TG Shader Infrastructure

Abstract

TBD

1 General notes on token format

A token is a 32-bit machine word that is split into bitfields. A bitfield that has a name and a range of bits it occupies in the token specified is called a token field. A set of token fields that compromise a token is called a token layout. In this document, token layouts are defined in tables similar to Table 1.

Bits	Name	Description
0:7	Field1	8-bit field occupying 8 least significant bits of the token
8:15	Field2	Another 8-bit field
16	A	1-bit field
17:31	Field3	Another field consisting of 15 bits

Table 1. Example token layout

Little-endian byte ordering is assumed. Thus the token layout presented in Table 1 has the bit layout as in Figure 1.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Field3															A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field2								Field1							

Figure 1. Example bit layout

To aid in future expansion of the language, there is a desire to have a general rule that applies to the bitfield layout in a token.

If a given token layout includes the Type field, it means that the 4 least significant bits designate this token by a unique numeric value. This value need not necessarily be unique globally.

If a token layout includes the Extended field, it means that the most significant bit indicates that another token is following this token. It is a general rule that the extended tokens immediately follow a given token and they take precedence over other possible extra tokens.

A token that has at least Type and Extended fields defined in its token layout is called a simple token.

Bits	Name	Description
0:3	Type	Simple token type
4:30	Data	Arbitrary data
31	Extended	If TRUE, another simple token follows

Table 2. Simple token layout

If a token layout includes the Size field, it tells how many tokens (including this one) should be skipped to get to the next token with a layout that also includes the Size field. The presence of the Size field requires the Type field to be also included in the token layout, as it occupies 8 least significant bits after the Type field. For the sake of completness, the Extended field must be present in the token layout.

A token that has at least Type, Size and Extended fields defined in its token layout is called a sized token.

Bits	Name	Description
0:3	Type	Sized token type
4:11	Size	Distance to the next sized token
12:30	Data	Arbitrary data
31	Extended	If TRUE, another simple token follows

Table 3. Sized token layout

1.1 Traversing the token stream

The token stream is a structured collection of tokens.

It is possible to traverse the whole token stream without the need to understand all the token types. If the Type field cannot be recognized, the token can be safely skipped.

If the Size field is present, the parser should skip next Size − 1 tokens to advance to the next token. If not, the Extended bit should be used to decide whether there is another token to advance to.

2 Token stream format

The first token in the stream is the VERSION token, described below.

Bits	Name	Description
0:7	MajorVersion	Major version of the token stream that is bumped every time the structure of the stream changes. If the number is higher than the maximum supported, the parser should refuse to load the token stream
8:15	MinorVersion	Minor version of the token stream, called sometimes the revision number. If the number is higher than the maximum supported for a given major version, the parser still should be able to load the token stream (possibly skipping some unrecognized tokens) but should refuse to execute it
16:31	Padding	Must be 0

Table 4. VERSION token layout

Immediately after the VERSION token the HEADER token follows, described below.

Bits	Name	Description
0:7	HeaderSize	Distance to the first token in the token stream body
8:31	BodySize	Number of tokens in the token stream body

Table 5. HEADER token layout

The token stream body is a sequence of sized tokens that contain instructions, declarations and the immediate buffer data. Those do not necessarily have to immediately follow the HEADER token. There is exactly HeaderSize − 1 tokens between the end of the header and the beginning of the body.

If HeaderSize is greater than 1, after the HEADER token the PROCESSOR token follows.

Bits	Name	Description
0:3	Processor	Target processor (see Table 7)
4:31	Padding	Must be 0

Table 6. PROCESSOR token layout

Ordinal	Name	Description
0	PROCESSOR_FRAGMENT	Fragment shader
1	PROCESSOR_VERTEX	Vertex shader
2	PROCESSOR_GEOMETRY	Geometry shader

Table 7. PROCESSOR enums

2.1 Current revision

VERSION

MajorVersion	=	1
MinorVersion	=	1
Padding	=	0

HEADER

HeaderSize	=	2
BodySize	$\geqslant$	0

PROCESSOR

Processor	see Table 7	
Padding	=	0

3 Declarations

The DECLARATION token declares program variables such as inputs, outputs or temporaries that are going to be utilized by the program. Variables are referenced by specyfing a particular file (see Table 9) and an index into this file. A single DECLARATION token can declare multiple variables belonging to the same file.

For input variable declaration targetting fragment processors, an optional token can follow that specifies the interpolation method used to interpolate given inputs.

Bits	Name	Description
0:3	Type	
4:11	Size	
12:15	File	Register file (see Table 9)
16:19	Declare	Type of declaration (see Table 10)
20	Interpolate	If TRUE, the DECLARATION_INTERPOLATE token follows
21:30	Padding	Must be 0.
31	Extended	

Table 8. DECLARATION token layout

Ordinal	Name	Description
0	FILE_NULL	Empty, write-only
1	FILE_CONSTANT	
2	FILE_INPUT	
3	FILE_OUTPUT	
4	FILE_TEMPORARY	
5	FILE_SAMPLER	
6	FILE_ADDRESS	
7	FILE_IMMEDIATE	Embedded constants, read-only (see Section 4)

Table 9. FILE enums

Ordinal	Name	Description
0	DECLARE_RANGE	See Section 3.1
1	DECLARE_MASK	See Section 3.2

Table 10. DECLARE enums

If the Declare field is DECLARE_RANGE, the DECLARATION_RANGE token follows (see Section 3.1).

If the Declare field is DECLARE_MASK, the DECLARATION_MASK token follows (see Section 3.2).

If the Interpolate field is TRUE, the DECLARATION_INTERPOLATION token follows (see Section 3.3).

3.1 Range declaration

DECLARATION

Type	= TOKEN_TYPE_DECLARATION
Size	= 2
File	see Table 9
Declare	= DECLARE_RANGE
Interpolate	= FALSE
Padding	= 0
Extended	= FALSE

Bits	Name	Description
0:15	First	Index of the first register in the file that is accessed
16:31	Last	Index of the last register in the file that is accessed

Table 11. DECLARATION_RANGE token layout

There can be multiple range declaration tokens. If so, they should be merged with the previous range by creating an union of the ranges. The default range is as if the following token was parsed.

DECLARATION_RANGE

First	= 0
Last	= 0

3.2 Mask declaration

DECLARATION

Type	= TOKEN_TYPE_DECLARATION
Size	= 2
File	see Table 9
Declare	= DECLARE_MASK
Interpolate	= FALSE
Padding	= 0
Extended	= FALSE

Bits	Name	Description
0:31	Mask	Marks individual registers in the file that are accessed by setting bit N to 1 for register N

Table 12. DECLARATION_MASK token layout

Mask declaration is a convenient way to declare register ranges for the first 32 registers.

3.3 Interpolation method declaration

DECLARATION

Type	= TOKEN_TYPE_DECLARATION
Size	= 3
File	see Table 9
Declare	= DECLARE_RANGE
Interpolate	= TRUE
Padding	= 0
Extended	= FALSE

Bits	Name	Description
0:3	Interpolate	Interpolation method (see Table 14)
4:31	Padding	Must be 0

Table 13. DECLARATION_INTERPOLATION token layout

Ordinal	Name	Description
0	INTERPOLATE_CONSTANT	Constant
1	INTERPOLATE_LINEAR	Linear
2	INTERPOLATE_PERSPECTIVE	Perspective-correct

Table 14. INTERPOLATE enums

4 Immediates

Bits	Name	Description
0:3	Type	
4:11	Size	
12:15	DataType	Type of data contained in the immediate buffer (see Table 16)
16:30	Padding	Must be 0
31	Extended	

Table 15. IMMEDIATE token layout

Ordinal	Name	Description
0	IMM_FLOAT32	32- bit IEEE floating point (see Section 4.1)

Table 16. IMM enums

4.1 Float data type

IMMEDIATE

Type	=	TOKEN_TYPE_IMMEDIATE
Size	=	$1+N$
DataType	=	IMM_FLOAT32
Padding	=	0
Extended	=	FALSE

The N specifies the number of IMMEDIATE_FLOAT32 tokens that follow.

Bits	Name	Description
0:31	Float	32-bit IEEE floating point value

Table 17. IMMEDIATE_FLOAT32 token layout

5 Instructions

Bits	Name	Description
0:3	Type	
4:11	Size	
12:19	Opcode	Operation code (see Table ?)
20:21	Saturate	Saturation mode (see Table 19)
22:23	NumDstRegs	Number of destination registers
24:27	NumSrcRegs	Number of source registers
28:30	Padding	Must be 0
31	Extended	

Table 18. INSTRUCTION token type

Ordinal	Name	Description
0	SAT_NONE	Do not saturate
1	SAT_ZERO_ONE	Clamp to $\langle 0,1\rangle$
2	SAT_MINUS_PLUS_ONE	Clamp to $\langle -1,1\rangle$

Table 19. SAT enums

5.1 NV instruction extension

Bits	Name	Description
0:3	Type	
4:7	Precision	Precision mode (see Table ?)
8:11	CondDstIndex	Conditional destination register index
12:15	CondFlowIndex	Conditional execution register index
16:19	CondMask	Condition code mask (see Table ?)
20:21	CondSwizzleX	Conditional execution register swizzle X (see Table 21)
22:23	CondSwizzleY	Conditional execution register swizzle Y (see Table 21)
24:25	CondSwizzleZ	Conditional execution register swizzle Z (see Table 21)
26:27	CondSwizzleW	Conditional execution register swizzle W (see Table 21)
28	CondDstUpdate	If TRUE, write the conditional destination register
29	CondFlowEnable	If TRUE, read the conditional execution register
30	Padding	Must be 0
31	Extended	

Table 20. INSTRUCTION_EXT_NV token layout

Ordinal	Name	Description
0	SWIZZLE_X	Component X
1	SWIZZLE_Y	Component Y
2	SWIZZLE_Z	Component Z
3	SWIZZLE_W	Component W

Table 21. SWIZZLE enums

5.1.1 Default values

Setting the token fields to the values given below effectively disables it.

INSTRUCTION_EXT_NV

Type	=	INSTRUCTION_EXT_TYPE_NV
Precision	=	PRECISION_DEFAULT
CondDstIndex	=	0
CondFlowIndex	=	0
CondMask	=	CC_TR
CondSwizzleX	=	SWIZZLE_X
CondSwizzleY	=	SWIZZLE_Y
CondSwizzleZ	=	SWIZZLE_Z
CondSwizzleW	=	SWIZZLE_W
CondDstUpdate	=	FALSE
CondFlowEnable	=	FALSE
Padding	=	0
Extended	=	FALSE

5.1.2 Enabling conditional register write

INSTRUCTION_EXT_NV

CondDstIndex	=	Register index
CondDstUpdate	=	TRUE

5.1.3 Controling precision

INSTRUCTION_EXT_NV		
Precision	=	Instruction precision

5.1.4 Enabling conditional execution of instruction

INSTRUCTION_EXT_NV		
CondFlowIndex	=	Register index
CondMask	=	Condition code mask
CondSwizzleX	=	Swizzle X
CondSwizzleY	=	Swizzle Y
CondSwizzleZ	=	Swizzle Z
CondSwizzleW	=	Swizzle W
CondFlowEnable	=	TRUE

5.2 LABEL instruction extension

Bits	Name	Description
0:3	Type	
4:27	Label	Symbolic name
28	Target	Specifies whether this is a target label (label declaration) or the one we should jump to
29:30	Padding	Must be 0
31	Extended	

Table 22. INSTRUCTION_EXT_LABEL token layout

The label is just a symbolic name. When declaring a label (Target = TRUE), the name must be a unique number other than 0, which is reserved.

5.2.1 Default values

Setting the token fields to values given below effectively disables it.

INSTRUCTION_EXT_LABEL		
Type	=	INSTRUCTION_EXT_TYPE_LABEL
Label	=	0
Target	=	TRUE
Padding	=	0
Extended	=	FALSE

5.2.2 Setting target label

INSTRUCTION_EXT_LABEL		
Label	=	Target label
Target	=	TRUE

5.2.3 Setting jump label

INSTRUCTION_EXT_LABEL		
Label	=	Instruction label
Target	=	FALSE

5.3 TEXTURE instruction extension

Bits	Name	Description
0:3	Type	
4:11	Texture	Texture target used by texture sample instructions (see Table 24)
12:30	Padding	Must be 0
31	Extended	

Table 23. INSTRUCTION_EXT_TEXTURE token layout

Ordinal	Name	Description
0	TEXTURE_UNKNOWN	
1	TEXTURE_1D	
2	TEXTURE_2D	
3	TEXTURE_3D	
4	TEXTURE_CUBE	
5	TEXTURE_RECT	
6	TEXTURE_SHADOW1D	
7	TEXTURE_SHADOW2D	
8	TEXTURE_SHADOWRECT	

Table 24. TEXTURE enums

5.3.1 Default values

INSTRUCTION_EXT_TEXTURE

Type	=	INSTRUCTION_EXT_TYPE_TEXTURE
Texture	=	TEXTURE_UNKNOWN
Padding	=	0
Extended	=	FALSE

5.3.2 Setting texture target

INSTRUCTION_EXT_TEXTURE

Texture	=	Texture target

5.4 Default values

INSTRUCTION

Type	=	TOKEN_TYPE_INSTRUCTION
Size		variable
Opcode	=	OPCODE_MOV
Saturate	=	SAT_NONE
NumDstRegs	=	1
NumSrcRegs	=	1
Padding	=	0
Extended	=	FALSE

5.5 Instruction setup

This section ignores the SRC_REGISTER and DST_REGISTER tokens for clarity. It mainly focuses on the extension tokens to the INSTRUCTION token.

The following subsections show how to setup instruction tokens for example instructions.

5.5.1 ADD dest, soruce0, source1

```
INSTRUCTION
    Opcode      = OPCODE_ADD
    NumDstRegs  = 1
    NumSrcRegs  = 2
```

5.5.2 MOV_SAT dest, source

```
INSTRUCTION
    Opcode      = OPCODE_MOV
    Saturate    = SAT_ZERO_ONE
    NumDstRegs  = 1
    NumSrcRegs  = 1
```

5.5.3 MOVC dest, source

```
INSTRUCTION
    Opcode      = OPCODE_MOV
    NumDstRegs  = 1
    NumSrcRegs  = 1
    Extended    = TRUE
INSTRUCTION_EXT_NV
    CondDstIndex   = 0
    CondDstUpdate  = TRUE
```

6 Instruction operands

6.1 Dimension index

Bits	Name	Description
0	Indirect	Enable indirect register (see Section ?)
1	Dimension	Enable dimension index
2:14	Padding	Must be 0
15:30	Index	Register index
31	Extended	

Table 25. DIMENSION token layout

6.1.1 Default values

```
DIMENSION
    Indirect   = FALSE
    Dimension  = FALSE
    Padding    = 0
    Index      = 0
    Extended   = FALSE
```

6.2 Source register

Bits	Name	Description
0:3	File	Register file (see Table 9)
4:5	SwizzleX	Register swizzle X (see Table 21)
6:7	SwizzleY	Register swizzle Y (see Table 21)
8:9	SwizzleZ	Register swizzle Z (see Table 21)
10:11	SwizzleW	Register swizzle W (see Table 21)
12	Negate	Register negate
13	Indirect	Indirect register enable
14	Dimension	Enable dimension index (see Section ?)
15:30	Index	Register index
31	Extended	

Table 26. SRC_REGISTER token layout

6.2.1 SWZ source register extension

Bits	Name	Description
0:3	Type	
4:7	ExtSwizzleX	Extended register swizzle X (see Table 28)
8:11	ExtSwizzleY	Extended register swizzle Y (see Table 28)
12:15	ExtSwizzleZ	Extended register swizzle Z (see Table 28)
16:19	ExtSwizzleW	Extended register swizzle W (see Table 28)
20	NegateX	Negate register X
21	NegateY	Negate register Y
22	NegateZ	Negate register Z
23	NegateW	Negate register W
24:27	ExtDivide	Register divisor (see Table 28)
28:30	Padding	Must be 0
31	Extended	

Table 27. SRC_REGISTER_EXT_SWZ token layout

Ordinal	Name	Description
0	EXTSWIZZLE_X	Component X
1	EXTSWIZZLE_Y	Component Y
2	EXTSWIZZLE_Z	Component Z
3	EXTSWIZZLE_W	Component W
4	EXTSWIZZLE_ZERO	Literal 0.0
5	EXTSWIZZLE_ONE	Literal 1.0

Table 28. EXTSWIZZLE enums

6.2.2 MOD source register extension

Bits	Name	Description
0:3	Type	
4	Complement	If TRUE, subtract register from 1.0
5	Bias	If TRUE, subtract 0.5 from register
6	Scale2X	If TRUE, multiply register by 2.0
7	Absolute	If TRUE, remove register sign
8	Negate	If TRUE, negate register
9:30	Padding	Must be 0
31	Extended	

Table 29. SRC_REGISTER_EXT_MOD token layout

6.2.3 Default values

SRC_REGISTER

File	= FILE_NULL
SwizzleX	= SWIZZLE_X
SwizzleY	= SWIZZLE_Y
SwizzleZ	= SWIZZLE_Z
SwizzleW	= SWIZZLE_W
Negate	= FALSE
Indirect	= FALSE
Dimension	= FALSE
Index	= 0
Extended	= FALSE

SRC_REGISTER_EXT_SWZ

Type	= SRC_REGISTER_EXT_TYPE_SWZ
ExtSwizzleX	= EXTSWIZZLE_X
ExtSwizzleY	= EXTSWIZZLE_Y
ExtSwizzleZ	= EXTSWIZZLE_Z
ExtSwizzleW	= EXTSWIZZLE_W
NegateX	= FALSE
NegateY	= FALSE
NegateZ	= FALSE
NegateW	= FALSE
ExtDivide	= EXTSWIZZLE_ONE
Padding	= 0
Extended	= FALSE

SRC_REGISTER_EXT_MOD

Type	= SRC_REGISTER_EXT_TYPE_MOD
Complement	= FALSE
Bias	= FALSE
Scale2X	= FALSE
Absolute	= FALSE
Negate	= FALSE
Padding	= 0
Extended	= FALSE

6.3 Destination register

Bits	Name	Description
0:3	File	Register file (see Table 9)
4:7	WriteMask	Register component update mask (see Table ?)
8	Indirect	Indirect register enable (see Section ?)
9	Dimension	Enable dimension index (see Section ?)
10:25	Index	Register index
26:30	Padding	Must be 0
31	Extended	

Table 30. DST_REGISTER token layout

There must be at most one extended token of each type. If there is more than one extended token of the same type, it is undefined which of them is effectively executed.

6.3.1 CONDCODE destination register extension

Bits	Name	Description
0:3	Type	
4:7	CondMask	Condition code mask (see Table ?)
8:9	CondSwizzleX	Conditional register swizzle X (see Table 21)
10:11	CondSwizzleY	Conditional register swizzle Y (see Table 21)
12:13	CondSwizzleZ	Conditional register swizzle Z (see Table 21)
14:15	CondSwizzleW	Conditional register swizzle W (see Table 21)
16:19	CondSrcIndex	Conditional register index
20:30	Padding	Must be 0
31	Extended	

Table 31. DST_REGISTER_EXT_CONCODE token layout

6.3.2 MODULATE destination register extension

Bits	Name	Description
0:3	Type	
4:7	Modulate	Destination register modulate mode (see Table ?)
8:30	Padding	Must be 0
31	Extended	

Table 32. DST_REGISTER_EXT_MODULATE token layout

6.3.3 Default values

DST_REGISTER

File	= FILE_NULL
WriteMask	= WRITEMASK_XYZW
Indirect	= FALSE
Dimension	= FALSE
Index	= 0
Padding	= 0
Extended	= FALSE

DST_REGISTER_EXT_CONCODE

Type	= DST_REGISTER_EXT_TYPE_CONCODE
CondMask	= CC_TR
CondSwizzleX	= SWIZZLE_X
CondSwizzleY	= SWIZZLE_Y
CondSwizzleZ	= SWIZZLE_Z
CondSwizzleW	= SWIZZLE_W
CondSrcIndex	= 0
Padding	= 0
Extended	= FALSE

DST_REGISTER_EXT_MODULATE

Type	= DST_REGISTER_EXT_TYPE_MODULATE
Modulate	= MODULATE_1X
Padding	= 0
Extended	= FALSE

6.4 Source register setup

This section ignores the INSTRUCTION and DST_REGISTER tokens for clarity.

6.4.1 MOV dest, -INPUT[7].xyyz

SRC_REGISTER

File	= FILE_INPUT
SwizzleX	= SWIZZLE_X
SwizzleY	= SWIZZLE_Y
SwizzleZ	= SWIZZLE_Y
SwizzleW	= SWIZZLE_Z
Negate	= TRUE
Index	= 7

6.4.2 SWZ dest, TEMPORARY[12], 0, -1, x, -w

This sequence of tokens will likely be emitted for languages defined by the following extensions. Note that the language is not limited to the SWZ opcode – it can be issued for any instruction, but the underlying hardware might not necessarily support that.

GL_ARB_vertex_program
GL_ARB_fragment_program
GL_NV_fragment_program_option
GL_NV_fragment_program2
GL_NV_vertex_program2_option
GL_NV_vertex_program3
GL_NV_gpu_program4

SRC_REGISTER

File	= FILE_TEMPORARY
Index	= 12
Extended	= TRUE

SRC_REGISTER_EXT_SWZ

ExtSwizzleX	= EXTSWIZZLE_ZERO
ExtSwizzleY	= EXTSWIZZLE_ONE
ExtSwizzleZ	= EXTSWIZZLE_X
ExtSwizzleW	= EXTSWIZZLE_W
NegateX	= FALSE
NegateY	= TRUE
NegateZ	= FALSE
NegateW	= TRUE

6.4.3 MOV dest, -|TEMPORARY[11]|

SRC_REGISTER

File	= FILE_TEMPORARY
Index	= 11
Extended	= TRUE

SRC_REGISTER_EXT_MOD

Absolute	= TRUE
Negate	= TRUE

Note that there are 2 independent Negate fields in two different tokens. The one contained in the SRC_REGISTER is applied before taking the absolute value of the source register, while the one in SRC_REGISTER_EXT_MOD is applied after the absolute operation. Also, it is legal to have both Negate fields set to TRUE. Depending on the Absolute field's value, they will effectively cancel each other or the first one will be ignored.

6.4.4 MOV dest, CONSTANT[A1.y+17]

This sequence of tokens will likely be emitted for languages defined by the following extensions. Note that the language is not limited to the ADDRESS file – it can be issued for any register file, but the underlying hardware might not necessarily support all of them. In general, older hardware supports ADDRESS file, and modern one – TEMPORARY.

```
    GL_NV_vertex_program
    GL_NV_vertex_program1_1
    GL_NV_vertex_program2
    GL_ARB_vertex_program
    GL_NV_vertex_program2_option
    GL_NV_vertex_program3
SRC_REGISTER
    File     = FILE_CONSTANT
    Indirect = TRUE
    Index    = 17
SRC_REGISTER
    File     = FILE_ADDRESS
    SwizzleX = SWIZZLE_Y
    Index    = 1
```

6.4.5 SWZ dest, CONSTANT_ARRAY[A0.x+2][TEMPORARY[5].w + 9], x, x, x, 1

This example exercises hypothetical 2-D indexing mode of the source operand that can be encoded within the language. Note that FILE_CONSTANT_ARRAY is not currently defined.

The other purpose of this example is to illustrate the order of tokens for complex operands. As a rule of thumb, first goes the extended, then indirect and then dimension token.

```
SRC_REGISTER
    File      = FILE_CONSTANT_ARRAY
    Indirect  = TRUE
    Dimension = TRUE
    Index     = 2
    Extended  = TRUE
SRC_REGISTER_EXT_SWZ
    Type        = SRC_REGISTER_EXT_TYPE_SWZ
    ExtSwizzleX = EXTSWIZZLE_X
    ExtSwizzleY = EXTSWIZZLE_X
    ExtSwizzleZ = EXTSWIZZLE_X
    ExtSwizzleW = EXTSWIZZLE_ONE
SRC_REGISTER
    File     = FILE_ADDRESS
    SwizzleX = SWIZZLE_X
    Index    = 0
DIMENSION
    Indirect = TRUE
    Index    = 9
SRC_REGISTER
    File     = FILE_TEMPORARY
    SwizzleX = SWIZZLE_W
    Index    = 5
```

Note that higher-dimension addressing can easily be implemented by chaining DIMENSION tokens by setting the Dimension flag to TRUE both in SRC_REGISTER and in DIMENSION tokens.

6.4.6 MOV dest, TEMPORARY[TEMPORARY[TEMPORARY[6].y+26].z]

This example exercises hypothetical nested indirect indexing mode of the source operand that can be encoded within the language.

```
SRC_REGISTER
    File      = FILE_TEMPORARY
    Indirect  = TRUE
    Index     = 0
SRC_REGISTER
    File      = FILE_TEMPORARY
    SwizzleX  = SWIZZLE_Z
    Indirect  = TRUE
    Index     = 26
SRC_REGISTER
    File      = FILE_TEMPORARY
    SwizzleX  = SWIZZLE_Y
    Index     = 6
```

6.5 Destination register setup

This section ignores INSTRUCTION and SRC_REGISTER tokens for clarity. Addressing modes that are applicable for both source and destination registers are covered in the source register setup section (see Section 6.4). This section gives examples of usages specific to destination registers only.

6.5.1 MOV TEMPORARY[3].yw, source

```
DST_REGISTER
    File       = FILE_TEMPORARY
    WriteMask  = WRITEMASK_YW
```

6.5.2 MOV TEMPORARY[0] (EQ1.xyww), source

```
DST_REGISTER
    File      = FILE_TEMPORARY
    Index     = 0
    Extended  = TRUE
    CondMask      = CC_EQ
    CondSwizzleX  = SWIZZLE_X
    CondSwizzleY  = SWIZZLE_Y
    CondSwizzleZ  = SWIZZLE_W
    CondSwizzleW  = SWIZZLE_W
    CondSrcIndex  = 1
```

7 Instruction execution

7.1 Arithmetic vector instructions

These operate component-wise. The computation is executed for every component.

ABS	DEST.c = abs(SRC0.c)
ADD	DEST.c = SRC0.c + SRC1.c
FLR	DEST.c = floor(SRC0.c)
FRC	DEST.c = frac(SRC0.c)
LRP	DEST.c = SRC0.$c \cdot$ SRC1.c + (1.0 − SRC0.c) $\cdot$ SRC2.c
MAD	DEST.c = SRC0.$c \cdot$ SRC1.c + SRC2.c
MOV	DEST.c = SRC0.c
MUL	DEST.c = SRC0.$c \cdot$ SRC1.c
SUB	DEST.c = SRC0.c − SRC1.c

7.1.1 Instruction emulation

The ABS instruction can be emulated using the MOV instruction.

ABS	MOV	DEST, |SRC0|

The ABS, ADD, MOV, MUL and SUB instructions can be emulated using the MAD instruction.

ABS	MAD	DEST, |SRC0|, SRC0{1,1,1,1}, SRC0{0,0,0,0}
ADD	MAD	DEST, SRC0, SRC0{1,1,1,1}, SRC1
MOV	MAD	DEST, SRC0, SRC0{1,1,1,1}, SRC0{0,0,0,0}
MUL	MAD	DEST, SRC0, SRC1, SRC1{0,0,0,0}
SUB	MAD	DEST, SRC0, SRC0{1,1,1,1}, -SRC1

The SUB instruction can be emulated using the ADD instruction.

SUB	ADD	DEST, SRC0, -SRC1

7.2 Comparison vector instructions

These operate component-wise. The computation is executed for every component.