swizzling)

When using registers with multiple components as input operands, you can rearrange (swizzle) the components that are used by calculations. If nothing is specified, components are used in x, y, z, w order. If you do not specify all four components, the component specified last is repeated.

### Code 7-3 Swizzling Example 1

```
add    r0, r1.xzyy, r2    // r0.x = r1.x + r2.x
                        // r0.y = r1.z + r2.y
                        // r0.z = r1.y + r2.z
                        // r0.w = r1.y + r2.w
mov     r0, r1.ywz        // r0.x = r1.y
                        // r0.y = r1.w
                        // r0.z = r1.z
                        // r0.w = r1.z ; The z-component, specified last,
                        // is repeated
add     r0, r1.zw, r2.xy  // r0.x = r1.z + r2.x
                        // r0.y = r1.w + r2.y
                        // r0.z = r1.w + r2.y
                        // r0.w = r1.w + r2.y
```

Note that input component swizzling and output component masking are interpreted differently when you do not specify all four components. The first, second, third, and fourth input components after swizzling are applied respectively to the x, y, z, and w components specified for output component masking.

### Code 7-4 Swizzling Example 2

```
mov     r0.x, r1.xy       // r0.x = r1.x ; (r1.xyyy)
mov     r0.y, r1.xy       // r0.y = r1.y ; (r1.xyyy)
mov     r0.z, r1.xy       // r0.z = r1.y ; (r1.xyyy)
mov     r0.w, r1.xy       // r0.w = r1.y ; (r1.xyyy)
add     r0.zw, r1.yx, r2.wz // r0.z = r1.x + r2.z; (r1.yxxx + r2.wzzz)
                        // r0.w = r1.x + r2.z; (r1.yxxx + r2.wzzz)
```

As shown above, the input component marked in red is used for each output component.

## 7.4 Adding a Negative Sign to Input Components

You can prefix input operands with a negative sign.

### Code 7-5 Negative Sign Example

```
add     r0, r1, -r2       // r0 = r1 - r2
mul     r0, -r1, r2       // r0 = (-r1) * r2
```

## 7.5 Using Register Index Offsets for Input Operands

You can offset the register number of an input operand by entering it in brackets ("[]"). You can enter the sum of multiple integers in the brackets. If this integer sum (indicating the offset) exceeds the number of registers, an error occurs during assembly.

The floating-point constant registers, and only these registers, can also be offset by the values of the address register and/or the loop-counter register. If you use the address register, you must specify either its x or y component. Behavior is undefined when the total index offset, including the address register and loop-counter register, is not within the range [0, 95]

### Code 7-6 Index Offset Example

```
c2           // The index is 2
v[5 + 2]     // The index is 5 + 2
r8[1 + 2 + 4] // The index is 8 + 1 + 2 + 4
c[3 + a0.x]  // The index is 3 + a0.x
c4[11 + aL]  // The index is 4 + 11 + aL
```

## 7.6 Labels

Labels are specified as jump targets for the `call` and `jmpb` instructions.

To encode a label, add a colon (":") after a name that is a combination of single-byte alphanumeric characters and underscores ("\_"). Labels cannot include a decimal or hexadecimal number (such as 123 or 0xf), a register name (such as r0 or c0), or any other reserved word. A line encoding a label cannot have any other notation except a comment.

### Code 7-7 Label Example

```
function0:
    mov    r0, r1
    ...
    ret
main:
    ...
    call   function0
```

### 7.6.1 The main Label

A vertex shader always begins executing at the `main` label. A shader without a `main` label is referenced as a subroutine. When vertex shader assembly code files are assembled and linked, at least one of the linked objects must be a main object that contains at least one `main` label. In the same way, main objects must also set at least one `endmain` label. Set the `endmain` label immediately after the last instruction to be run (this is the instruction that does the final write to the output registers).

---

## 7.6.2 Label Name Collisions

---

You cannot use the same label name more than once in a single object file. Likewise, when linking multiple object files, you cannot use the same label name more than once in object files used for multiple subroutines. Main objects do not reference each other, so the same label name can be used within more than one main object.

---

## 7.7 Reserved Words

---

The assembler defines the following strings to be reserved words. Do not use reserved words for labels, symbols, or other names.

---

### 7.7.1 Register Names

---

v0–v15, r0–r15, c0–c95, a0, b0–b15, i0–i3, aL, o0–o15

---

### 7.7.2 Assembly Language Instructions

---

def, defb, defi, add, dp3, dp4, dph, dst, exp, flr, litp, log, mad, max, min, mov, mova, mul, nop, rcp, rsq, sge, slt, sub, abs, crs, frc, lrp, m3x2, m3x3, m3x4, m4x3, m4x4, nrm, pow, sgn, sincos, call, callc, callb, jpb, jpc, ret, ifb, ifc, else, endif, loop, endloop, breakc, cmp, end

---

### 7.7.3 Preprocessor

---

include, define, undef, ifdef, ifndef, if, defined, else, elif, endif, error, pragma, bind\_symbol, output\_map

## 8 Preprocessor Pseudo-Instructions

### 8.1 #include

---

Use this to insert a file. Place the file in double quotes.

#### Code 8-1 #include Example

```
#include "defs.asm"
```

### 8.2 #define

---

This defines a macro. A macro is denoted using no more than 128 single-byte alphanumeric characters and underscores ("\_") and cannot begin with a number. Behavior is undefined if more than 128 characters are used.

#### Code 8-2 #define Example

```
#define MAX_COUNT 100
```

### 8.3 #undef

---

This deletes a macro defined by #define.

#### Code 8-3 #undef Example

```
#undef MAX_COUNT
```

### 8.4 #ifdef, #ifndef, #if, #else, #elif, #endif

---

These are used for conditional compilation with macros.

#### Code 8-4 #ifdef, endif Example

```
#define USE_FUNCTION_A
#ifdef USE_FUNCTION_A
FunctionA:
    ...
#endif
```

Evaluation of the following #if and #elif expressions is also supported.

- Literal values (that can be expressed as signed 32-bit integers; behavior is undefined for all other values)
- Minus signs on literal values (addition and subtraction are not supported)
- `defined(macro)`
- `!defined(macro)`

- Equality signs (`==`, `!=`)
- Inequality signs (`<`, `<=`, `>`, `>=`)
- Bitwise AND (`&&`) and OR (`||`)
- Any combination of the above

**Code 8-5 #if, #endif Example**

```
#if (defined(AAA) && (BBB == 1))
...
#endif
```

**Note:** An error is generated when an expression uses an undefined macro.

## 8.5 #error

This outputs an error.

**Code 8-6 #error Example**

```
#error "error message"
```

## 8.6 #pragma

The `pragma` instruction configures extended information for the assembler.

### 8.6.1 bind\_symbol ( symbol\_name , start\_index [, end\_index] )

This binds a symbol name to a register. Specify any symbol name in `symbol_name` and specify the starting and ending positions of registers to bind in `start_index` and `end_index`, respectively. You can bind to input registers, floating-point constant registers, integer registers, or Boolean registers.

If you use a `#pragma bind_symbol(symbol_name, start_index)` definition without specifying `end_index`, `start_index` specifies both the starting and ending positions of the registers to bind and thereby specifies a single register.

When an application loads the executable file for the shader assembly code that defines `bind_symbol`, it can use the defined symbol names as arguments to `glGetAttribLocation`, `glBindAttribLocation`, and other functions, and thereby configure the input registers. In the same way, these can also be used as arguments to `glGetUniformLocation` and other functions to configure the floating-point constant registers, integer registers, and Boolean registers.

The `def`, `defb`, and `defi` instructions cannot be used to define a constant value for any registers already specified by `bind_symbol`, either within the same shader assembly code file or within any objects referenced at link time.

You can specify and configure individual components of floating-point constant registers when configuring those registers. To do so, specify the components in the format `".xyzw"` after the name in `symbol_name`. Specify only consecutive components in `xyzw` order (although you can configure



subsets of components such as "xy", "yzw", and "zw", you cannot configure nonconsecutive subsets such as "xz", "yw", or "xyw").

With input registers, you cannot set more than one symbol name for a single input register. Also, the same value must be used for `start_index` and `end_index`.

**Note:** The input register settings determine which vertex attributes are used as inputs to the graphics pipeline. In other words, an input register that does not have a symbol bound to it by these settings is not recognized as an input vertex attribute and thus stores undefined values.

When you specify individual components of an input register, the number of components only affects the value of `type` that is obtained by the `glGetActiveAttrib` function. If you set `size` equal to 1, 2, or 3 in the `glVertexAttribPointer` function, vertex attribute data is loaded respectively into either the `x`, `xy`, or `xyz` components of the corresponding register. The default value is loaded into any component into which vertex attribute data is not loaded. By default, the value for `y` is 0, `z` is 0, and `w` is 1.

**Note:** The hardware does not operate properly when the vertex shaders have not read any input registers. Reading even a single component of any input register during a single cycle of vertex processing is enough to fulfill this requirement. It is acceptable for this input register to have undefined content.

#### Code 8-7 bind\_symbol Example, in Assembly Code and Application Code

```
// Assembly language source code
#pragma bind_symbol ( ModelViewMatrix , c0 , c3 )
#pragma bind_symbol ( Position , v0 )
#pragma bind_symbol ( LoopCounter0 , i1 , i1 )
#pragma bind_symbol ( bFirst , b2 , b2 )
#pragma bind_symbol ( Scalar.x , c4 , c4 ) // c4.x is assigned to the Scalar
symbol

// Application source code
glBindAttribLocation ( program , 0 , "Position" );
glEnableVertexAttribArray ( 0 );

uniform_location = glGetUniformLocation ( program , "ModelViewMatrix" );
GLfloat matrix[4][4];
glUniform4fv ( uniform_location , 4 , matrix );

GLfloat scalar_value;
uniform_location = glGetUniformLocation ( program , "Scalar" );
glUniform1f ( uniform_location , scalar_value );

uniform_location = glGetUniformLocation ( program , "bFirst" );
glUniform1i ( uniform_location , GL_TRUE );
```

```
GLint loop_setting[3] = { 4 , 0 , 1 } ;           // loop_count-1 , init , step
uniform_location = glGetUniformLocation ( program , "LoopCounter0" );
glUniform3iv ( uniform_location , loop_setting );
```

## 8.6.2 output\_map ( data\_name , mapped\_register )

This configures the attributes of the data output by the vertex shaders. These settings determine which data to output to the fragment pipeline.

You can specify any of the following for `data_name`.

- `position`            Vertex coordinates (x y z w)
- `color`                Vertex color (R G B A)
- `texture0`            Texture coordinate 0 (u v)
- `texture0w`          The third component (w) of texture coordinate 0
- `texture1`            Texture coordinate 1 (u v)
- `texture2`            Texture coordinate 2 (u v)
- `quaternion`        Quaternion (x y z w)
- `view`                View vector (x y z)
- `generic`             General-purpose attribute (freely definable components)

Specify the corresponding output register in `mapped_register`. Because you can specify individual components of the output register, you can pack multiple attributes into a single register. The `generic` attribute is used with geometry shaders.

**Note:** Once a vertex shader has written to all of the output registers specified by these settings, that vertex shader is forced to end its processing and control moves to the next vertex data operation. (The `end` instruction must be called after all the output registers have been written to.) As a result, instructions might not be run if they come after the last attribute data has been written to the output registers.

### Code 8-8 Running an Instruction After Writing to the Output Registers

```
mov      o0, r0
mov      o1, r1 // Execution ends here if only o0 and o1 are specified
end      // The end instruction is required
nop      // This instruction might not run
nop      // This instruction might not run
```

**Note:** Vertex shaders are required to write to the entirety of all of the registers specified by this setting. (All components—x, y, z, and w—of the specified registers must be written. Even if `output_map` has not set all of the components, they must all be written. Dummy values can be written to components that have not been set.)

**Code 8-9 Writing to All Specified Registers**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( texture0 , o2.xy )
#pragma output_map ( texture1 , o3.xy )
#pragma output_map ( texture2 , o4.xy )

...
mov      o0, r0
mov      o1, r1
mov      o2, r2 // Although only 'xy' is specified for o2, o3, and o4,
mov      o3, r3 // some value must be written to the zw components
mov      o4, r4 //
```

**Note:** After a vertex shader has written to all of the output registers specified by these settings, you must not use any instructions that read or write the various registers. We do not guarantee behavior if you attempt to use register read-write instructions after the last write to an output register.

**Code 8-10 Running Instructions Illegally After Writing to the Output Registers**

```
mov      o0, r0
mov      o1, r1 // Execution ends here if only o0 and o1 were specified
mov      r0, c0 // Instructions that access registers here are prohibited
end      // The end instruction is required
mov      r1, c1 // Instructions that access registers here are prohibited
```

**Note:** Vertex shaders cannot write to the same output register more than once. Perform only one write operation on all of an output register's components per processing of each single vector. You cannot use multiple writes to write data one component at a time. We do not guarantee behavior if you use multiple write operations.

**Code 8-11 Writing to a Register Multiple Times**

```
// Example of valid processing
mov o0.xy, c0
mov o0.zw, c1.w // Each component has only been written once

// Example of invalid processing
mov o0.xy, c0
mov o0, c1.w // o0.x and o0.y have been written twice
```

**Note:** You can use `output_map` to specify attributes other than `generic` in up to 7 output registers. To set 8 or more attributes other than `generic` in the output registers, you must pack multiple attributes into a single output register. When attributes other than `generic` are set in 8 or more output registers, an `INVALID_OPERATION` error will be generated when the `glShaderBinary` function loads the code.

**Code 8-12 Packing Multiple Attributes into a Single Output Register**

```
#pragma output_map ( position , o0 )
#pragma output_map ( color , o1 )
#pragma output_map ( texture0 , o2.xy )
#pragma output_map ( texture1 , o2.zw ) // Pack multiple attributes into o2

...
mov o0, r0
mov o1, r1
mov o2.xy, r2.xy
mov o2.zw, r3.xy
```

## 8.7 #line

This changes the line number and the filename using the following syntax.

```
#line line_number ["filename"]
```

There must be a space between the line number and filename. Use double quotes around the filename and specify a line number that can be represented as a 32-bit integer (`int`) greater than or equal to 1. Behavior is undefined for any other value. Use a filename that is no longer than 128 single-byte alphanumeric characters and does not include spaces or special characters other than the following: `\/:*?"<>.`

**Code 8-13 #line Example**

```
#line 100 "newname.vsh"
```

## 9 Assembly Language Instruction Reference

### 9.1 Define Instructions

A define instruction sets a constant register value. The value does not change while the shader is running. Regardless of where it is declared in shader assembly code, it is valid throughout the entire linked object. You cannot set more than one value for the same register within the same set of linked objects. You also cannot use define instructions on registers already specified with `#pragma bind_symbol` within the same set of linked objects.

#### 9.1.1 `def` : Define Floating-Point Constants

##### 9.1.1.1 Calling Format

```
def    dest , value0 , value1 , value2 , value3
```

##### 9.1.1.2 Operands

- `dest`: Floating-point constant register
- `value0-4`: Floating-point values

##### 9.1.1.3 Overview

Sets the value of a floating-point constant register. You can set the value in decimal notation (using a decimal point) or integer notation (base 10 or base 16). When you specify values in hexadecimal notation, the hexadecimal bitmap specifies a 24-bit floating-point value (1 sign bit, 7 exponent bits, 16 significand bits). If a value larger than 24 bits is specified, the lower 24 bits are used and the rest is thrown away.

##### 9.1.1.4 Example

```
def    c0 , 1 , 1.5 , -0.5 , 0.25
def    c1 , 0x3f0000 , 0x3f8000 , 0xbe0000 , 0x3d0000 // This sets the same
                                                    // values as above
```

#### 9.1.2 `defb` : Define Boolean Constant

##### 9.1.2.1 Calling Format

```
defb   dest , value
```

##### 9.1.2.2 Operands

- `dest`: Boolean register
- `value`: true or false

##### 9.1.2.3 Overview

Sets the value of a Boolean register.

#### 9.1.2.4 Example

```
defb    b0 , true
defb    b1 , false
```

### 9.1.3 defi : Define Integer Constants

#### 9.1.3.1 Calling Format

```
defi    dest , count , init , step
```

#### 9.1.3.2 Operands

- **dest:** Integer register
- **count:** Integer value
- **init:** Integer value
- **step:** Integer value

#### 9.1.3.3 Overview

Sets the value of an integer register. Integer registers are used by the `loop` instruction. The `count` operand specifies a value that is one less than the number of times to run the group of instructions between the `loop` and `endloop` instructions. The `init` operand specifies the initial value of the loop-counter register. The `step` operand specifies the amount by which to increment the loop-counter register during each loop iteration. Specify a value in the range `[0, 255]` for both `count` and `init`. Specify a value in the range `[-128, 127]` for `step`. When its integer value is given in hexadecimal, `step` is specified as a two's complement number.

#### 9.1.3.4 Example

```
defi    i0 , 8 , 0 , 1 // Loops 9 times. aL has an initial value of 0 and is
                        // incremented by one during each loop iteration.
defi    i1 , 10 , 4 , 2 // Loops 11 times. aL has an initial value of 4 and is
                        // incremented by two during each loop iteration.
```

## 9.2 Arithmetic Instructions

Arithmetic instructions run arithmetic computations. In all these instructions, the components `x`, `y`, `z`, and `w` in the `src` operand(s) are swizzled before the instruction operation.

### 9.2.1 add : Add

#### 9.2.1.1 Calling Format

```
add    dest , src0 , src1
```

#### 9.2.1.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register

- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.1.3 Overview

Stores the sum of **src0** and **src1** in **dest**. You cannot specify a floating-point constant register for both **src0** and **src1**. Nor can you specify input registers with different indices in **src0** and **src1** at the same time.

### 9.2.1.4 Operation

```
dest.x = src0.x + src1.x
dest.y = src0.y + src1.y
dest.z = src0.z + src1.z
dest.w = src0.w + src1.w
```

### 9.2.1.5 Example

```
add    r0 , c1 , v2
add    o0.xy , r7.yz , c4.xx
```

## 9.2.2 dp3: Three-Component Dot Product

---

### 9.2.2.1 Calling Format

```
dp3    dest , src0 , src1
```

### 9.2.2.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.2.3 Overview

Stores the dot product of three components of **src0** and **src1** in **dest**. You cannot specify a floating-point constant register for both **src0** and **src1**. Nor can you specify input registers with different indices in **src0** and **src1** at the same time.

### 9.2.2.4 Operation

```
dot = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z )
dest.x = dot
dest.y = dot
dest.z = dot
dest.w = dot
```

### 9.2.2.5 Example

```
dp3    r0 , c1 , v2
dp3    o0 , r7.yzw , c4.xxy
```

## 9.2.3 dp4: Four-Component Dot Product

### 9.2.3.1 Calling Format

```
dp4      dest , src0 , src1
```

### 9.2.3.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.3.3 Overview

Stores the dot product of four components of `src0` and `src1` in `dest`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

### 9.2.3.4 Operation

$$\text{dot} = (\text{src0.x} \times \text{src1.x}) + (\text{src0.y} \times \text{src1.y}) + (\text{src0.z} \times \text{src1.z}) + (\text{src0.w} \times \text{src1.w})$$

```
dest.x = dot
```

```
dest.y = dot
```

```
dest.z = dot
```

```
dest.w = dot
```

### 9.2.3.5 Example

```
dp4      r0 , c1 , v2
dp4      o0 , r7.yzwx , c4.xxyw
```

## 9.2.4 dp4h: Homogeneous Dot Product

### 9.2.4.1 Calling Format

```
dp4h     dest , src0 , src1
```

### 9.2.4.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.4.3 Overview

Stores in `dest` the result of adding the `w` component of `src1` to the dot product of three components of `src0` and `src1`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.



#### 9.2.4.4 Operation

```
dot = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z ) +  
src1.w  
dest.x = dot  
dest.y = dot  
dest.z = dot  
dest.w = dor
```

#### 9.2.4.5

```
dest.w = dot
```

#### 9.2.4.6 Example

```
dph      r0 , c1 , v2  
dph      o0 , r7.yzwx , c4.xxyw
```

### 9.2.5 dst: Distance Vector

---

#### 9.2.5.1 Calling Format

```
dst      dest , src0 , src1
```

#### 9.2.5.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

#### 9.2.5.3 Overview

Calculates the vector distance. Set the y and z components of `src0` to the squared distance, and the y and w components of `src1` to the reciprocal of the distance. The components of `dest` respectively store 1, the distance, the squared distance, and the reciprocal of the distance. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

#### 9.2.5.4 Operation

```
dest.x = 1  
dest.y = src0.y × src1.y  
dest.z = src0.z  
dest.w = src1.w
```

#### 9.2.5.5 Example

#### 9.2.5.6

```
dst      r0 , c1 , v2
```

## 9.2.6 exp: Exponential Base 2

### 9.2.6.1 Calling Format

```
exp      dest , src{ .x | .y | .z | .w }
```

### 9.2.6.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.6.3 Overview

Calculates a power of 2. Because this instruction can only calculate a single component, you must specify only one component in `src`.

### 9.2.6.4 Operation

```
tmp = src { .x | .y | .z | .w }
dest.x = 2 ^ tmp
dest.y = 2 ^ tmp
dest.z = 2 ^ tmp
dest.w = 2 ^ tmp
exp      r0 , c1.x
```

## 9.2.7 flr: Floor

### 9.2.7.1 Calling Format

```
flr      dest , src
```

### 9.2.7.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.7.3 Overview

Stores the largest integer less than or equal to `src` in `dest`.

### 9.2.7.4 Operation

```
dest.x = floor ( src.x )
dest.y = floor ( src.y )
dest.z = floor ( src.z )
dest.w = floor ( src.w )
```

### 9.2.7.5

```
flr      r0, r1
```

## 9.2.8 litp: Light Coefficients

### 9.2.8.1 Calling Format

```
litp    dest , src
```

### 9.2.8.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.8.3 Overview

Partially calculates lighting. This instruction also changes the status registers at the same time.

### 9.2.8.4 Operation

```
dest.x = ( src.x < 0 ) ? 0 : src.x
dest.y = ( src.y < -128 ) ? -128 : ( src.y > 128 ? 128 : src.y )
dest.z = 0
dest.w = ( src.w < 0 ) ? 0 : src.w
status_reg0 = ( src.x > 0 ) ? 1 : 0
status_reg1 = ( src.w > 0 ) ? 1 : 0
```

### 9.2.8.5 Example

```
litp    r0 , c1
```

## 9.2.9 log: Logarithm Base 2

### 9.2.9.1 Calling Format

```
log     dest , src{ .x | .y | .z | .w }
```

### 9.2.9.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.9.3 Overview

Calculates the base-2 logarithm. Because this instruction can only calculate a single component, you must specify only one component in **src**.

### 9.2.9.4 Operation

```
tmp = src { .x | .y | .z | .w }
dest.x = log2 ( tmp )
dest.y = log2 ( tmp )
dest.z = log2 ( tmp )
dest.w = log2 ( tmp )
```

### 9.2.9.5 Example

```
log     r0 , c1.x
```

## 9.2.10 **mad**: Multiply and Add

### 9.2.10.1 Calling Format

```
mad    dest , src0 , src1 , src2
```

### 9.2.10.2 Operands

- **dest**: Output register or temporary register
- **src0**: Temporary register or input register
- **src1**: Temporary register, input register, or floating-point constant register
- **src2**: Temporary register, input register, or floating-point constant register

### 9.2.10.3 Overview

Stores in **dest** the result of adding **src2** to the product of **src0** and **src1**. You cannot specify a floating-point constant register for both **src1** and **src2**. Nor can you specify input registers with different indices in **src0**, **src1**, and **src2** at the same time.

### 9.2.10.4 Operation

```
dest.x = src0.x × src1.x + src2.x
dest.y = src0.y × src1.y + src2.y
dest.z = src0.z × src1.z + src2.z
dest.w = src0.w × src1.w + src2.w
```

### 9.2.10.5 Example

```
mad    r0 , r1, c1 , v2
```

## 9.2.11 **max**: Maximum

### 9.2.11.1 Calling Format

```
max    dest , src0 , src1
```

### 9.2.11.2 Operands

- **dest**: Output register or temporary register
- **src0**: Temporary register, input register, or floating-point constant register
- **src1**: Temporary register, input register, or floating-point constant register

### 9.2.11.3 Overview

Compares **src0** and **src1** and stores the larger value in **dest**. You cannot specify a floating-point constant register for both **src0** and **src1**. Nor can you specify input registers with different indices in **src0** and **src1** at the same time.

### 9.2.11.4 Operation

```
dest.x = ( src0.x > src1.x ) ? src0.x : src1.x
dest.y = ( src0.y > src1.y ) ? src0.y : src1.y
```

```
dest.z = ( src0.z > src1.z ) ? src0.z : src1.z  
dest.w = ( src0.w > src1.w ) ? src0.w : src1.w
```

#### 9.2.11.5 Example

```
max      r0 , r1 , c1
```

### 9.2.12 min: Minimum

---

#### 9.2.12.1 Calling Format

```
min      dest , src0 , src1
```

#### 9.2.12.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

#### 9.2.12.3 Overview

Compares `src0` and `src1` and stores the smaller value in `dest`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

#### 9.2.12.4 Operation

```
dest.x = ( src0.x > src1.x ) ? src1.x : src0.x  
dest.y = ( src0.y > src1.y ) ? src1.y : src0.y  
dest.z = ( src0.z > src1.z ) ? src1.z : src0.z  
dest.w = ( src0.w > src1.w ) ? src1.w : src0.w
```

#### 9.2.12.5 Example

```
min      r0 , r1 , c1
```

### 9.2.13 mov: Move

---

#### 9.2.13.1 Calling Format

```
mov      dest , src
```

#### 9.2.13.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

#### 9.2.13.3 Overview

Copies the content of `src` into `dest`.

#### 9.2.13.4 Operation

```
dest = src
```

### 9.2.13.5 Example

```
mov      r0 , c1
```

## 9.2.14 mova: Move to Address Register

### 9.2.14.1 Calling Format

```
mova     dest{ .x | .y | .xy } , src
```

### 9.2.14.2 Operands

- **dest:** Address register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.14.3 Overview

Copies the content of `src` into `dest`. The fractional part of `src` (everything below the decimal point) is truncated. Behavior is undefined if you assign a value that does not lie in the range  $[-95, 95]$ .

The address register must use a mask of `.x`, `.y`, or `.xy`. Consecutive calls to this instruction will result in an error.

### 9.2.14.4 Operation

`dest = src`

### 9.2.14.5 Example

```
mova     a0.x , c1
```

## 9.2.15 mul: Multiply

### 9.2.15.1 Calling Format

```
mul      dest , src0 , src1
```

### 9.2.15.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.15.3 Overview

Stores the product of `src0` and `src1` in `dest`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

### 9.2.15.4 Operation

```
dest.x = src0.x × src1.x
dest.y = src0.y × src1.y
dest.z = src0.z × src1.z
dest.w = src0.w × src1.w
```

### 9.2.15.5 Example

```
mul    r0 , c1 , v2
mul    o0.xy , r7.yzww , c4.xxyz
```

## 9.2.16 nop: No Operation

---

### 9.2.16.1 Calling Format

```
nop
```

### 9.2.16.2 Operands

None

### 9.2.16.3 Overview

This instruction does nothing.

### 9.2.16.4 Operation

None

### 9.2.16.5 Example

```
nop
```

## 9.2.17 rcp: Reciprocal

---

### 9.2.17.1 Calling Format

```
rcp    dest , src{ .x | .y | .z | .w }
```

### 9.2.17.2 Operands

- **dest:** Output register or temporary register
- **src:** Temporary register, input register, or floating-point constant register

### 9.2.17.3 Overview

Calculates the reciprocal. Because this instruction can only calculate a single component, you must specify only one component in **src**.

### 9.2.17.4 Operation

```
tmp = src { .x | .y | .z | .w }
dest.x = 1 / tmp
dest.y = 1 / tmp
dest.z = 1 / tmp
dest.w = 1 / tmp
```

### 9.2.17.5 Example

```
rcp    r0 , c1.x
```

## 9.2.18 **rsq**: Reciprocal Square Root

### 9.2.18.1 Calling Format

```
rsq      dest , src{ .x | .y | .z | .w }
```

### 9.2.18.2 Operands

- **dest**: Output register or temporary register
- **src**: Temporary register, input register, or floating-point constant register

### 9.2.18.3 Overview

Calculates the square root of the reciprocal. Because this instruction can only calculate a single component, you must specify only one component in **src**.

### 9.2.18.4 Operation

```
tmp = src { .x | .y | .z | .w }
dest.x = 1 / sqrt ( tmp )
dest.y = 1 / sqrt ( tmp )
dest.z = 1 / sqrt ( tmp )
dest.w = 1 / sqrt ( tmp )
```

### 9.2.18.5 Example

```
rsq      r0 , c1.x
```

## 9.2.19 **sge**: Set on Greater Than or Equal

### 9.2.19.1 Calling Format

```
sge      dest , src0 , src1
```

### 9.2.19.2 Operands

- **dest**: Output register or temporary register
- **src0**: Temporary register, input register, or floating-point constant register
- **src1**: Temporary register, input register, or floating-point constant register

### 9.2.19.3 Overview

Stores 1 in **dest** when **src0** is greater than or equal to **src1**; otherwise, stores 0 in **dest**. You cannot specify a floating-point constant register for both **src0** and **src1**. Nor can you specify input registers with different indices in **src0** and **src1** at the same time.

### 9.2.19.4 Operation

```
dest.x = ( src0.x >= src1.x ) ? 1 : 0
dest.y = ( src0.y >= src1.y ) ? 1 : 0
dest.z = ( src0.z >= src1.z ) ? 1 : 0
dest.w = ( src0.w >= src1.w ) ? 1 : 0
```



### 9.2.19.5 Example

```
sge      r0 , c1 , v2
sge      o0.xy , r7.yzww , c4.xyz
```

## 9.2.20 slt: Set on Less Than

---

### 9.2.20.1 Calling Format

```
slt      dest , src0 , src1
```

### 9.2.20.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.2.20.3 Overview

Stores 1 in `dest` when `src0` is smaller than `src1`; otherwise, stores 0 in `dest`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

### 9.2.20.4 Operation

```
dest.x = ( src0.x < src1.x ) ? 1 : 0
dest.y = ( src0.y < src1.y ) ? 1 : 0
dest.z = ( src0.z < src1.z ) ? 1 : 0
dest.w = ( src0.w < src1.w ) ? 1 : 0
```

### 9.2.20.5 Example

```
slt      r0 , c1 , v2
slt      o0.xy , r7.yzww , c4.xyz
```

## 9.3 Macro Instructions

---

Macro instructions expand into a combination of arithmetic instructions.

### 9.3.1 sub: Subtract

---

#### 9.3.1.1 Calling Format

```
sub      dest , src0 , src1
```

#### 9.3.1.2 Operands

- **dest:** Output register or temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.1.3 Overview

Stores the difference of `src0` and `src1` in `dest`. You cannot specify a floating-point constant register for both `src0` and `src1`. Nor can you specify input registers with different indices in `src0` and `src1` at the same time.

### 9.3.1.4 Operation

```
dest.x = src0.x - src1.x
dest.y = src0.y - src1.y
dest.z = src0.z - src1.z
dest.w = src0.w - src1.w
```

### 9.3.1.5 Post-Macro Expansion

```
add    dest , src0 , -src1
```

### 9.3.1.6 Example

```
sub    r0 , c1 , v2
sub    o0.xy , r7.yzww , c4.xxyz
```

## 9.3.2 abs : Absolute

### 9.3.2.1 Calling Format

```
abs    dest , src
```

### 9.3.2.2 Operands

- `dest`: Output register or temporary register
- `src`: Temporary register or input register

### 9.3.2.3 Overview

Stores the absolute value of `src` in `dest`.

### 9.3.2.4 Operation

```
dest.x = abs ( src0.x )
dest.y = abs ( src0.y )
dest.z = abs ( src0.z )
dest.w = abs ( src0.w )
```

### 9.3.2.5 Post-Macro Expansion

```
max    dest , src , -src
```

### 9.3.2.6 Example

```
abs    r0 , r1
abs    o0 , r7.yzww
```

### 9.3.3 **crs**: Cross Product

#### 9.3.3.1 Calling Format

```
crs      dest{ .x | .y | .z | .xy | .xz | .yz | .xyz } , src0 , src1
```

#### 9.3.3.2 Operands

- **dest**: Temporary register
- **src0**: Temporary register, input register, or floating-point constant register
- **src1**: Temporary register, input register, or floating-point constant register

#### 9.3.3.3 Overview

Stores the cross product of three components of **src0** and **src1** in **dest**. Neither **src0** nor **src1** can be swizzled. You must use one of the following masks for the **dest** operand: **.x**, **.y**, **.z**, **.xy**, **.xz**, **.yz**, or **.xyz**.

You cannot specify any of the following:

- the same register for **src0** and **dest**
- the same register for **src1** and **dest**
- a mask other than the default (**.xyzw**) for either **src0** or **src1**
- a floating-point constant register for both **src0** and **src1**
- input registers with different indices for **src0** and **src1** at the same time

#### 9.3.3.4 Operation

```
dest.x = src0.y × src1.z - src0.z × src1.y
dest.y = src0.z × src1.x - src0.x × src1.z
dest.z = src0.x × src1.y - src0.y × src1.x
```

#### 9.3.3.5 Post-Macro Expansion

```
mul      dest.xyz , src0.yzx , src1.zxy
mad      dest.xyz , -src1.yzx , src0.zxy , dest
```

#### 9.3.3.6 Example

```
crs      r0.xyz , c1 , v2
```

### 9.3.4 **frc**: Fraction

#### 9.3.4.1 Calling Format

```
frc      dest , src
```

#### 9.3.4.2 Operands

- **dest**: Temporary register
- **src**: Temporary register, input register, or floating-point constant register

### 9.3.4.3 Overview

Stores in `dest` the difference between the value of `src` and the largest integer less than or equal to `src`. You cannot specify the same register for `src` and `dest`.

### 9.3.4.4 Operation

```
dest.x = src.x - floor ( src.x )
dest.y = src.y - floor ( src.y )
dest.z = src.z - floor ( src.z )
dest.w = src.w - floor ( src.w )
```

### 9.3.4.5 Post-Macro Expansion

```
flr      dest , src
add      dest , src , -dest
```

### 9.3.4.6 Example

```
frc      r0 , v1
```

## 9.3.5 lrp: Linear Interpolation

### 9.3.5.1 Calling Format

```
lrp      dest , src0 , src1 , src2
```

### 9.3.5.2 Operands

- `dest`: Temporary register
- `src0`: Temporary register, input register, or floating-point constant register
- `src1`: Temporary register, input register, or floating-point constant register
- `src2`: Temporary register, input register, or floating-point constant register

### 9.3.5.3 Overview

Stores in `dest` the result of using `src0` to linearly interpolate between `src1` and `src2`. You can specify only one floating-point constant register among the three operands `src0`, `src1`, and `src2`. However, there is one exception: you can specify two floating-point constant registers at the same time if you specify them for `src0` and `src1`.

You cannot do any of the following:

- specify the same register for `dest` and `src0`
- use the same register for `dest` and `src2`
- specify input registers with different indices in any combination of `src0`, `src1`, and `src2` at the same time

### 9.3.5.4 Operation

```
dest.x = src0.x × src1.x + (1 - src0.x) × src2.x
dest.y = src0.y × src1.y + (1 - src0.y) × src2.y
```

```
dest.z = src0.z × src1.z + (1 - src0.z) × src2.z
dest.w = src0.w × src1.w + (1 - src0.w) × src2.w
```

### 9.3.5.5 Post-Macro Expansion

```
add      dest , src1 , -src2
mad      dest , dest , src0 , src2
```

### 9.3.5.6 Example

```
lrp      r0 , v1 , c2 , r3
```

## 9.3.6 m3x2: 3x2 Multiply

### 9.3.6.1 Calling Format

```
m3x2     dest.xy , src0 , src1
```

### 9.3.6.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.6.3 Overview

Stores the result of multiplying a 3x2 matrix and a 3-component vector in **dest**. Specify the first register of the 3x2 matrix in **src1** (in other words, when **src1** is **r0**, that means the 3x2 matrix is stored in **r0** and **r1**).

You cannot specify any of the following:

- a floating-point constant register for both **src0** and **src1**
- an input register for both **src0** and **src1**
- the same register for **dest** and **src0**

You must use the mask **.xy** for **dest**.

**Note:** If you set **dest** to the register that consecutively follows **src1** (this is **src2** in the expanded macro below), the content of that register is updated after macro expansion when the instruction executes, causing unexpected results.

### 9.3.6.4 Operation

```
src2 = Next_Index_Of ( src1 )
dest.x = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z )
dest.y = ( src0.x × src2.x ) + ( src0.y × src2.y ) + ( src0.z × src2.z )
```

### 9.3.6.5 Post-Macro Expansion

```
dp3      dest.x , src0 , src1
dp3      dest.y , src0 , src2          // src2 = next index after src1
```

### 9.3.6.6 Example

```
m3x2      r0.xy , r1 , c0
// This is expanded as follows
// dp3     r0.x , r1 , c0
// dp3     r0.y , r1 , c1
```

## 9.3.7 m3x3: 3x3 Multiply

### 9.3.7.1 Calling Format

```
m3x3      dest.xyz , src0 , src1
```

### 9.3.7.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.7.3 Overview

Stores the result of multiplying a 3x3 matrix and a 3-component vector in **dest**. Specify the first register of the 3x3 matrix in **src1** (in other words, when **src1** is **r0**, that means the 3x3 matrix is stored in **r0**, **r1**, and **r2**).

You cannot specify any of the following:

- a floating-point constant register for both **src0** and **src1**
- an input register for both **src0** and **src1**
- the same register for **dest** and **src0**

You must use the mask **.xyz** for **dest**.

**Note:** If you set **dest** to a register that consecutively follows **src1** (these registers are **src2** and **src3** in the expanded macro below), the content of that register is updated after macro expansion when the instruction executes, causing unexpected results.

### 9.3.7.4 Operation

```
src2 = Next_Index_Of ( src1 )
src3 = Next_Index_Of ( src2 )
dest.x = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z )
dest.y = ( src0.x × src2.x ) + ( src0.y × src2.y ) + ( src0.z × src2.z )
dest.z = ( src0.x × src3.x ) + ( src0.y × src3.y ) + ( src0.z × src3.z )
```

### 9.3.7.5 Post-Macro Expansion

```
dp3      dest.x , src0 , src1
dp3      dest.y , src0 , src2      // src2 = next index after src1
dp3      dest.z , src0 , src3      // src3 = next index after src2
```

### 9.3.7.6 Example

```
m3x3      r0.xyz , r1 , c0
// This is expanded as follows
// dp3     r0.x , r1 , c0
// dp3     r0.y , r1 , c1
// dp3     r0.z , r1 , c2
```

## 9.3.8 m3x4: 3x4 Multiply

### 9.3.8.1 Calling Format

```
m3x4      dest , src0 , src1
```

### 9.3.8.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.8.3 Overview

Stores the result of multiplying a 3x4 matrix and a 3-component vector in **dest**. Specify the first register of the 3x4 matrix in **src1** (in other words, when **src1** is **r0**, that means the 3x4 matrix is stored in **r0**, **r1**, **r2**, and **r3**).

You cannot specify any of the following:

- a floating-point constant register for both **src0** and **src1**
- an input register for both **src0** and **src1**
- a mask other than the default (**.xyzw**) for **dest**
- the same register for **dest** and **src0**

**Note:** If you set **dest** to a register that consecutively follows **src1** (these registers are **src2**, **src3** and **src4** in the expanded macro below), the content of that register is updated after macro expansion when the instruction executes, causing unexpected results.

### 9.3.8.4 Operation

```
src2 = Next_Index_Of ( src1 )
src3 = Next_Index_Of ( src2 )
src4 = Next_Index_Of ( src3 )
dest.x = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z )
dest.y = ( src0.x × src2.x ) + ( src0.y × src2.y ) + ( src0.z × src2.z )
dest.z = ( src0.x × src3.x ) + ( src0.y × src3.y ) + ( src0.z × src3.z )
dest.w = ( src0.x × src4.x ) + ( src0.y × src4.y ) + ( src0.z × src4.z )
```

### 9.3.8.5 Post-Macro Expansion

```
dp3      dest.x , src0 , src1
dp3      dest.y , src0 , src2          // src2 = next index after src1
```

```
dp3      dest.z , src0 , src3      // src3 = next index after src2
dp3      dest.w , src0 , src4      // src4 = next index after src3
```

### 9.3.8.6 Example

```
m3x4     r0 , r1 , c0
// This is expanded as follows
// dp3    r0.x , r1 , c0
// dp3    r0.y , r1 , c1
// dp3    r0.z , r1 , c2
// dp3    r0.w , r1 , c3
```

## 9.3.9 m4x3: 4x3 Multiply

### 9.3.9.1 Calling Format

```
m4x3     dest.xyz , src0 , src1
```

### 9.3.9.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.9.3 Overview

Stores the result of multiplying a 4x3 matrix and a 4-component vector in **dest**. Specify the first register of the 4x3 matrix in **src1** (in other words, when **src1** is **r0**, that means the 4x3 matrix is stored in **r0**, **r1**, and **r2**).

You cannot specify any of the following:

- a floating-point constant register for both **src0** and **src1**
- an input register for both **src0** and **src1**
- the same register for **dest** and **src0**

You must use the mask **.xyz** for **dest**.

**Note:** If you set **dest** equal to a register that consecutively follows **src1** (these registers are **src2** and **src3** in the expanded macro below), the content of that register is updated after macro expansion when the instruction executes, causing unexpected results.

### 9.3.9.4 Operation

```
src2 = Next_Index_Of ( src1 )
src3 = Next_Index_Of ( src2 )
dest.x = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z ) +
( src0.w × src1.w )
dest.y = ( src0.x × src2.x ) + ( src0.y × src2.y ) + ( src0.z × src2.z ) +
( src0.w × src2.w )
```



```
dest.z = ( src0.x × src3.x ) + ( src0.y × src3.y ) + ( src0.z × src3.z ) +
( src0.w × src3.w )
```

### 9.3.9.5 Post-Macro Expansion

```
dp4      dest.x , src0 , src1
dp4      dest.y , src0 , src2      // src2 = next index after src1
dp4      dest.z , src0 , src3      // src3 = next index after src2
```

### 9.3.9.6 Example

```
m4x3     r0.xyz , r1 , c0
// This is expanded as follows
// dp4    r0.x , r1 , c0
// dp4    r0.y , r1 , c1
// dp4    r0.z , r1 , c2
```

## 9.3.10 m4x4: 4x4 Multiply

### 9.3.10.1 Calling Format

```
m4x4     dest , src0 , src1
```

### 9.3.10.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.10.3 Overview

Stores the result of multiplying a 4x4 matrix and a 4-component vector in **dest**. Specify the first register of the 4x4 matrix in **src1** (in other words, when **src1** is **r0**, that means the 4x4 matrix is stored in **r0**, **r1**, **r2**, and **r3**).

You cannot specify any of the following:

- a floating-point constant register for both **src0** and **src1**
- an input register for both **src0** and **src1**
- the same register for **dest** and **src0**

You cannot use a mask other than the default (**.xyzw**) for **dest**.

**Note:** If you set **dest** equal to a register that consecutively follows **src1** (these registers are **src2**, **src3**, and **src4** in the expanded macro below), the content of that register is updated after macro expansion when the instruction executes, causing unexpected results.

### 9.3.10.4 Operation

```
src2 = Next_Index_Of ( src1 )
src3 = Next_Index_Of ( src2 )
src4 = Next_Index_Of ( src3 )
```

```

dest.x = ( src0.x × src1.x ) + ( src0.y × src1.y ) + ( src0.z × src1.z ) +
( src0.w × src1.w )
dest.y = ( src0.x × src2.x ) + ( src0.y × src2.y ) + ( src0.z × src2.z ) +
( src0.w × src2.w )
dest.z = ( src0.x × src3.x ) + ( src0.y × src3.y ) + ( src0.z × src3.z ) +
( src0.w × src3.w )
dest.w = ( src0.x × src4.x ) + ( src0.y × src4.y ) + ( src0.z × src4.z ) +
( src0.w × src4.w )

```

### 9.3.10.5 Post-Macro Expansion

```

dp4      dest.x , src0 , src1
dp4      dest.y , src0 , src2      // src2 = next index after src1
dp4      dest.z , src0 , src3      // src3 = next index after src2
dp4      dest.w , src0 , src4      // src4 = next index after src3

```

### 9.3.10.6 Example

```

m4x3      r0 , r1 , c0
// This is expanded as follows
// dp4      r0.x , r1 , c0
// dp4      r0.y , r1 , c1
// dp4      r0.z , r1 , c2
// dp4      r0.w , r1 , c3

```

## 9.3.11 nrm: Normalize

### 9.3.11.1 Calling Format

```
nrm      dest , src
```

### 9.3.11.2 Operands

- **dest:** Temporary register
- **src:** Temporary register or input register

### 9.3.11.3 Overview

Stores the result of normalizing **src** in **dest**. You cannot specify the same register for **src** and **dest**.

### 9.3.11.4 Operation

```

tmp = sqrt ( src.x × src.x + src.y × src.y + src.z × src.z + src.w ×
src.w )
dest.x = src.x × ( 1 / tmp )
dest.y = src.y × ( 1 / tmp )
dest.z = src.z × ( 1 / tmp )
dest.w = src.w × ( 1 / tmp )

```

### 9.3.11.5 Post-Macro Expansion

```
dp4      dest.x , src , src
rsq      dest.x , dest.x
mul      dest , src , dest.x
```

### 9.3.11.6 Example

```
nrm      r0 , v0
```

## 9.3.12 pow : Power

### 9.3.12.1 Calling Format

```
pow      dest , src0{ .x | .y | .z | .w} , src1{ .x | .y | .z | .w}
```

### 9.3.12.2 Operands

- **dest:** Temporary register
- **src0:** Temporary register, input register, or floating-point constant register
- **src1:** Temporary register, input register, or floating-point constant register

### 9.3.12.3 Overview

Stores the result of raising **src0** to the **src1** power in **dest**. You must specify one of the four components (.x, .y, .z, or .w) for **src0** and **src1**. You cannot specify the same register for **src1** and **dest**.

### 9.3.12.4 Operation

```
tmp = src0{ .x | .y | .z | .w} ^ src1{ .x | .y | .z | .w}
dest.x = tmp
dest.y = tmp
dest.z = tmp
dest.w = tmp
```

### 9.3.12.5 Post-Macro Expansion

```
log      dest.z , src0{ .x | .y | .z | .w}
mul      dest.z , dest.z , src1{ .x | .y | .z | .w}
exp      dest , dest.z
```

### 9.3.12.6 Example

```
pow      r0 , r1.y , r2.x
```

## 9.3.13 sgn: Sign

### 9.3.13.1 Calling Format

```
sgn      dest , src0 , src1 , src2
```

### 9.3.13.2 Operands

- **dest:** Temporary register or output register
- **src0:** Temporary register or input register
- **src1:** Temporary register
- **src2:** Temporary register

### 9.3.13.3 Overview

Stores 1, 0, or -1 in **dest** when **src0** is positive, zero, or negative, respectively. **src1** and **src2** are used to perform the calculations. You cannot use swizzling or specify negative signs with **src1** and **src2**. You cannot specify the same register for **src1** and **src2**. You cannot specify the same register for **src0** and **src1**.

### 9.3.13.4 Operation

```
for ( each component )
{
    if ( src0.component < 0 )
        dest.component = -1
    else if ( src0.component == 0 )
        dest.component = 0
    else
        dest.component = 1
}
```

### 9.3.13.5 Post-Macro Expansion

```
slt      src1 , src0 , -src0
slt      src2 , -src0 , src0
add      dest , src2 , -src1
```

### 9.3.13.6 Example

```
sgn      r0 , v1 , r2 , r3
```

## 9.3.14 sincos: Sine and Cosine

### 9.3.14.1 Calling Format

```
sincos    dest{ .x | .y | .xy } , src0{ .x | .y | .z | .w } , src1 , src2
```

### 9.3.14.2 Operands

- **dest:** Temporary register
- **src0:** Temporary register or input register
- **src1:** Temporary register
- **src2:** Temporary register

### 9.3.14.3 Overview

Calculates the sine and cosine of the component value specified by `src0` and stores the result in `dest`. Values are given in radians. The cosine and sine values are output to the x and y components, respectively, in `dest`. You must specify one of the following masks for `dest`: `.x`, `.y`, or `.xy`. The content of the z component in `dest` is not preserved because it is used during calculations. You must specify one of the four components (`.x`, `.y`, `.z`, or `.w`) for `src0`. `src1` and `src2` are used to perform the calculations. You cannot use swizzling or specify negative signs with `src1` and `src2`. A different register must be specified for each operand. The component specified by `src0` must have a value between  $-\pi$  and  $\pi$ . This macro calculates an approximate value using a Taylor expansion. Because a Taylor expansion requires several coefficients, the floating-point constant registers `c93`, `c94`, and `c95` are automatically defined. You cannot use this instruction and define either `c93`, `c94`, or `c95` with the `def` instruction.

### 9.3.14.4 Operation

```
tmp = src0{ .x | .y | .z | .w }
dest.x = cos ( tmp )
dest.y = sin ( tmp )
dest.z = undefined      // This is used during calculations
```

### 9.3.14.5 Post-Macro Expansion

```
def    c93 , 0xbe0000, 0xbc5555, 0x3f0000, 0x3f0000    // -0.5, -1/3!, 1.0, 1.0
def    c94 , 0xb56c16, 0xb2a01a, 0x3a5555, 0x381111    // -1/6!, -1/7!, 1/4!, 1/5!
def    c95 , 0xa927e4, 0xa5ae64, 0x2fa01a, 0x2c71de    // -1/10!, -1/11!, 1/8!, 1/9!

mov     src1 , c95
mov     src2 , c94
mul     dest.z , src0{ .x | .y | .z | .w } , src0{ .x | .y | .z | .w }
mad     dest.xy , dest.z , src1.xy , src1.zw
mad     dest.xy , dest.z , dest.xy , src2.xy
mad     dest.xy , dest.z , dest.xy , src2.zw
mov     src1 , c93
mad     dest.xy , dest.z , dest.xy , src1.xy
mad     dest.xy , dest.z , dest.xy , src1.zw
mul     dest.y , dest.y , src0{ .x | .y | .z | .w }
```

### 9.3.14.6 Example

```
sincos   r0.xy , v1.x , r2 , r3
```

## 9.4 Flow Control Instructions

The flow control instructions control the flow of execution.

## 9.4.1 `call`: Call Subroutine

### 9.4.1.1 Calling Format

```
call    label
```

### 9.4.1.2 Operands

- `label`: Label name

### 9.4.1.3 Overview

Causes control to jump to the address of the specified label name. If processing ends before reaching a `ret` instruction after the label address, it will return to the address immediately after this instruction. You cannot call a label if a `ret` instruction has not been set for it. You can nest up to four call instructions (`call`, `callc`, and `callb`). Behavior is undefined for five or more nested calls. When call instructions are nested, behavior is undefined if a call instruction is invoked immediately before a `ret` instruction.

### 9.4.1.4 Operation

```
retaddr = pc + 1
pc = get_label_address ( label )
while (1)
{
    execute_current_instruction ( )
    if ( current_instruction ( ) == ret )
    {
        pc = retaddr
        break
    }
}
```

### 9.4.1.5 Example

```
call    subfunction0

subfunction0:
..
ret
```

## 9.4.2 `callb`: Boolean Call

### 9.4.2.1 Calling Format

```
callb    src, label
```

### 9.4.2.2 Operands

- `src`: Boolean register

- **label:** Label name

#### 9.4.2.3 Overview

Causes control to jump to the address of the specified label name when the content of the specified Boolean register is `true`. If processing ends before reaching a `ret` instruction after the label address, it will return to the address immediately after this instruction. You cannot call a label if a `ret` instruction has not been set for it. You can nest up to four call instructions (`call`, `callc`, and `callb`). Behavior is undefined for five or more nested calls. When call instructions are nested, behavior is undefined if a call instruction is invoked immediately before a `ret` instruction.

#### 9.4.2.4 Operation

```
if ( src )
    call    label
```

#### 9.4.2.5 Example

```
callb     b0 , subfunction0

subfunction0:
..
ret
```

### 9.4.3 `callc`: Condition Call

---

#### 9.4.3.1 Calling Format

```
callc     status0 , status1 , mode , label
```

#### 9.4.3.2 Operands

- **status0:** Value (either 0 or 1) of status register 0
- **status1:** Value (either 0 or 1) of status register 1
- **mode:** Conditional mode
  - 0: OR
  - 1: AND
  - 2: Only status register 0
  - 3: Only status register 1
- **label:** Label name

#### 9.4.3.3 Overview

Calls a function conditionally based on the status register values.

The equality of the values specified by `status0` (or `status1`) and status register 0 (or 1) is taken to be the condition. This condition is `true` when either status register 0 or 1 match when `mode` is 0; when both status registers match when `mode` is 1; when status register 0 matches when `mode` is 2; and when status register 1 matches when `mode` is 3.

This instruction causes control to jump to the address of the specified label when the condition is true. If processing ends before reaching a `ret` instruction after the label address, it will return to the address immediately after this instruction. You cannot call a label if a `ret` instruction has not been set for it. You can nest up to four call instructions (`call`, `callc`, and `callb`). Behavior is undefined for five or more nested calls. When call instructions are nested, behavior is undefined if a call instruction is invoked immediately before a `ret` instruction.

#### 9.4.3.4 Operation

```
switch ( mode )
{
case 0 :
    if ( status0 == Status_register0 || status1 == Status_register1 )
        call label
    break;
case 1 :
    if ( status0 == Status_register0 && status1 == Status_register1 )
        call label
    break;
case 2 :
    if ( status0 == Status_register0 )
        call label
    break;
case 3 :
    if ( status1 == Status_register1 )
        call label
    break;
}
```

#### 9.4.3.5 Example

```
Callc  1 , 1 , 0 , subfunction0 // Calls subfunction0 when status register 0
                                // or status register 1 is equal to 1

subfunction0:
..
ret
```

### 9.4.4 jpb: Boolean Jump

#### 9.4.4.1 Calling Format

```
jpb      src, value , label
```

#### 9.4.4.2 Operands

- `src`: Boolean register



- **value:** true or false
- **label:** Label name

#### 9.4.4.3 Overview

Causes control to jump to the address of the specified label name when the content of the Boolean register specified by `src` matches the value specified by `value`. Unlike the `call` instruction, control does not return at a `ret` instruction and you can also specify labels without setting a `ret` instruction. You cannot call this instruction within an `if` or `loop` block. Jumping to an external location from between the `main` and `endmain` labels and from within subroutines results in undefined behavior. In the same way, jumping to a `ret` instruction within a subroutine also results in undefined behavior.

#### 9.4.4.4 Operation

```
if ( src == value )  
    jump to      label
```

#### 9.4.4.5 Example

```
jpb      b0 , true , subfunction0  
jpb      b1 , false , subfunction0  
  
subfunction0:  
..
```

### 9.4.5 jpc: Condition Jump

---

#### 9.4.5.1 Calling Format

```
jpc      status0 , status1 , mode , label
```

#### 9.4.5.2 Operands

- **status0:** Value (either 0 or 1) of status register 0
- **status1:** Value (either 0 or 1) of status register 1
- **mode:** Conditional mode
  - 0: OR
  - 1: AND
  - 2: Only status register 0
  - 3: Only status register 1
- **label:** Label name

#### 9.4.5.3 Overview

Causes control to jump conditionally based on the status register values.

The equality of the values specified by `status0` (or `status1`) and status register 0 (or 1) is taken to be the condition. This condition is `true` when either status register 0 or 1 match when `mode` is 0; when both status registers match when `mode` is 1; when status register 0 matches when `mode` is 2; and when status register 1 matches when `mode` is 3.

This instruction causes control to jump to the address of the specified label when the condition is true. Unlike the `call` instruction, control does not return at a `ret` instruction and you can also specify labels without setting a `ret` instruction. You cannot call this instruction within an `if` or `loop` block. Jumping to an external location from between the `main` and `endmain` labels and from within subroutines results in undefined behavior. In the same way, jumping to a `ret` instruction within a subroutine also results in undefined behavior.

#### 9.4.5.4 Operation

```
switch ( mode )
{
case 0 :
    if ( status0 == Status_register0 || status1 == Status_register1 )
        jump to label
    break;
case 1 :
    if ( status0 == Status_register0 && status1 == Status_register1 )
        jump to label
    break;
case 2 :
    if ( status0 == Status_register0 )
        jump to label
    break;
case 3 :
    if ( status1 == Status_register1 )
        jump to label
    break;
}
```

#### 9.4.5.5 Example

```
jpc 1, 1, 0, subfunction0 // Calls subfunction0 when status register 0 or
                          // status register 1 is equal to 1

subfunction0:
..
```

### 9.4.6 `ret`: Return from Subroutine

#### 9.4.6.1 Calling Format

```
ret
```

#### 9.4.6.2 Operands

None

### 9.4.6.3 Overview

Jumps to the caller of a call instruction. This instruction does nothing if control is not in the middle of a jump from a call instruction (unlike `nop`, no processing occurs). To call a label as a subroutine from a call instruction, this instruction must be called after the label is set.

### 9.4.6.4 Operation

```
if ( retaddr )
    pc = retaddr
```

### 9.4.6.5 Example

```
callb    b0 , subfunction0

subfunction0:
..
ret
```

---

## 9.4.7 ifb: Start if Block by Boolean

### 9.4.7.1 Calling Format

```
ifb      src
```

### 9.4.7.2 Operands

- `src`: Boolean register

### 9.4.7.3 Overview

Executes conditional processing based on the content of the Boolean register specified by `src`. When it is `true`, the instructions between `ifb` and `endif` are executed. If there is an `else` instruction between `ifb` and `endif`, the instructions between `ifb` and `else` are executed. When it is `false`, the instructions between `ifb` and `endif` are skipped and control moves to the instruction immediately after `endif`. If there is an `else` instruction between `ifb` and `endif`, the instructions between `else` and `endif` are executed. This instruction must be followed by an `endif` instruction. You can nest up to eight `ifb` and `ifc` instructions. You must denote at least one instruction between `ifb` and `endif` as well as between `ifb` and `else`.

### 9.4.7.4 Operation

```
if ( src == true )
{
    ...
}
```

### 9.4.7.5 Example

```
ifb      b0
...
endif
```

## 9.4.8 ifc: Start if Block by Condition

### 9.4.8.1 Calling Format

```
ifc      status0 , status1 , mode
```

### 9.4.8.2 Operands

- status0: Value (either 0 or 1) of status register 0
- status1: Value (either 0 or 1) of status register 1
- mode: Conditional mode
  - 0: OR
  - 1: AND
  - 2: Only status register 0
  - 3: Only status register 1

### 9.4.8.3 Overview

Runs conditional processing based on the status register values.

The equality of the values specified by `status0` (or `status1`) and status register 0 (or 1) is taken to be the condition. This condition is `true` when either status register 0 or 1 match when `mode` is 0; when both status registers match when `mode` is 1; when status register 0 matches when `mode` is 2; and when status register 1 matches when `mode` is 3.

When the condition is `true`, the instructions between `ifc` and `endif` are executed. If there is an `else` instruction between `ifc` and `endif`, the instructions between `ifc` and `else` are executed. When the condition is `false`, the instructions between `ifc` and `endif` are skipped and control moves to the instruction immediately after `endif`. If there is an `else` instruction between `ifc` and `endif`, the instructions between `else` and `endif` are executed. This instruction must be followed by an `endif` instruction. You can nest up to eight `ifb` and `ifc` instructions. You must denote at least one instruction between `ifc` and `endif` as well as between `ifc` and `else`.

### 9.4.8.4 Operation

```
switch ( mode )
{
case 0 :
    if ( status0 == Status_register0 || status1 == Status_register1 )
        condition = true
    break;
case 1 :
    if ( status0 == Status_register0 && status1 == Status_register1 )
        condition = true
    break;
case 2 :
    if ( status0 == Status_register0 )
        condition = true
```

```
    break;
case 3 :
    if ( status1 == Status_register1 )
        condition = true
    break;
}
if ( condition == true )
{
    ...
}
```

#### 9.4.8.5 Example

```
ifc      1 , 1 , 0      // Runs the instructions between ifc and else when
...                               // status register 0 or status register 1 is equal to 1
else
    ...
endif
..
```

### 9.4.9 else: Start else Block

---

#### 9.4.9.1 Calling Format

```
else
```

#### 9.4.9.2 Operands

None

#### 9.4.9.3 Overview

This is used in combination with `ifc` or `ifb`. When the `if` statement is `true`, processing runs until this instruction and then skips all instructions until the next `endif`. When the `if` statement is `false`, processing skips from the `if` instruction to this one and then runs all instructions between this and `endif`. You must denote at least one instruction between `ifb` and `else`, between `ifc` and `else`, and between `else` and `endif`.

#### 9.4.9.4 Operation

```
if ( src == true )
{
    ...
}
else
{
    ...
}
```

```

}
endif

```

#### 9.4.9.5 Example

```

ifb      b0
...
else
...
endif

```

### 9.4.10 endif: End if Block

#### 9.4.10.1 Calling Format

```
endif
```

#### 9.4.10.2 Operands

None.

#### 9.4.10.3 Overview

Ends a control block started by `ifc` or `ifb`. You must denote at least one instruction between `ifb` and `endif`, between `ifc` and `endif`, and between `else` and `endif`.

#### 9.4.10.4 Operation

```

if ( src == true )
{
    ...
}
else
{
    ...
}
endif

```

#### 9.4.10.5 Example

```

ifb      b0
...
else
...
endif

```

### 9.4.11 Loop: Start Loop Statement

#### 9.4.11.1 Calling Format

```
loop      src
```

### 9.4.11.2 Operands

- **src:** Integer register

### 9.4.11.3 Overview

This is used together with `endloop`. It repeatedly runs the instructions between `loop` and `endloop` according to the content of the integer register specified by `src`. The integer register comprises a loop count, an initial value for the loop-counter register, and an amount by which to increment the loop-counter register. The loop-counter register (`aL`) is initialized when the `loop` instruction is executed. When control reaches `endloop`, the loop-counter register is incremented by the increment amount and control returns to the `loop` instruction. After this process repeats one time more than the loop count in the integer register, the `loop` instruction ends and the next instruction after `endloop` is executed. Up to four `loop` instructions can be nested. If the loop-counter register is incremented by a negative number, its value could become 0 or less but will actually be set to 255 by an underflow. Behavior is undefined if a floating-point constant register is offset by a value of 96 or greater. You must denote at least one instruction between `loop` and `endloop`.

### 9.4.11.4 Operation

```
for ( int i = 0, aL = src.init ; i < src.count+1 ; i++, aL += src.step )
{
    ...
}
```

### 9.4.11.5 Example

```
defi      i0 , 10 , 0 , 1

loop      i0
add       r0 , r0 , c0[ aL ]    // Adds the total value of c0-c10 to r0
endloop
```

---

## 9.4.12 `endloop`: End Loop Statement

### 9.4.12.1 Calling Format

```
endloop
```

### 9.4.12.2 Operands

None

### 9.4.12.3 Overview

Specifies the location to end a loop controlled by the `loop` instruction. You must denote at least one instruction between `loop` and `endloop`.

### 9.4.12.4 Operation

```
for ( int i = 0, aL = src.init ; i < src.count+1 ; i++, aL += src.step )
{
```

```

    ...
}

```

#### 9.4.12.5 Example

```

defi      i0 , 10 , 0 , 1

loop      i0
add       r0 , r0 , c0[ aL ]      // Adds the total value of c0-c9 to r0
endloop

```

### 9.4.13 breakc: Break from Loop Statement by Condition

#### 9.4.13.1 Calling Format

```
breakc      status0 , status1 , mode
```

#### 9.4.13.2 Operands

- status0: Value (either 0 or 1) of status register 0
- status1: Value (either 0 or 1) of status register 1
- mode: Conditional mode
  - 0: OR
  - 1: AND
  - 2: Only status register 0
  - 3: Only status register 1

#### 9.4.13.3 Overview

Forcibly exits a loop control block based on the status register values.

The equality of the values specified by `status0` (or `status1`) and status register 0 (or 1) is taken to be the condition. This condition is `true` when either status register 0 or 1 match when `mode` is 0; when both status registers match when `mode` is 1; when status register 0 matches when `mode` is 2; and when status register 1 matches when `mode` is 3.

When the condition is `true`, control is forced to exit a loop started by a `loop` instruction and then jumps to the next instruction following `endloop`.

#### 9.4.13.4 Operation

```

for ( int i = 0, aL = src.init ; i < src.count+1 ; i++, aL += src.step )
{
    ...
    switch ( mode )
    {
    case 0 :
        if ( status0 == Status_register0 || status1 == Status_register1 )
            condition = true
        break;

```



```
case 1 :
    if ( status0 == Status_register0 && status1 == Status_register1 )
        condition = true
    break;
case 2 :
    if ( status0 == Status_register0 )
        condition = true
    break;
case 3 :
    if ( status1 == Status_register1 )
        condition = true
    break;
}
if ( condition == true )
    break;
}
```

#### 9.4.13.5 Example

```
defi      i0 , 10 , 0 , 1

loop      i0
add       r0 , r0 , c0[ aL ]    // Adds the total value of c0-c9 to r0
breakc    1, 0, 2              // Break if status register 0 is equal to 1
endloop
```

### 9.4.14 **cmp**: Compare

---

#### 9.4.14.1 Calling Format

```
cmp      mode0 , mode1 , src0 , src1
```

#### 9.4.14.2 Operands

- mode0: Comparison mode 0
  - 0: ==
  - 1: !=
  - 2: <
  - 3: <=
  - 4: >
  - 5: >=
- mode1: Comparison mode 1
  - 0: ==
  - 1: !=
  - 2: <

- 3: <=
  - 4: >
  - 5: >=
- `src0`: Temporary register, input register, or floating-point constant register
  - `src1`: Temporary register, input register, or floating-point constant register

#### 9.4.14.3 Overview

Compares the content of registers `src0` and `src1` and stores the result in the status registers.

Only the x and y components (after swizzling) of `src0` and `src1` are compared. Status register 0 stores the result of comparing the x components with the condition specified by `mode0`. Status register 1 stores the result of comparing the y components with the condition specified by `mode1`.

A status register is set to 1 if the following comparison results are true and 0 if they are false.

- `src0 == src1` when `mode0` (or `mode1`) is 0
- `src0 != src1` when `mode0` (or `mode1`) is 1
- `src0 < src1` when `mode0` (or `mode1`) is 2
- `src0 <= src1` when `mode0` (or `mode1`) is 3
- `src0 > src1` when `mode0` (or `mode1`) is 4
- `src0 >= src1` when `mode0` (or `mode1`) is 5

Both status register 0 and 1 are always updated. You cannot update only one of them.

You cannot specify a floating-point constant register for both `src0` and `src1`. You cannot specify input registers with different indices in `src0` and `src1` at the same time.

#### 9.4.14.4 Operation

```
switch ( mode0 )
{
case 0: status_register0 = ( src0.x == src1.x ) ? 1 : 0 ; break ;
case 1: status_register0 = ( src0.x != src1.x ) ? 1 : 0 ; break ;
case 2: status_register0 = ( src0.x < src1.x ) ? 1 : 0 ; break ;
case 3: status_register0 = ( src0.x <= src1.x ) ? 1 : 0 ; break ;
case 4: status_register0 = ( src0.x > src1.x ) ? 1 : 0 ; break ;
case 5: status_register0 = ( src0.x >= src1.x ) ? 1 : 0 ; break ;
}
switch ( mode1 )
{
case 0: status_register1 = ( src0.y == src1.y ) ? 1 : 0 ; break ;
case 1: status_register1 = ( src0.y != src1.y ) ? 1 : 0 ; break ;
case 2: status_register1 = ( src0.y < src1.y ) ? 1 : 0 ; break ;
case 3: status_register1 = ( src0.y <= src1.y ) ? 1 : 0 ; break ;
case 4: status_register1 = ( src0.y > src1.y ) ? 1 : 0 ; break ;
}
```

```
case 5: status_register1 = ( src0.y >= src1.y ) ? 1 : 0 ; break ;  
}
```

#### 9.4.14.5 Example

```
def      c0 , 0, 1, 2, 3  
  
mov      r0, c0  
cmp      0 , 0 , r0, c0  
ifc      1 , 1 , 2  
    // This is executed when r0.x == 0  
endif
```

### 9.4.15 end: End Process

---

#### 9.4.15.1 Calling Format

```
end
```

#### 9.4.15.2 Operands

None

#### 9.4.15.3 Overview

Ends vertex shader processing. This instruction must be called after a vertex shader has finished writing all output data. If this instruction is called before all output data has finished being written, the output data will be undefined. This instruction uses two instructions' worth of the program size.

## 10 Debug Build

Specifying the `-debug` option to `ctr_VertexShaderAssembler32.exe` will result in a debug build. Although you can use the shader debugger to debug assembler objects created by a debug build, the objects may run more slowly in POD. If the `-debug` option is specified to `ctr_VertexShaderLinker32.exe`, all linked assembler objects will be forced to use a debug build. If the `-nodebug` option is specified to `ctr_VertexShaderLinker32.exe`, all linked assembler objects will be forced to not use a debug build.

When you link assembler objects created by a debug build with ones that are not, each of the main objects will use a debug build when it references at least one assembler object that does.

When you do not specify the `-debug` option to `ctr_VertexShaderAssembler32.exe`, the input file path is removed from the object file. Without the full input file path, the shader debugger is sometimes unable to find the source file.

# 11 Map Files

## 11.1 Overview

---

Specifying the `-M` option to `ctr_VertexShaderLinker32.exe` when linking executable files will cause files to be output with information on the executable files. These are called *map files*. A map file is created with the same name as its executable file but uses the extension `.map`.

The following information is generated: *loading objects order*, *image sizes*, and *object information*. The following sections provide more details.

## 11.2 Loading Objects Order

---

This item shows the order in which the main objects were linked, which is the same as the order specified as an argument to `ctr_VertexShaderLinker32.exe`. The object order indicated here is the same as the individual reference points for shader objects specified with the `glShaderBinary` function. This also shows which main objects use a debug build.

## 11.3 Image Sizes

---

This item shows the data size of the linked object files. `Instruction` indicates the number of assembly code instructions and `Swizzle` indicates the number of swizzling and masking patterns (for details, see section 12.3 Pattern Counts for Swizzling and Masking). `Total` indicates the entire size after linking.

## 11.4 Program Code Information

---

This item shows the location used to store the program code in the executable file. `Program code offset` is the offset (in bytes) from the start of the executable file to the address at which the program code's data is stored. `Program code size` is the number of bytes of program code's data.

## 11.5 Object Information

---

This item shows individual symbol information, output data attribute information, and the starting program address for the linked main objects. These settings are specified using `#pragma bind_symbol` and `#pragma output_map`, and the program address is set by the `main` label, respectively.

## 11.6 Swizzle Pattern Data

---

This item shows the swizzle pattern data in the executable file.

## 12 Precautions and Restrictions

Vertex shaders have the following limitations due to characteristics of the hardware.

### 12.1 Starting and Ending a Shader

A shader starts processing from the `main` label. If all components (x, y, z, and w) are written to the output registers specified by `#pragma output _map` and the `end` instruction is called, processing ends and then restarts for the next vertex. The shader does not run properly if values are not written to the registers specified by `#pragma output _map`. You must explicitly call the `end` instruction at the end of processing.

Once data has been written to every output register, all required processing is recognized to be complete. It is therefore uncertain whether instructions will run after the last instruction that writes to the output registers. Normal operations might not result when calling an instruction to read or write registers after the last instruction to write to the output registers. Do not call any instructions other than `nop` between the instruction for the last write to output registers and the `end` instruction.

You can only write to an output register once. Behavior is not guaranteed if you write to an output register more than once. This also applies to writing to each component.

If you don't read an input register at least once during processing of a single vertex, the shader might not behave normally. Be sure to execute at least one read instruction on at least one component of any input register.

### 12.2 Step Count

Programs have a maximum step count of 512.

The `def`, `defi`, `defb`, `ret`, `else`, `endif`, and `endloop` instructions are not calculated as step counts.

### 12.3 Pattern Counts for Swizzling and Masking

There is a maximum number of patterns for the combination of masking, replacing (swizzling), and adding signs to input components. This upper limit is 128, of which no more than 32 patterns are usable with the `mad` instruction, as well.

#### Code 12-1 Pattern Count Example 1

```
add    r0 , r1.xy , -r2.zw
add    r2 , r0.x, r3
mul    r3 , r2.xy , -r3.zw
add    r4 , r2.xxxx, r5.xyzw
```

The first and third instructions used here have the same pattern. The second and fourth instructions also have the same pattern, so the combined pattern count is 2.

#### Code 12-2 Pattern Count Example 2

```
add      r0 , r1.xy , -r2.zw
mul      r2.xy , r0.x , c2.y
mad      r3 , r2.xy , -r3.zw , r1.w
cmp      0 , 1 , r1.x , c0.y
```

The first and third instructions used here are considered to have the same pattern because the combination for the third instruction (`mad`) is the same as the combination for the first instruction (`add`) except for the `src2` operand. This pattern is treated as being usable by the `mad` instruction, as well.

The combination of `src0` and `src1` in the fourth instruction (`cmp`) is the same as the combination of `src0` and `src1` in the second instruction (`mul`), so the second and fourth instructions are considered to have the same pattern.

## 12.4 Control Instruction Limitations

---

You must call an ending control instruction after a starting control instruction. Specifically, you must use the following combinations.

- `ifb (-else)-endif`
- `ifc (-else)-endif`
- `call-ret`
- `callb-ret`
- `callc-ret`
- `loop-endloop`

It is illegal to jump outside of these control blocks using the `jpc` or `jpb` jump instructions. You also cannot call `ret` from within an `if` or `loop` block.

When `call` instructions are nested, behavior is undefined when another `call` instruction is called immediately before a `ret` instruction within a subroutine.

You cannot use the `jpc` or `jpb` instruction between the `main` and `endmain` labels to jump to an external location. You also cannot call the instructions from a subroutine to jump to an external location. You cannot jump to a `ret` instruction within a subroutine. Behavior is undefined for all of these controls.

## 12.5 Instructions That Cannot Be Called Consecutively

---

The `else`, `endif`, `ret`, and `endloop` instructions cannot be called consecutively.

#### Code 12-3 Instructions That Cannot Be Called Consecutively

```
ifb      b0
```

```

nop
nop
ifb    b1
    nop
else
    nop
    nop
endif          // Error
else
    nop
    nop
    call    subroutine    // This causes an error because ret is called at the end
                        // of the call destination
endif

```

In the same way, the `movb` instruction cannot be called consecutively.

## 12.6 Registers That Cannot Be Used Simultaneously

In general, you cannot specify two or more floating-point constant registers for assembly code instructions that specify two or more `src` operands. You also cannot specify two or more input registers. However, you can specify any number of input registers if they have the same index.

### Code 12-4 Registers That Cannot Be Used Simultaneously

```

add r0 , c0 , c0          // Error
add r0 , c0 , c1          // Error
add r0 , v0 , v0          // This is not an error.
add r0 , v0 , v1          // Error

```

Macro instructions are checked for errors after they have been expanded.

## 12.7 Instruction Latency

The following table shows the latency of running each instruction.

**Table 12-1 Instruction Latency**

Instruction	Latency (in clock cycles)
add	3
dp3	5



Instruction	Latency (in clock cycles)
dp4	5
dph	5
dst	3
exp	4
flr	2
litp	2
log	4
mad	4
max	2
min	2
mov	2
mova	4
mul	3
nop	1
rcp	4
rsq	4
sge	2
slt	2

Instruction	Latency (in clock cycles)
<code>cmp</code>	4
Other branch instructions	3 or 1

### 12.7.1 Arithmetic and `cmp` Instruction Latency

Although the previous table gives the approximate number of clock cycles for the latency of arithmetic instructions and the `cmp` instruction, these values may change depending on the preceding and following instructions. Latency may be reduced by queueing up instructions that use unrelated registers for their calculations.

### 12.7.2 Branch Instruction Latency

The previous table gives a latency of 3 or 1 for branch instructions. The latency is 1 when a branch increments the program counter by one and is 3 in all other cases.

### 12.7.3 Output Order of Calculation Results

Instruction results are never written to a register before the results of an earlier instruction. This is guaranteed even when low-latency instructions are executed after high-latency instructions.

#### Code 12-5 Output Order of Calculation Results

```
exp r0 , r1.x
mov r1 , c0
```

When this code is run, `exp` takes four clock cycles and `mov` takes two clock cycles. However, the `mov` result is never written to `r1` before `exp` finishes. In this case, `mov` stalls while reading from the register and thus delays execution. This is not unique to `mov`: other instructions stall while reading from a register in similar situations.

### 12.7.4 Stalls Due to Conflicts When Outputting Calculation Results

If a low-latency instruction is executed after a high-latency instruction and they both complete simultaneously, the results of the low-latency instruction are output after one clock cycle. Multiple registers are never written simultaneously.

#### Code 12-6 Simultaneous Instruction Completion

```
exp r0 , r1.x
mul r2 , c3 , r4
```

When this code is run, `exp` takes four clock cycles and `mul` takes three clock cycles, such that the results of both `r0` and `r2` would be output simultaneously. However, to avoid multiple register writes at once, the output of `r2` is delayed by one clock cycle.

### 12.7.5 Stalls Due to Conflicts Among Arithmetic Units

---

The `mad`, `dp3`, `dp4`, `dph`, and `add` instructions all require use of the adder. If these instructions are executed in sequence, the adder pipeline may get backed up, with instructions executed later waiting for previously executed instructions to finish, causing higher latency.

The adder is used during the first cycle of the `add` instruction, the second cycle of the `mad` instruction, and the second and third cycles of the `dp3`, `dp4`, and `dph` instructions.

### 12.7.6 Stalls Due to Instruction Dependencies

---

Dependencies between issued instructions may cause stalls. This phenomenon occurs when an instruction stores its calculated results in a register that is used as a source register by the next instruction, as shown in the following code.

#### Code 12-7 Register Use Causing a Stall

```
add r0, r1, r2
mul r4, r0, r3
```

When run, the preceding code stalls because the results output to `r0` are used by the next instruction. This code would not stall, however, if the register is the same but the components are different.

#### Code 12-8 Using Different Register Components Avoids a Stall

```
add r0.x, r1, r2
mul r4, r0.y, r3
```

The preceding code does not stall because the results output to `r0.x` are not used by the next instruction. Register `r0` is used consecutively but with different components, so it does not satisfy the conditions for stalling.

## 12.8 Results of Exceptional Operations

---

The vertex shader exhibits the following behavior as the result of calculating an exceptional operation.

- NaN is output for the logarithm of a negative value or -0.
- NaN is output for the square root of a negative value or -0.
- NaN is output for operations that use NaN as an input value (except for the `cmp` instruction).
- Negative infinity ( $-\infty$ ) is output when infinity ( $\infty$ ) is subtracted from a number.
- Negative infinity ( $-\infty$ ) is output for the logarithm of a non-normalized number (a number with an exponent of 0 and a nonzero significand) or +0.
- Infinity ( $\infty$ ) is output when there is an overflow.
- Negative infinity ( $-\infty$ ) is output when there is an underflow.
- Infinity ( $\infty$ ) or negative infinity ( $-\infty$ ) is output for a division by positive or negative 0.

## 12.9 Limitations Related to Invalid Data Output

Behavior is not guaranteed when a vertex attribute value of NaN (Not a Number) is output from the vertex shader (written to an output register). Do not give NaN as an input vertex attribute or uniform value from the application, nor as the calculation result output from a vertex shader.

## 12.10 Shader Implementations That Cause Invalid Operations

The order in which shader assembler instructions are executed can cause the hardware to carry out invalid operations. This can cause the GPU to hang. The shader assembler or linker will output warnings if a shader is implemented in a way that could cause invalid operations.

The following sections describe the kinds of conditions that could cause invalid operations.

### 12.10.1 Invalid Operation due to a `mov` Instruction

Invalid operations can be caused by any number of conditions. This section describes invalid operations specifically caused by a `mov` instruction.

Some of the conditions documented here are understood completely, while other conditions are highly suspect. Use the provided information to avoid known issues and troubleshoot current problems, but understand that your implementation of other portions of the shader may introduce results heretofore not seen. Factors to consider include the types of instructions preceding what appears to be the offending instruction, the combination of registers used, and branch instructions in registers. Factors to ignore include execution timing and register contents other than branch instructions. Ultimately, a successful shader implementation is determined by trouble-free behavior as exhibited on a production unit.

#### 12.10.1.1 Executing a `mov` Instruction as the Second-to-Last Instruction

Shaders that execute a `mov` instruction as the second-to-last instruction with no dependence on the final instruction can cause the hardware to hang. For vertex shaders, the last instruction executed is the last one written to the register. For geometry shaders, the last instruction specifies an `end` instruction.

##### Code 12-9 Vertex Shader Causing Invalid Operations

```
// Vertex Shader 1
mov a0.x, r0    // Final instruction right after mov.
mov o0, r2      // Final instruction does not use a0.x, raising risk of a hang.
end
```

##### Code 12-10 Vertex Shader Not Causing Invalid Operations

```
// Vertex Shader 2
mov a0.x, r0    // Final instruction right after mov.
mov o0, c[a0.x] // However, final instruction does use a0.x, so no problem.
end
```

**Code 12-11 Geometry Shader Causing Invalid Operations**

```
// Geometry Shader
mov a0.x, r0      // Execute mova.
end               // End right after mova raises risk of a hang.
```

The shader assembler outputs warning 400a0003 whenever a `mova` instruction comes just before the `end` instruction. The shader assembler outputs warning 400a0004 whenever a `mova` instruction is followed by an instruction to write to an output register and then by an `end` instruction, and the register written to by the `mova` instruction is not used by the next instruction. Avoid this set of circumstances by taking steps such as inserting a `nop` instruction just before the final instruction, or changing the order of instructions.

**12.10.1.2 Executing `mova` Instructions Just Before and Just After Certain Other Instructions**

Combinations of the `mova` instruction with the `else-endif`, `call-ret`, and `loop-endloop` instructions can cause the hardware to hang. This section describes all three combinations.

The hardware might hang for an `ifb` or `ifc` code clause when there is a `mova` instruction both right before the `else` and right after the `endif`. The shader assembler outputs warning 40070003 when it detects such an implementation. You can avoid this sequence by taking steps such as inserting a `nop` instruction just before the final instruction or changing the order of instructions.

**Code 12-12 `else-endif` Clause Causing Invalid Operations**

```
ifb b0
...
    mova a0.x, r0 // Executes a mova just before the else. Jumps to just after the
endif after executing.
else
...
endif
mova a0.x, r1     // This can cause a hang when jumping from just before the else.
```

The hardware might hang for a `call`, `callb`, or `callc` code clause when there is a `mova` instruction both right before the final `ret` of the called subroutine and right after the subroutine returns. The shader assembler outputs warning 4009000c when it detects such an implementation. You can avoid this sequence by taking steps such as inserting a `nop` instruction just before the final instruction or changing the order of instructions.

**Code 12-13 `call-ret` Clause Causing Invalid Operations**

```
main:           // main function.
...
call l_function // Jump to l_function.
mova a0.x, r0    // mova again right after returning from l_function.
...
```

```

end                // main function ends.

l_function:        // Subroutine.
...
mov a0.x, r1       // mova just before ret.
ret

```

The hardware might hang when there is a `mova` instruction both right after a `loop` instruction and right before an `endloop` instruction. The shader assembler outputs warning 40070004 when it detects such an implementation. You can avoid this sequence by taking steps such as inserting a `nop` instruction just before the final instruction or changing the order of instructions.

#### Code 12-14 loop-endloop Clause Causing Invalid Operations

```

loop i0
    mova a0.y, r0 // mova right after loop.
    ...
    mova a0.x, r1 // mova right before endloop.
endloop

```

#### 12.10.1.3 Stalling on a `mova` Instruction and Branching Right Afterwards

The hardware might hang when executing a `mova` instruction that a) stalls due to a dependence on the preceding instruction and b) is immediately followed by a branch instruction. Branch instructions are `jpb`, `jpc`, `call`, `callb`, `callc`, `ifb`, `ifc`, and `breakc`.

#### Code 12-15 `mova` Followed by a Branch Instruction Causing Invalid Operations

```

dp4 r0, r1, r2      // Write to r0.
mova a0.x, r0.x      // r0 depends on dp4, so this stalls.
call l_function      // Branch instruction right after a stalling mova.

```

The shader assembler outputs warning 400a0005 when it detects such an implementation (a branch instruction following a `mova` instruction that has a temporary register as its source). You can avoid this sequence by taking steps such as inserting a `nop` instruction just before the final instruction or changing the order of instructions.

### 12.10.2 Invalid Operation due to a Specific Order of Instructions

The hardware carries out invalid operations when four instructions in a row meet all of the following conditions. These combinations have repeatedly been performed to successfully reproduce invalid operations.

1. The first and third instructions are latency-2 instructions (`flr`, `litp`, `max`, `min`, `mov`, `sge`, `slt`, `abs`).

2. The second instruction is a latency-2 or lower instruction (`flr`, `litp`, `max`, `min`, `mov`, `sge`, `slt`, `abs`, `nop`).
3. The fourth instruction is a branch instruction (`jpb`, `jpc`, `call`, `callb`, `callc`, `ifb`, `ifc`, `breakc`).
4. The first instruction stalls. This stall must be for at least two clock cycles when the second instruction is `nop`, and at least three clock cycles when the second instruction is not `nop`.
5. The first and second instructions do not depend on each other.
6. The second and third instructions do not depend on each other.
7. The first and third instructions do depend on each other.
8. One of the `dst` and one of the `src` operands for the third command must be the same.

The dependencies in conditions 5, 6, and 7 above describe when one of the `dst` operands of the earlier instruction is the same register and same component as one of the `src` operands of the later instruction. Having the same register index but different components is not a dependency.

The dependency in condition 8 above describes when the `dst` and `src` operands have the same register index regardless of the component. In other words, this would apply to `r0` and `r0`, or `r1.x` and `r1.y`.

The following code shows an example of this situation.

#### Code 12-16 Instruction Ordering that Causes Invalid Operations

```
rcp r1, r2.x           // First instruction causes a stall.
min r0, r1, r2         // First instruction of the four that meet these
                        // conditions.
max r3, r4, r5         // Second instruction.
slt r5, r0, r5         // Third instruction.
call l_function        // Fourth instruction.
```

The first instruction of this chain is `min`, and the third is `slt`, both latency-2 instructions, thus meeting condition 1. The second instruction is `max`, also latency-2 and thus fulfilling condition 2. The fourth instruction is `call`, meeting condition 3. The `min` instruction waits for register `r1` to be written to by the `rcp` instruction, causing a stall for three clock cycles and fulfilling condition 4. `min` and `max` are not dependent, meeting condition 5. `max` and `slt` are not dependent, meeting condition 6. `min` and `slt` are dependent, meeting condition 7. Last, the `dst` and `src1` operands are the same for `slt`, meeting condition 8.

The shader assembler outputs warning 400a0001 or 400a0002 when it detects an implementation that meets all these conditions except condition 4.

The shader assembler has no means of telling whether the first instruction stalls, and so cannot evaluate condition 4. It is possible to get a rough estimate of the length of any stall by using the performance checking feature of the shader linker, but this is not a conclusive evaluation. Stalling conditions depend on the instructions executed previously and how registers are used. Shaders

either definitely cause invalid operations, or definitely do not cause invalid operations. A shader that operates properly even when the shader assembler outputs a warning must therefore not meet condition 4, and is thus safe to continue using. If a shader does cause invalid operations, avoid this sequence by taking steps such as inserting a `nop` instruction just before the final instruction or changing the order of instructions.



## 13 Error Messages for the Assembler and Linker

### 13.1 Overview

---

This chapter describes the error messages output by the assembler and linker. Errors are output in the following format.

Input filename (error line number): Error level (Error Code): Error description

The error level is either *warning* or *error*. Processing can continue when there is a warning. The input filename, error line number, and other information may not be displayed for some types of errors.

### 13.2 Assembler Error Messages

---

This section describes the errors and error codes output by the assembler.

#### 13.2.1 80010001

---

```
(80010001): -O option cannot be specified more than once.
```

The -O option cannot be specified more than once.

#### 13.2.2 80010003

---

```
(80010003): Definition key is not specified with -D option
```

The key to define for the -D option has not been specified.

Specify it in the format "-Dkey" or "-Dkey=value".

#### 13.2.3 80010004

---

```
(80010004): Definition value is not specified with 'argument name' macro.
```

A value has not been correctly set for the definition macro with the -D option.

Specify it in the format "-Dkey" or "-Dkey=value".

#### 13.2.4 80010005

---

```
(80010005): 'argument name' include illegal character.
```

You attempted to define a macro for the -D option using illegal characters.

Use single-byte alphanumeric characters and underscores for macro names.

### 13.2.5 80010006

```
(80010006): 'Macro name' macro is redefinition.
```

More than one macro with the same name has been defined with the -D option.

### 13.2.6 80010007

```
(80010007): Only one assembler file can be specified as input.
```

More than one assembler file has been specified as an input file. Specify only one assembler file.

### 13.2.7 80010008

```
(80010008): Input file is not specified.
```

An assembler file has not been specified as an input file.

### 13.2.8 8001000b

```
(8001000b): 'Macro name' macro name cannot start from number.
```

You cannot use a number as the first character of a macro with the -D option.

### 13.2.9 80030001

```
(80030001): Cannot open 'filename'
```

Could not open the specified assembler file.

### 13.2.10 80030002

```
(80030002): Include file name is not specified.
```

A filename has not been specified with an #include statement.

Specify it in the format "#include 'filename'".

### 13.2.11 80030003

```
(80030003): Syntax error in #include
```

An #include statement was denoted incorrectly.

Specify it in the format "#include 'filename'".

**13.2.12 80030004**

```
(80030004): Cannot open include file "filename".
```

Could not open an include file.

Specify the include path using the -I option.

**13.2.13 80030005**

```
(80030005): Definition key is not specified.
```

A `#define` statement was denoted incorrectly.

Use the format `"#define key value"`.

**13.2.14 80030006**

```
(80030006): Definition key include illegal character.
```

Use single-byte alphanumeric characters and underscores for macro names defined by `#define` statements.

**13.2.15 80030007**

```
(80030007): 'Macro name' macro is redefined.
```

Duplicate macros have been defined by `#define` statements.

**13.2.16 80030008**

```
(80030008): Definition key is not specified.
```

An `#undef` statement was denoted incorrectly.

Use the format `"#undef key"`.

**13.2.17 8003000b**

```
(8003000b): Correspondent "#ifdef" is not found.
```

An `#endif` statement is missing a corresponding `#ifdef` statement.

**13.2.18 8003000c**

```
(8003000c): Undefined directive.
```

An unsupported preprocessor pseudo-instruction has been specified.

### 13.2.19 8003000d

```
(8003000d): #ifdef is not closed.
```

An `#ifdef` statement is missing a corresponding `#endif` statement.

### 13.2.20 8003000e

```
(8003000e): Syntax error. Macro is not specified.
```

A macro has not been specified for an `#ifdef` statement.

Specify the format `"#ifdef macro"`.

### 13.2.21 8003000f

```
(8003000f): Syntax error. Invalid string is detected after macro.
```

An `#ifdef` statement was entered incorrectly.

Specify the format `"#ifdef macro"`.

### 13.2.22 80030010

```
(80030010): Syntax error. Macro is not specified.
```

A macro has not been specified for an `#ifndef` statement.

Specify the format `"#ifndef macro"`.

### 13.2.23 80030011

```
(80030011): Syntax error. Invalid string is detected after macro.
```

An `#ifndef` statement was entered incorrectly.

Specify the format `"#ifndef macro"`.

### 13.2.24 80030012

```
(80030012): Syntax Error. Invalid string is detected after directive.
```

Use single-byte alphanumeric characters and underscores for macro names specified by `#if`, `#ifdef`, and `#ifndef` statements.

### 13.2.25 80030013

```
(80030013): Syntax Error. Invalid string is detected after directive.
```

An invalid string was detected after an `#else` statement.

### 13.2.26 80030014

```
(80030014): Correspondent "#ifdef" is not found.
```

Could not find an `#ifdef` statement corresponding to an `#else` statement.

### 13.2.27 80030015

```
(80030015): Syntax error. Invalid expression is detected.
```

An `#if` statement was denoted incorrectly.

### 13.2.28 80030017

```
(80030017): Syntax error. Invalid expression is detected.
```

A macro entered after an `#if` statement uses invalid characters.

Use single-byte alphanumeric characters and underscores for macro names specified with `#if` statements.

### 13.2.29 80030018

```
(80030018): #error
```

This error is intentionally output by an `#error` statement.

### 13.2.30 80030019

```
(80030019): The top character of definition key must not be number.
```

You cannot use a number as the first character of a macro name defined using a `#define` statement.

### 13.2.31 8003001a

```
(8003001a): Macro parentheses have not been closed properly.
```

Parentheses are not denoted correctly in a definition using the `#define` statement.

### 13.2.32 8003001b

```
(8003001b): Invalid character is detected in macro argument.
```

One of the arguments to a function macro uses an illegal string.

Use single-byte alphanumeric characters and underscores for macro names.

### 13.2.33 8003001c

```
(8003001c): Duplicate macro argument is detected.
```

The same string is used more than once in the arguments to a function macro.

### 13.2.34 8003001d

```
(8003001d): Invalid macro argument is specified.
```

An incorrectly defined macro function was used.

### 13.2.35 8003001e

```
(8003001e): pragma command bind_symbol is invalid format..
```

The `#pragma bind_symbol` statement was denoted incorrectly.

### 13.2.36 8003001f

```
(8003001f): Undefined pragma command.
```

An unsupported `pragma` command was specified.

### 13.2.37 80030020

```
(80030020): Start index should be less than or equal to end index.
```

The starting register index is larger than the ending register index in a `#pragma bind_symbol` statement.

### 13.2.38 80030021

```
(80030021): Binding symbol name is duplicated.
```

Duplicate symbol names are defined by `#pragma bind_symbol` statements.

### 13.2.39 80030022

```
(80030022): Invalid register index is specified.
```

An invalid register index has been defined by a `#pragma bind_symbol` statement.

(The maximum number of registers was exceeded.)

### 13.2.40 80030023

(80030023): Specified registers are already bound to other symbol.

The same input register was bound to more than one symbol by `#pragma bind_symbol` statements.

An input register corresponds to a single register and cannot be bound to more than one symbol name.

### 13.2.41 80030024

(80030024): Pragma command `output_map` is invalid format.

The `#pragma output_map` statement was denoted incorrectly.

### 13.2.42 80030025

(80030025): Invalid data name is specified for pragma command `output_map`.

An invalid data attribute name has been specified in a `#pragma output_map` statement.

### 13.2.43 8003002c

(8003002c): Specified register is already mapped.

The register specified by a `#pragma output_map` statement has already been specified by another `#pragma output_map` statement.

### 13.2.44 8003002d

(8003002d): Specified attribute is already mapped.

The data attribute name specified by a `#pragma output_map` statement has already been specified by another `#pragma output_map` statement.

### 13.2.45 80030033

(80030033): If all textures are mapped, `texture1` and `texture2` need to be mapped to same register.

`texture1` and `texture2` must be mapped to the same register when all textures have been defined by `#pragma output_map` statements.

### 13.2.46 80030034

(80030034): comment `/* */` is not closed.

A comment of the `/* ... */` type has not been closed properly.

### 13.2.47 80040001

(80040001): No vertex shader instruction.

No shader instruction has been denoted.

### 13.2.48 80040005

(80040005): loop instruction is not closed by endloop.

A loop instruction is missing a corresponding endloop instruction.

### 13.2.49 80040007

(80040007): if or else instruction is not closed by endif.

An ifc or ifb instruction is missing a corresponding endif instruction.

### 13.2.50 80040009

(80040009): Unknown instruction.

An unknown shader instruction was denoted.

### 13.2.51 8004000c

(8004000c): The number of operand is short.

There are not enough operands.

### 13.2.52 8004000d

(8004000d): There are some extra operand.

Too many operands have been specified.

### 13.2.53 8004000e

(8004000e): "Operand" is unknown operand type.

An unknown operand type was specified.

### 13.2.54 8004000f

(8004000f): "Operand" is invalid format operand.

An invalid operand type was specified.



**13.2.55 80040010**

```
(80040010): "Operand" is invalid offset.
```

The register offset notation is not correct.

**13.2.56 80040011**

```
(80040011): "Operand" is invalid address register offset.
```

The register offset notation for the address register is not correct.

**13.2.57 80040012**

```
(80040012): "Operand" include unknown component.
```

An unknown component has been specified.

**13.2.58 80040015**

```
(80040015): break instruction is not between loop and endloop.
```

A break instruction was not placed between a loop and endloop instruction.

**13.2.59 80040016**

```
(80040016): loop instruction nest achieved limit.
```

A loop instruction was used beyond the nesting limit.

Up to 4 loop statements can be nested.

**13.2.60 80040017**

```
(80040017): Correspondent loop instruction is not found.
```

An endloop instruction is missing a corresponding loop instruction.

**13.2.61 8004001a**

```
(8004001a): if instruction nest achieved limit.
```

An ifb or ifc instruction was used beyond the nesting limit.

You can nest up to eight ifb and ifc instructions.

**13.2.62 8004001b**

```
(8004001b): Correspondent if instruction is not found.
```

An else instruction is missing a corresponding ifc or ifb instruction.

### 13.2.63 8004001d

(8004001d): loop instruction is not closed, but ret instruction is called.

A ret instruction cannot be placed between a loop and endloop instruction.

### 13.2.64 8004001f

(8004001f): if else instruction is not closed, but ret instruction is called.

A ret instruction cannot be placed between an ifc, ifb, or else instruction and an endif instruction.

### 13.2.65 80040021

(80040021): "Operand" is invalid format operand.

An invalid operand format was specified.

### 13.2.66 80040022

(80040022): "Operand" is invalid index.

An invalid index was specified for an operand register.

### 13.2.67 80040023

(80040023): "Operand" is invalid format operand.

The parentheses used to specify an offset for an operand register have not been closed.

### 13.2.68 80040024

(80040024): "Operand" is invalid offset.

An invalid offset has been specified for an operand register.

### 13.2.69 80040025

(80040025): "Operand" is invalid offset.

A register specified with an offset for an operand register is not allowed to use an index.

### 13.2.70 80040026

(80040026): "Operand" is invalid offset.

An invalid index was specified for an operand register.

### 13.2.71 80040027

(80040027): Correspondent if instruction is not found.

An `endif` instruction is missing a corresponding `ifc` or `ifb` instruction.

### 13.2.72 8004002a

(8004002a): Const register definition is duplicate.

Duplicate floating-point constant registers have been defined by `def` instructions.

### 13.2.73 8004002b

(8004002b): Bool register definition is duplicate.

Duplicate Boolean registers have been defined by `defb` instructions.

### 13.2.74 8004002c

(8004002c): Integer register definition is duplicate.

Duplicate integer registers have been defined by `defi` instructions.

### 13.2.75 80040031

(80040031): "Label name" is already used label name.

Duplicate label names have been used.

### 13.2.76 80040032

(80040032): Invalid label name is specified.

An invalid character has been used in a label name.

Use single-byte alphanumeric characters and underscores for label names.

### 13.2.77 80040035

(80040035): Error occurred while replacing macro instruction.

An error occurred while expanding a macro instruction.

Check whether you have already used a `def` instruction to define the floating-point constant registers that are automatically defined by `sincos` and other instructions.

### 13.2.78 80040039

```
(80040039): Cannot break from if statement.
```

You cannot place a `break` instruction within an `ifc` or `ifb` control block.

### 13.2.79 8004003a

```
(8004003a): ret instruction cannot be used just after endloop or endif.
```

A `ret` instruction cannot be called immediately after an `endloop` or `endif` instruction.

### 13.2.80 8004003b

```
(8004003b): At least 1 instruction need to be between if and else.
```

At least one assembly code instruction is required between an `ifb` or `ifc` instruction and an `else` instruction.

### 13.2.81 8004003c

```
(8004003c): At least 1 instruction need to be between else and endif.
```

At least one assembly code instruction is required between an `else` instruction and an `endif` instruction.

### 13.2.82 8004003d

```
(8004003d): def instruction cannot specify the register defined by pragma  
bind_symbol.
```

A register specified with `#pragma bind_symbol` cannot be specified with a `def` instruction.

### 13.2.83 8004003e

```
(8004003e): defb instruction cannot specify the register defined by pragma  
bind_symbol.
```

A register specified with `#pragma bind_symbol` cannot be specified with a `defb` instruction.

### 13.2.84 8004003f

```
(8004003f): defi instruction cannot specify the register defined by pragma  
bind_symbol.
```

A register specified with `#pragma bind_symbol` cannot be specified with a `defi` instruction.

**13.2.85 80040040**

(80040040): At least 1 instruction need to be between loop and endloop.

At least one assembly code instruction is required between a loop instruction and an endloop instruction.

**13.2.86 80040041**

(80040041): mova cannot be called continuously.

The mova instruction cannot be called consecutively.

**13.2.87 80050001**

(80050001): Cannot open output file.

Cannot open the output file.

Confirm whether there is another file with the same name and read-only or other attributes.

**13.2.88 80050003**

(80050003): The size of swizzle register is short.

The maximum number of swizzling and masking patterns has been exceeded.

**13.2.89 80050005**

(80050005): ret instruction cannot be found for Label "label name".

A label is missing a corresponding ret instruction.

A ret instruction is required for labels that are called as subroutines.

**13.2.90 80050007**

(80050007): The number of label is too big.

You cannot configure more than 65,535 labels.

**13.2.91 8005000a**

(8005000a): The exceptional jump is detected.

Exceptional jump control has occurred.

See section 12.5 Instructions That Cannot Be Called Consecutively.

**13.2.92 8005000b**

(8005000b): Cannot jump out from if statement and loop statement.

A jpb or jpc instruction cannot be used to jump from within an ifc or ifb instruction and an endif instruction, or from within a loop instruction and an endloop instruction, to outside a control block.

**13.2.93 80060004**

(80060004): "Operand" is invalid operand type.

An unsupported operand was specified for a shader instruction.

**13.2.94 80060005**

(80060005): Value cannot be specified for "operand".

You cannot specify a direct value for an operand.

**13.2.95 80060006**

(80060006): Index cannot be specified for "operand".

You cannot specify a register number for an operand.

**13.2.96 80060007**

(80060007): Component cannot be specified for "operand".

You cannot specify a component for the operand.

**13.2.97 80060009**

(80060009): '-' cannot be specified for "operand".

You cannot specify a minus sign ("-") with the operand.

**13.2.98 8006000b**

(8006000b): Offset index cannot be specified for "operand".

You cannot specify an index offset for the operand.

**13.2.99 8006000c**

(8006000c): Address register offset cannot be specified for "operand".

You cannot specify an index offset using an address register for the operand.

**13.2.100      8006000d**

(8006000d): Loop counter register offset cannot be specified for "operand".

You cannot specify an index offset using the loop-counter register for the operand.

**13.2.101      8006000e**

(8006000e): Loop counter register and address register cannot be specified together.

The loop-counter register and address register cannot be used at the same time.

**13.2.102      8006000f**

(8006000f): Index is not specified in "operand".

A register number has not been specified for the operand's register.

**13.2.103      80060010**

(80060010): Invalid index is specified in "operand".

The operand's register number exceeds the maximum number of registers.

**13.2.104      80060011**

(80060011): Invalid mask is specified for dest.

Masking has been incorrectly specified with the dest operand.

Specify masking in x, y, z, w order.

**13.2.105      80060012**

(80060012): Multiple constant registers cannot be specified at the same time.

You cannot specify more than one floating-point constant register as an operand at the same time.

**13.2.106      80060016**

(80060016): Src must have one of the following masks: .x|.y|.z|.w.

You must use one of the following swizzling specifications with the src operand: .x, .y, .z, or .w.

**13.2.107      80060017**

(80060017): Src0 and dest cannot be the same.

You cannot specify the same register for src0 and dest.

**13.2.108 80060018**

(80060018): Src0 cannot have any swizzle except the default swizzle (.xyzw)

You cannot use a swizzling specification other than .xyzw with src0.

**13.2.109 80060019**

(80060019): Dest must have one of the following masks: .x|.y|.z|.xy|.xz|.yz|.xyz.

You must specify one of the following masks with dest: .x, .y, .z, .xy, .xz, .yz, or .xyz.

**13.2.110 8006001b**

(8006001b): Dest must have "mask pattern" mask.

You must specify one of the masks shown by the mask pattern with dest.

**13.2.111 8006001f**

(8006001f): Dest and src0 cannot be the same.

You cannot specify the same register for dest and src0.

**13.2.112 80060020**

(80060020): Dest and src cannot be the same.

You cannot specify the same register for dest and src.

**13.2.113 80060021**

(80060021): Src0 must have one of the following masks: .x|.y|.z|.w.

You must use one of the following swizzling specifications with src0: .x, .y, .z, or .w.

**13.2.114 80060022**

(80060022): Src1 must have one of the following masks: .x|.y|.z|.w.

You must use one of the following swizzling specifications with src1: .x, .y, .z, or .w.

**13.2.115 80060023**

(80060023): Dest and src1 cannot be the same.

You cannot specify the same register for dest and src1.

**13.2.116 80060024**

(80060024): All operand must be the different register.



All operands must be different registers.

**13.2.117      80060025**

(80060025): Dest must have one of the following masks: `.x|.y|.xy`.

You must specify one of the following masks with dest: `.x`, `.y`, or `.xy`.

**13.2.118      80060026**

(80060026): Source modifier and swizzling cannot be specified for src1 and src2.

You cannot use swizzling or minus signs with src1 and src2.

**13.2.119      80060028**

(80060028): Dest and src1 cannot be the same.

You cannot specify the same register for dest and src1.

**13.2.120      80060029**

(80060029): Src1 cannot have any swizzle except the default swizzle (`.xyzw`).

You cannot use a swizzling specification other than `.xyzw` with src1.

**13.2.121      8006002a**

(8006002a): Dest must have one of the following masks: `.x|.y|.z|.xy|.xz|.yz|.xyz`.

You must specify one of the following masks with dest: `.x`, `.y`, `.z`, `.xy`, `.xz`, `.yz`, or `.xyz`.

**13.2.122      8006002c**

(8006002c): Source modifier and swizzling cannot be specified for src1.

You cannot use swizzling or minus signs with src1.

**13.2.123      8006002d**

(8006002d): Source modifier and swizzling cannot be specified for src2.

You cannot use swizzling or minus signs with src2.

**13.2.124      8006002e**

(8006002e): Invalid index is specified in "operand".

An invalid register number has been specified for the operand.

Specify the first register number for the operands of `m4x4` and other instructions. Confirm whether the register numbers used following macro expansion exceed the maximum number of registers.

**13.2.125      8006002f**

(8006002f): Constant register cannot be used for `src0`.

You cannot use a floating-point constant register for `src0`.

**13.2.126      80060030**

(80060030): Compare mode must be 0 or 1, 2, 3, 4, 5.

You must specify a value between 0 and 5 for the comparison mode with the `cmp` instruction.

**13.2.127      80060031**

(80060031): Status register bit must be 0 or 1.

You must specify a value of 0 or 1 for the status register.

**13.2.128      80060032**

(80060032): Condition mode is 0:OR 1:AND 2:OnlyStatus0 3:OnlyStatus1.

You must specify one of the following conditional modes: 0 (OR), 1 (AND), 2 (OnlyStatus0), or 3 (OnlyStatus1).

**13.2.129      80060033**

(80060033): Address register component must be x or y.

You must specify either the x or y component for the address register.

**13.2.130      80060036**

(80060036): `src0` and `src1` cannot be the same.

You cannot specify the same register for `src0` and `src1`.

**13.2.131      80060037**

(80060037): `src1` and `src2` cannot be the same.

You cannot specify the same register for `src1` and `src2`.

**13.2.132      80060038**

(80060038): `Dest` and `src2` cannot be the same.

You cannot specify the same register for `dest` and `src2`.

### 13.2.133      8006003b

(8006003b): Loop count must be in the range [0, 255].

You must use a value between 0 and 255 to set the loop count for the integer register defined by the `defi` instruction.

### 13.2.134      8006003c

(8006003c): Loop counter initial value must be in the range [0, 255].

You must use a value between 0 and 255 to set the initial value for the loop-counter register for the integer register defined by the `defi` instruction.

### 13.2.135      8006003d

(8006003d): Loop counter step must be in the range [-128, 127].

You must use a value between -128 and 127 to set the amount by which to increment the loop-counter register for the integer register defined by the `defi` instruction.

### 13.2.136      80060040

(80060040): Multiple input registers cannot be specified at the same time.

You cannot specify more than one input register at the same time for the `src` operand. You can specify the same register twice at the same time.

#### Code 13-1 Example for Error 80060040

```
add r0 , v0 , v0 // This does not cause an error
add r0 , v0, v1  // This causes an error
```

### 13.2.137      40070001

(40070001): Label "label name" is undefined.

A label could not be found.

(You can resolve this by linking another object that includes the label.)

### 13.2.138      40070003

(40070003): `mov` instruction both before `else` and after `endif` might cause the hardware to hang.

The hardware might hang when there is a `mov` instruction both right before an `else` instruction and right after an `endif` instruction. See section 12.10.1 Invalid Operation due to a `mov` Instruction.

**13.2.139      40070004**

(40070004): `mov` instruction at both first and last code in loop statement might cause the hardware to hang.

The hardware might hang when there is a `mov` instruction both right after a `loop` instruction and right before an `endloop` instruction. See section 12.10.1 Invalid Operation due to a `mov` Instruction.

**13.2.140      400a0001**

(400a0001): The series of 4 instructions from here might cause the hardware to hang if this instruction stalls for more than 3 clock cycles.

This series of four instructions meets the set of conditions that might cause the hardware to hang. The hardware might hang if the instruction on the indicated line stalls for at least three clock cycles. See section 12.10.2 Invalid Operation due to a Specific Order of Instructions.

**13.2.141      400a0002**

(400a0002): The series of 4 instructions from here might cause the hardware to hang if this instruction stalls for more than 1 clock cycle.

This series of four instructions meets the set of conditions that might cause the hardware to hang. The hardware might hang if the instruction on the indicated line stalls for at least two clock cycles. See section 12.10.2 Invalid Operation due to a Specific Order of Instructions.

**13.2.142      400a0003**

(400a0003): `mov` instruction just before `end` instruction might cause the hardware to hang.

The hardware might hang whenever a `mov` instruction comes just before the `end` instruction. See section 12.10.1 Invalid Operation due to a `mov` Instruction.

**13.2.143      400a0004**

(400a0004): `mov` instruction just before the last instruction writing to output register might cause the hardware to hang.

The hardware might hang whenever a `mov` instruction is followed by an instruction to write to an output register and then by an `end` instruction. See section 12.10.1 Invalid Operation due to a `mov` Instruction.

**13.2.144      400a0005**

(400a0005): `mov` instruction just before branch instruction might cause the hardware to hang if `mov` stalled due to a register dependency.

The hardware might hang whenever a branch instruction follows a `movb` instruction that depends on the register of a preceding instruction and therefore stalls. See section 12.10.1 Invalid Operation due to a `movb` Instruction.

## 13.3 Linker Error Messages

---

This chapter describes the errors and error codes output by the linker.

### 13.3.1 80080001

```
(80080001): Input file is not specified.
```

An input file has not been specified.

### 13.3.2 80080005

```
(80080005): "Argument" is not found.
```

The input file could not be found.

### 13.3.3 80080006

```
(80080006): Exceeded max number of long swizzle masks/patterns.
```

Exceeded the maximum number of swizzling patterns for the `mad` instruction.

### 13.3.4 80080007

```
(80080007): Exceeded max number of swizzle masks/patterns.
```

The total number of swizzling patterns has exceeded the limit.

### 13.3.5 8008000f

```
(8008000f): Label "label name" is duplicate.
```

Duplicate label names have been defined in a subroutine object.

### 13.3.6 80080012

```
(80080012): Cannot open output file.
```

Cannot generate the executable file.

Confirm whether another file with the same name and read-only or other attributes exists.

### 13.3.7 80080014

```
(80080014): "Input filename" is invalid file format.
```

The input file is not an object file.

### 13.3.8 80080015

```
(80080015): Some input files are the same name.
```

Input files with the same name have been specified.

### 13.3.9 8008001d

```
(8008001d): "Label name" is not subroutine.
```

A `ret` instruction has not been set for a label called as a subroutine from a `call` instruction.

### 13.3.10 8008000f

```
(8008001f): "Label name" cannot be found in input object files.
```

Could not find a label referenced from an input file.

### 13.3.11 80080020

```
(80080020): Vertex shader size is over the limit.
```

The maximum number of shader instructions has been exceeded.

You can link a shader with up to 512 instructions.

### 13.3.12 80080022

```
(80080022): "Register name" is duplicatedly defiend in "object name" and "object name".
```

A register has been defined with different values in multiple objects using `def`, `defi`, or `defb` instructions.

### 13.3.13 80080024

```
(80080024): "Register name" is duplicatedly defined in "object name" and "object name".
```

`#pragma output_map` definitions have mapped an output register to different output data attributes in multiple objects.

### 13.3.14 80080025

```
(80080025): symbol "symbol name" is duplicatedly defined in "object name" and "object name".
```

#pragma bind\_symbol definitions have bound a symbol name to different registers in multiple objects.

### 13.3.15 8008002a

(8008002a): symbol "symbol name" in "object name" and "symbol name" in "object name" are bound to the same register.

#pragma bind\_symbol definitions have bound symbols in two different objects to the same input register.

### 13.3.16 8008002b

(8008002b): "Label name" is duplicatedly defined in "subroutine object name"

A label name in a main object has also been defined in a subroutine object.

### 13.3.17 8008002c

(8008002c): "Output data attribute name" is duplicatedly defined in "object name" and "object name".

#pragma output\_map definitions have mapped an output data attribute to different output registers in multiple objects.

### 13.3.18 8008002d

(8008002d): Main routine cannot be found.

The input files do not have an object with the main and endmain labels.

### 13.3.19 8008002e

(8008002e): Cannot open map file.

Cannot generate the map file.

Confirm whether another file with the same name and read-only or other attributes exists.

### 13.3.20 8008002f

(8008002f): No input attribute is defined.

No input attributes have been defined.

### 13.3.21 80080030

(80080030): No output map is defined.

No output attributes have been defined.

### 13.3.22 80080031

```
(80080031): -debug and -nodebug cannot be specified together.
```

You cannot specify the `-debug` and `-nodebug` options together.

### 13.3.23 80080032

```
(80080032): def(bi) in ***.obj and bind_symbol in ***.obj specify the same register
**.
```

You cannot specify the same register with `bind_symbol` and `def` instructions.

### 13.3.24 80080033

```
(80080033): texture1 and texture2 need to be mapped to same register if 4 textures
are mapped.
```

`texture1` and `texture2` must be mapped to the same register when four textures have been set by `output_map` statements.

### 13.3.25 40090001

```
(40090001): end instruction is not found.
```

The end instruction is not found.

### 13.3.26 40090002

```
(40090002): end instruction is found in loop statement.
```

The end instruction was found in the `loop-endloop` statement.

### 13.3.27 40090003

```
(40090003): end instruction is found in only one of if and else statement.
```

The end instruction was found in only one of the `if-else` or `else-endif` statements.

### 13.3.28 40090004

```
(40090004): input register "Register Name" is not used.
```

There is a possibility that an input register defined by `#pragma bind_symbol` is not used.

### 13.3.29 40090005

```
(40090005): The access patterns of input registers are different between if and
else statement.
```



The input registers for the `if-else` statement and the `else-endif` statement are different.

### 13.3.30 40090006

```
(40090006): output register "Register Name" is not set.
```

The output register defined with `#pragma output_map` may not be written.

### 13.3.31 40090007

```
(40090007): output register is set in loop statement.
```

The output register is set in the `loop-endloop` instruction.

### 13.3.32 40090008

```
(40090008): The access patterns of output registers are different between if and else statement.
```

The output register in use differs between the `if-else` statement and the `else-endif` statement.

### 13.3.33 40090009

```
(40090009): output register "Register Name" is already set before.
```

The output register is being written to multiple times.

### 13.3.34 4009000a

```
(40090009a): Recursive call is found, and skipped.
```

A subroutine is being called recursively. The consistency checker feature will skip the call that is being called recursively.

### 13.3.35 4009000b

```
(40090009b): Cannot open file for performance report.
```

Cannot create a file for the output of the result of the performance checker feature.

### 13.3.36 4009000c

```
(4009000c): mova instruction both before and after returning from subroutine might cause the hardware to hang.
```

The hardware might hang for a `call`, `callb`, or `callc` code clause when there is a `mova` instruction both right before the final `ret` of the called subroutine and right after the subroutine returns. See section 12.10.1 Invalid Operation due to a `mova` Instruction.

## 14 File Format

This chapter describes the format of files generated by the assembler tools.

### 14.1 Intermediate Object Files

This section describes the format of intermediate object files generated by `ctr_VertexShaderAssembler32.exe`.

#### 14.1.1 Overview

Each file has the following structure.

**Figure 14-1 Intermediate Object File Structure**

File Header
Setup Information Block
Label Information Block
Program Code Information Block
Swizzle Data Information Block
Line Information Block
Relocation Information Block
Outmap Information Block
Bind Symbol Information Block
String Data Block

The following sections give details on each component.

#### 14.1.2 File Header

A fixed file header is placed at the beginning of each file. The header information is used to get the placement and number of data entries for each information block. The file header has the following structure.

**Code 14-1 File Header Structure**

```

typedef struct tagOBJFILEHEADER {
    char        signature[4];
    char        version[2];
    unsigned char  shaderType;
    unsigned char  mergeOutputMapsDebug;
    unsigned short inputMask;
    unsigned short outputMask;
    unsigned char  geometryDataMode;
    unsigned char  startIndex;
    unsigned char  subdivPatchSize;
    unsigned char  constVertexNumber;
    unsigned int   setupOffset;
    unsigned int   setupCount;
    unsigned int   labelOffset;
    unsigned int   labelCount;
    unsigned int   instOffset;
    unsigned int   instCount;
    unsigned int   swizzleOffset;
    unsigned int   swizzleCount;
    unsigned int   lineOffset;
    unsigned int   lineCount;
    unsigned int   relocOffset;
    unsigned int   relocCount;
    unsigned int   outmapOffset;
    unsigned int   outmapCount;
    unsigned int   bsymOffset;
    unsigned int   bsymCount;
    unsigned int   stringOffset;
    unsigned int   stringSize;
} OBJFILEHEADER

```

**Table 14-1 File Header Fields**

Name	Description
signature	Stores the string "DVOJ".
version	Includes the version of the assembler tool. The first byte is the major version and the second byte is the minor version.
shaderType	This is set to 0 for a vertex shader object and to 1 for a geometry shader object.
mergeOutputMapsDebug	Bit 0 is used by internal settings for the geometry shader. Bit 1 is set equal to 1 for debug builds and to 0 otherwise.

Name	Description
inputMask	The input registers information to use. A value of 1 is set for input registers defined by <code>#pragma bind_symbol</code> .
outputMask	The output registers information to use. A value of 1 is set for output registers defined by <code>#pragma output_map</code> .
geometryDataMode	Internal information for the geometry shader.
startIndex	Internal information for the geometry shader.
subdivPatchSize	Internal information for the geometry shader.
constVertexNumber	Internal information for the geometry shader.
setupOffset	The byte index within the file to the setup information block.
setupCount	The number of data entries for setup information.
labelOffset	The byte index within the file to the label information block.
labelCount	The number of data entries for label information.
instOffset	The byte index within the file to the program code information block.
instCount	The number of data entries for program code information.
swizzleOffset	The byte index within the file to the swizzle data information block.
swizzleCount	The number of data entries for swizzle data information.
lineOffset	The byte index within the file to the line information block.
lineCount	The number of data entries for line information.
relocOffset	The byte index within the file to the relocation information block.
relocCount	The number of data entries for relocation information.
outmapOffset	The byte index within the file to the Outmap information block.
outmapCount	The number of data entries for Outmap information.
bsymOffset	The byte index within the file to the Bind symbol information block.
bsymCount	The number of data entries for Bind symbol information.
stringOffset	The byte index within the file to the string data block.
stringSize	The number of bytes in the string data block.

### 14.1.3 Setup Information

There are `setupCount` entries in the setup information block given by `setupOffset` in the file header that store the setup information. Setup data is configured by the `def`, `defi` and `defb` instructions in shader assembly. Each setup information entry has the following structure.

#### Code 14-2 Setup Information Structure

```
typedef struct tagSETUPINFO{
    unsigned short  type;
    unsigned short  index;
    unsigned int    value[4];
} SETUPINFO
```

**Table 14-2 Setup Information Fields**

Name	Description
type	<ul style="list-style-type: none"> <li>0: Setup information for a Boolean register</li> <li>1: Setup information for an integer register</li> <li>2: Setup information for a floating-point constant register</li> </ul>
index	Register index
value	For a Boolean register, this has a value of 1 when <code>value[0]</code> is true and 0 when <code>value[0]</code> is false. For an integer register, this stores the three values defined by a <code>defi</code> instruction in bits [7:0], [15:8], and [23:16] of <code>value[0]</code> , respectively. For a floating-point constant register, this converts the four values defined by a <code>def</code> instruction into 24-bit floating-point numbers and then stores them in <code>value[0]</code> through <code>value[3]</code> .

### 14.1.4 Label Information

There are `labelCount` entries in the label information block given by `labelOffset` in the file header where label information is stored. Label information is set in shader assembly. Each label information entry has the following structure.

#### Code 14-3 Label Information Structure

```
typedef struct tagLABELINFO{
    unsigned int  index;
    unsigned int  address;
    unsigned int  length;
    unsigned int  stringIndex;
} LABELINFO
```

**Table 14-3 Label Information Fields**

Name	Description
index	The label information index. These are numbered starting at 0x00010000 in the order that they were defined in shader assembly.
address	The shader program address set by the label.
length	The distance from the address set by the label to the <code>ret</code> instruction. This is the subroutine length.
stringIndex	The byte index in the string data block that stores the label name.

### 14.1.5 Program Code Information

There are `instCount` entries in the program code information block given by `instOffset` in the file header that store the program code information. 32 bits of program code information correspond to a single instruction in shader assembly.

**Note:** This does not include definition instructions. Macro instructions and control flow instructions may not always have a one-to-one correspondence with program code information.

### 14.1.6 Swizzle Data Information

There are `swizzleCount` entries in the swizzle data information block given by `swizzleOffset` in the file header that store the swizzle data information. Each swizzle data information entry has the following structure.

**Code 14-4 Swizzle Data Information Structure**

```
typedef struct tagSWIZZLEINFO{
    unsigned int    value;
    unsigned short  usedInfo;
    unsigned short  reserve;
} SWIZZLEINFO
```

**Table 14-4 Swizzle Data Information Fields**

Name	Description
value	The swizzle data itself.
usedInfo	Internal information used when linking.
reserve	A reserved region.

### 14.1.7 Line Information

There are `lineCount` entries in the line information block indicated by `lineOffset` in the file header that store the line information. Each line information entry has a one-to-one correspondence with a program code information entry: it stores the filename and line count of the shader object for the program code information entry with the same index. Each line information entry has the following structure.

#### Code 14-5 Line Information Structure

```
typedef struct tagLINEINFO{
    unsigned int  stringIndex;
    unsigned int  lineNo;
} LINEINFO
```

**Table 14-5 Line Information Fields**

Name	Description
<code>stringIndex</code>	The index to the region storing the filename of the shader assembly for the corresponding program code. This is the index (in bytes) within the string data block.
<code>lineNo</code>	The number of lines of shader assembly for the corresponding program code.

### 14.1.8 Relocation Information

There are `relocCount` entries in the relocation information block indicated by `relocOffset` in the file header that store the relocation information. The relocation information is accessed at link time. Each relocation entry has the following structure.

#### Code 14-6 Relocation Information Structure

```
typedef struct tagRELOCATIONINFO{
    unsigned int  address;
    unsigned short type;
    unsigned short reserve;
    unsigned int  stringIndex;
} RELOCATIONINFO
```

**Table 14-6 Relocation Information Fields**

Name	Description
<code>address</code>	The program address associated with the relocation entry.
<code>type</code>	<ul style="list-style-type: none"> <li>0: Address relocation</li> <li>1: Unresolved subroutine relocation</li> <li>4: Swizzle index relocation</li> </ul>

reserve	A reserved region.
stringIndex	The byte offset within the string data block to the location storing the label name for the unresolved subroutine.

### 14.1.9 Outmap Information

There are `outmapCount` entries in the Outmap information block indicated by `outmapOffset` in the file header that stores the Outmap information. Outmap information is defined by `#pragma output_map` in shader assembly. Each Outmap information entry has the following structure.

#### Code 14-7 Outmap Information Structure

```
typedef struct tagOUTMAPINFO{
    unsigned short  type;
    unsigned short  index;
    unsigned short  mask;
    unsigned short  reserve;
} OUTMAPINFO
```

Table 14-7 Outmap Information Fields

Name	Description
type	This stores the attribute type. <ul style="list-style-type: none"> <li>• 0: position</li> <li>• 1: quaternion</li> <li>• 2: color</li> <li>• 3: texcoord0</li> <li>• 4: texcoord0w</li> <li>• 5: texcoord1</li> <li>• 6: texcoord2</li> <li>• 8: view</li> <li>• 9: generic</li> </ul>
index	The output register index.
mask	The specified components. These are x, y, z, and w in order from the least-significant bit.
reserve	A reserved region.

### 14.1.10 Bind Symbol Information

There are `bsymCount` entries in the Bind symbol information block indicated by `bsymOffset` in the file header that store the Bind symbol information. Bind symbol information is defined by



`#pragma bind_symbol` in shader assembly. Each `Bind symbol` information entry has the following structure.

#### Code 14-8 Bind Symbol Information Structure

```
typedef struct tagBINDSYMBOLINFO{
    unsigned int    stringIndex;
    unsigned short  startIndex;
    unsigned short  endIndex;
} BINDSYMBOLINFO
```

**Table 14-8 Bind Symbol Information Fields**

Name	Description
stringIndex	The byte index within the string data block that stores symbol names.
startIndex	The starting register index. <ul style="list-style-type: none"><li>• 0–15: Input registers 0–15</li><li>• 16–111: Floating-point constant registers 0–95</li><li>• 112–115: Integer registers 0–3</li><li>• 120–135: Boolean registers 0–15</li></ul>
endIndex	The ending register index. This has the same values as <code>startIndex</code> .

### 14.1.11 String Data

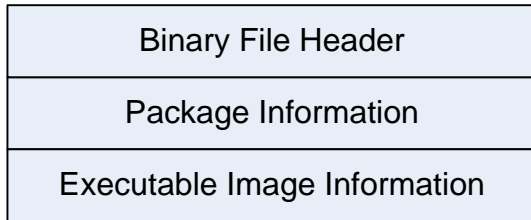
There are `stringSize` bytes of string data placed in the string data block indicated by `stringOffset` in the file header. These include label names, symbol names, filenames, and so on. Each string is delimited by the null character (`'\0'`).

## 14.2 Executable Binary Files

This section describes the format of executable binary files generated by `ctr_VertexShaderLinker32.exe`.

### 14.2.1 Overview

Each file has the following structure.

**Figure 14-2 Executable Binary File Structure**

The following sections give details on each component.

## 14.2.2 Binary File Header

A variable-length binary file header is placed at the start of the file. The binary file header has the following structure.

**Code 14-9 Binary File Header Structure**

```
typedef struct tagBINFILEHEADER{
    char        signature[4];
    unsigned int exeCount;
    unsigned int exeOffsetTop;
} BINFILEHEADER
```

**Table 14-9 Binary File Header Fields**

Name	Description
signature	Stores the string "DVLB".
exeCount	The number of information entries for executable images.
exeOffsetTop	Stores the byte offset within the file to the first information entry for an executable image. If <code>exeCount</code> is 2 or greater, the binary file has multiple executable image information entries. The byte offsets to the executable image information are stored immediately after <code>exeOffsetTop</code> using four bytes per entry.

## 14.2.3 Package Information

Package information is placed immediately after the binary file header. Package information has the following structure.

**Figure 14-3 Package Information Structure**

Package Information Header
Program Code Information Block
Swizzle Data Information Block
Line Information Block
String Data Block

The following sections describe each component.

#### 14.2.3.1 Package Information Header

A fixed header is placed at the start of the package information. The placement and number of data entries for each information block is obtained from the header information. The package information header has the following structure.

**Code 14-10 Package Information Header Structure**

```
typedef struct tagPKGHEADER{
    char        signature[4];
    char        version[2];
    unsigned short reserved0;
    unsigned int instOffset;
    unsigned int instCount;
    unsigned int swizzleOffset;
    unsigned int swizzleCount;
    unsigned int lineOffset;
    unsigned int lineCount;
    unsigned int stringOffset;
    unsigned int stringSize;
} PKGHEADER
```

**Table 14-10 Package Information Header Fields**

Name	Description
signature	Stores the string "DVLP".
version	Includes the version of the linker tool. The first byte is the major version and the second byte is the minor version.
instOffset	The byte index within the package information to the program code information block.
instCount	The number of data entries for program code information.

Name	Description
swizzleOffset	The byte index within the package information to the swizzle data information block.
swizzleCount	The number of swizzle data information entries.
lineOffset	The byte index within the package information to the line information block.
lineCount	The number of data entries for line information.
stringOffset	The byte index within the package information to the string data block.
stringSize	The number of bytes in the string data block.

#### 14.2.3.2 Package Information Blocks

Package information and intermediate object file information use the same structure for program code, swizzle data, and line information blocks. For details on each of these, see sections 14.1.5 Program Code Information, 14.1.6 Swizzle Data Information, and 14.1.7 Line Information, respectively. Each package information block that references string data uses an index within the string data block in the package information.

#### 14.2.4 Executable Image Information

Executable image information holds information that is set for each linked object file. Executable image information is generated for each linked main object. Executable image files are placed at the byte offset (within the file) stored in `exeOffsetTop` in the binary file header. When there are multiple executable image information entries, byte offsets to them are stored immediately after `exeOffsetTop` using four bytes apiece. Each information entry for an executable image has the following structure.

**Figure 14-4 Executable Image Information Structure**

Executable Image Information Block
Setup Information Block
Label Information Block
Outmap Information Block
Bind Symbol Information Block
String Data Block

The following sections describe each component.

#### 14.2.4.1 Executable Image Information Header

Each information entry for an executable image has a header. The executable image's information header has the following structure.

**Code 14-11 Executable Image Information Header Structure**

```
typedef struct tagEXEIMAGEHEADER{
    char        signature[4];
    char        version[2];
    unsigned char  shaderType;
    unsigned char  mergeOutputMapsDebug;
    unsigned int  mainAddr;
    unsigned int  endAddr;
    unsigned short inputMask;
    unsigned short outputMask;
    unsigned char  geometryDataMode;
    unsigned char  startIndex;
    unsigned char  subdPatchSize;
    unsigned char  constVertexNumber;
    unsigned int  setupOffset;
    unsigned int  setupCount;
    unsigned int  labelOffset;
    unsigned int  labelCount;
    unsigned int  outmapOffset;
    unsigned int  outmapCount;
    unsigned int  bsymOffset;
    unsigned int  bsymCount;
    unsigned int  stringOffset;
    unsigned int  stringSize;
} EXEIMAGEHEADER
```

**Table 14-11 Executable Image Information Header Fields**

Name	Description
signature	Stores the string "DVLE".
version	Includes the version of the assembler tool. The first byte is the major version and the second byte is the minor version.
shaderType	This is set to 0 for a vertex shader object and to 1 for a geometry shader object.
mergeOutputMapsDebug	Bit 0 is used by internal settings for the geometry shader. Bit 1 is set equal to 1 for debug builds and to 0 otherwise.
mainAddr	The program address set by the main label.

Name	Description
endAddr	The program address set by the <code>endmain</code> label.
inputMask	The input registers information to use. A value of 1 is set for input registers defined by <code>#pragma bind_symbol</code> .
outputMask	The output registers information to use. A value of 1 is set for output registers defined by <code>#pragma output_map</code> .
geometryDataMode	Internal information for the geometry shader.
startIndex	Internal information for the geometry shader.
subdivPatchSize	Internal information for the geometry shader.
constVertexNumber	Internal information for the geometry shader.
setupOffset	The byte index within the executable image information to the setup information block.
setupCount	The number of data entries for setup information.
labelOffset	The byte index within the executable image information to the label information block.
labelCount	The number of data entries for label information.
outmapOffset	The byte index within the executable image information to the Outmap information block.
outmapCount	The number of data entries for Outmap information.
bsymOffset	The byte index within the executable image information to the Bind symbol information block.
bsymCount	The number of data entries for Bind symbol information.
stringOffset	The byte index within the executable image information to the string data block.
stringSize	The number of bytes in the string data block.

#### 14.2.4.2 Executable Image Information Blocks

Executable image information and intermediate object file information use the same information data structure for setup, label, Outmap, and Bind symbol information blocks. For details on each of these, see sections 14.1.3 Setup Information, 14.1.4 Label Information, 14.1.9 Outmap Information, and 14.1.10 Bind Symbol Information, respectively. Each executable image information block that references string data uses an index within the string data block in the executable image information.

## 15 Shader Checking Feature

This section describes the shader checking feature performed by the assembler tool.

### 15.1 Consistency Checker Feature

---

The consistency checker feature confirms that the assembly code is properly implemented. This feature is enabled by specifying the `-check_consistency` option in `ctr_VertexShaderLinker32.exe` and is implemented in the linker. For each main object in the link target, it checks several items. It follows the instructions from the `main` label of the main object and checks up until the `endmain` label. When it finds something it checks for has been implemented, it outputs a warning.

The consistency checker feature checks each instruction according to the following conditions.

- For `call` instructions, it also checks the call destination.
- It assumes that conditional jump and conditional call instructions do not branch. (The branch destinations for `jpb`, `jpc`, `callb`, and `calld` are not considered.)
- For `ifb` and `ifc` instructions, it checks both the `if` items and `else` items.
- Because an infinite loop occurs when a call instruction recursively calls a subroutine, it skips executing the call instructions and goes on to the next instruction when the same subroutine is called more than once by nested `call` instructions. (A warning is output in this case.)

It checks the following items.

- The execution of `end` instructions
- The reading of input registers
- The writing of output registers

The following gives details for each item checked. In this section, the `if` instruction indicates both the `ifb` and `ifc` instructions.

#### 15.1.1 end Instruction Execution Check

---

Checks whether the `end` instruction was properly called. A warning is output if the following check items are found.

- No `end` instruction is found.
- No `end` instruction is found in the `loop-endloop` instruction.
- Only the `if-else` statement or `else-endif` statement has an `end` instruction. If the `if` instruction has no corresponding `else` instruction, and the `end` instruction is found in the `if-endif` statement, a warning is output.

### 15.1.2 Input Register Read Check

---

Checks whether the input registers are being used properly. This check targets the input registers and components specified with `#pragma bind_symbol`. A warning is output if the following check items are found.

- When not all input registers are being used, the check reports the names of registers and components that might not be in use.
- When the way in which the input registers are used in the `if-else` and `else-endif` statements differ, this is reported because these differences might be caused by how the `if` instruction branches. If there is no `else` instruction corresponding to the `if` instruction, and an input register is used in the `if-endif` statement, a warning is output.

### 15.1.3 Output Register Write Check

---

Checks whether the output registers are being used properly. It targets output registers specified with `#pragma output_map`. In addition, all `xyzw` components are targeted. A warning is output if the following check items are found.

- When not all output registers are written, the check reports the names of registers and components that might not have been written.
- When an output register is set in the `loop-endloop` instruction, the check reports this because, depending on how many times the loop statement is repeated, the output register may be written to several times.
- When the way in which the output registers are used in the `if-else` and `else-endif` statements differ, this is reported because these differences might be caused by how the `if` instruction branches. If there is no `else` instruction corresponding to the `if` instruction, and an output register is used in the `if-endif` statement, a warning is output.
- When an output register is being written to multiple times, this check reports the names of the registers or components that might be written multiple times due to writing the instructions in question.

## 15.2 Performance Checker Feature

---

The performance checker has two features: It estimates the number of clock cycles per vertex it will take during execution based on the implementation of the shader assembly code, and it detects instructions that will generate stalls. The performance checker feature is implemented in the linker, and it is enabled by specifying the `-check_performance` option in `ctr_VertexShaderLinker32.exe`. It checks each main object in the link target. Instructions are tracked from the main label of the main object, and the check is carried out until the `end` instruction or the `endmain` label. The result of the check is output to the command prompt where the linker was executed and at the same time to a file. This output file is created with the same name as the binary file created by the linker, but with the extension changed to `"perf.txt"`. (It is also created in the same location as the binary file.)



The performance checker feature checks each instruction according to the following conditions.

- For `call` instructions, it also checks the call destination.
- It assumes that conditional jump and conditional call instructions do not branch. (The branch destinations for `jpb`, `jpc`, `callb`, and `callc` are not considered. The condition is always determined to be `FALSE`.)
- For `ifb` and `ifc` instructions, it only checks the `else` item. (The condition is always determined to be `FALSE`.)
- Because an infinite loop occurs when a `call` instruction recursively calls a subroutine, it skips executing the `call` instructions and goes on to the next instruction when the same subroutine is called more than once by nested `call` instructions.

The following items are output as the result of the check:

- The total number of clock cycles for the executed instructions
- The number of executed instructions
- Instructions that stall, the number of clock cycles, and the cause(s) of the stall.

In calculating the total number of clock cycles, one tick is counted for each instruction, and if there is a stall, the number of ticks of the stall is added to the total. Note that this process does not account for all of the causes of stalls that can take place on the actual hardware. The calculated total number of clock cycles is simply a guideline and will not completely match the total on the actual hardware.

Details of the performance checker feature are presented below.

## 15.2.1 Detectable Causes of Stalls

---

This feature can detect a number of the factors that can cause stalls. The following sections provide details about these various factors.

### 15.2.1.1 Stalls Due to Instruction Dependencies

The performance checker detects stalls caused by dependent relationships between registers used by instructions. Processing will stall for two given instructions called in sequence when the instruction called later references the computation result of the preceding instruction before that computation has been completed, as the later instruction must wait for the computation to complete. The number of clock cycles for this stall depends on the latency of the instruction that was called first and on the number of other instructions that execute between these two instructions. To learn about the latency of various instructions, see section 12.7 Instruction Latency. The registers that are subject to this are the temporary registers and the status registers. For status registers, the `cmp` instructions are sometimes dependent on `ifc`, `callc` and `breakc` instructions.

#### Code 15-1 Dependency Stall Example

```
dp4 r0 , r1 , r2
add r3 , r4 , r0
```

In this case above, the `dp4` calculation result written to `r0` is used as a source operand in the very next `add` instruction. The `dp4` instruction has a latency of five clock cycles, so the performance checker detects that execution of the `add` instruction will stall for four clock cycles.

If a given instruction is dependent on a number of other instructions, the performance checker uses the instruction that stalls for the largest number of clock cycles when calculating the stall duration of that given instruction.

#### Code 15-2 Multiple Dependency Stall Example 1

```
dp4 r0.x , r1 , r2
mov r0.y, c0
add r3 , r4 , r0
```

The `add` instruction in the example above depends on the `r0.x` destination operand of the `dp4` instruction, and on the `r0.y` destination operand of the `mov` instruction. This means that the `add` instruction here will stall for three clock cycles due to `dp4` and stall for one clock cycle due to `mov`. Three clock cycles is the largest value, so the performance checker result is three clock cycles.

##### 15.2.1.2 Stalls Due to `mov` Instruction Calls

Calls to the `mov` instruction always stall for three clock cycles.

##### 15.2.1.3 Stalls Due to Branching

Processing stalls for two clock cycles when the program counter is changed by +1 or more by a `call` or `if` instruction. When processing returns from the subroutine called by the `call` instruction, there is a stall of two clock cycles for the instruction just before the `ret` instruction.

### 15.2.2 When There Are Multiple Stall Causes

If there are multiple stall causes for a given instruction, then the stall with the largest number of clock cycles is used when calculating the stall for the given instruction.

#### Code 15-3 Multiple Dependency Stall Example 2

```
main:
call label0 // The call to label0
...
end

label0:
dp4 r0.x , r1, r2
mov a0.x, r0.x
ret
```

In the example above, the final `mov` instruction in the `label0` subroutine stalls for four clock cycles due to the dependency on the prior `dp4` instruction, then the `mov` call itself stalls for three clock

cycles, and finally the `call` branch instruction causes a stall of two clock cycles. The largest stall value of four clock cycles is used as the calculated result.

### 15.2.3 Outputting the Results

The performance checker outputs its results to the command prompt where the linker was executed and also to a file. This output file is created with the same name as the binary file created by the linker, but with the extension changed to `perf.txt`.

The content of this output file looks something like the example below.

#### Code 15-4 Performance Checker Output

```
Main object: obj\VShader.o

Total executed clock count          14 clock
Total executed instruction count     7 instructions

Detail of stall
=====

VShader.vsh(26): 2 clock stall is caused by branch.

VShader.vsh(36): 4 clock stall is caused.
|
+--- 3 clock stall is by mova instruction.
|
+--- 2 clock stall is by branch.
|
+--- VShader.vsh(35): 4 clock is to wait for this instruction to finish writing r0.x.

VShader.vsh(30): 1 clock stall is caused by reading temporary register.
|
+--- VShader.vsh(29): To wait for this instruction to finish writing r1.y.
...

```

Main object is the name of the main object that is the target of the performance check. A check result is output for each main object in the link target. Total executed clock count is the total number of clock cycles per vertex required for execution. The information below Detail of stall shows the location and cause of the stall and the number of clock cycles of the stall. For a stall due to dependencies, the locations of instructions with dependency relationships and the registers that are the causes are shown. If there are multiple causes of stalls, then the number of clock cycles for each

cause is shown, together with the number of clock cycles that is the result of the combination of the causes.

The check results are output in the order of the execution of the instructions, starting from the `main` label. If an instruction is called multiple times in the same subroutine, then the performance check is conducted multiple times on the instruction and the same stall information is output multiple times.

Basically, almost the same information that is output to the file is also output to the command prompt where the linker was executed. (The information regarding dependencies is shown indented.) If the linker is executed from an environment like Microsoft Visual Studio, then you can jump to the source file of the shader assembly code by clicking the output result for causes of stalls in the Output window. This is useful for checking the places where problems arise.

DMP and PICA are registered trademarks of Digital Media Professionals Inc.

All other company and product names in this document are the trademarks or registered trademarks of their respective companies.

Copyright © 2009-2010 Digital Media Professionals Inc. All rights reserved.

This documentation is the confidential and proprietary property of Digital Media Professionals Inc. The possession or use of this documentation and its content requires a written license from Digital Media Professionals Inc.

© 2009-2011 Nintendo

The contents of this document cannot be duplicated, copied, reprinted, transferred, distributed, or loaned in whole or in part without the prior approval of Nintendo.