The SPIR™ Specification

Standard Portable Intermediate Representation

Version 1.2

Revision Date: 2014-01-21

Editor: Boaz Ouriel

Contents

List of Tables

Acknowledgements

Editor: Boaz Ouriel, Intel

Contributors:

- David Neto, Altera
- Anton Lokhmotov, ARM
- Mike Houston, AMD
- Micah Villmow, AMD
- Brian Sumner, AMD
- Richard Relph, AMD
- Yaxun Liu, AMD
- Tanya Lattner, Apple
- Aaftab Munshi, Apple
- Holger Waechtler, Broadcom
- Christopher Thomson-Walsh, Broadcom
- Andrew Richards, Codeplay
- Guy Benyei, Intel
- Javier E. Martinez, Intel
- Boaz Ouriel, Intel
- Dillon Sharlet, Intel
- Vinod Grover, NVIDIA
- Kedar Patil, NVIDIA
- Sumesh Udayakumaran, QUALCOMM
- Chihong Zhang, QUALCOMM
- James Price, University of Bristol
- Henry Styles, Xilinx

1 Introduction

This document defines version 1.2 of the Standard Portable Intermediate Representation (SPIR). SPIR 1.2 is a mapping from the OpenCL C programming language into LLVM IR.

This version of the specification is based on LLVM 3.2 [4] [3], and on OpenCL C as specified in the OpenCL 1.2 Specification [2].

The goal of SPIR 1.2 is to provide a portable interchange format for partly compiled OpenCL C programs. The format:

- Is vendor neutral.
- Is not C source code.
- Supports almost all core features and KHR extensions for version 1.2 of OpenCL C. (A small number of features of OpenCL C are not expressible in SPIR.)
- Is designed to support vendor extensions.
- Is compact.
- Is designed to be efficiently loaded by an OpenCL implementation.
- Is designed to be useful as a target format for compilers of programming languages other than OpenCL C. This is a secondary goal of SPIR.

1.1 One format, two notations

LLVM IR has three semantically equivalent representations:

- An in-memory data structure manipulated by the LLVM software.
- A compact external binary representation, known as *bitcode* [3]. [1]
- A human readable assembly language notation [4].

SPIR adopts two of these: the bitcode and assembly language notations from LLVM. For ease of exposition, the remainder of this document uses only the assembly language notation.

1.2 Name mangling

OpenCL C has many overloaded built-in functions, meaning the same function name is used with different argument and return types. For example, the `sin` built-in function is defined for both scalar and vector floating point argument and return types. SPIR distinguishes between all of the variations of the `sin` function by mangling the root name `sin` with its argument types.

This means that in SPIR all of the OpenCL C built-in functions are mangled based on their argument types.

Other kinds of names are not mangled in SPIR. In particular, regular and kernel user functions from OpenCL C are not mangled when mapped into SPIR.

By *not* mangling the names of regular functions, SPIR supports being the target for language families (other than C/C++) having their own distinctive type systems. In other words, mangling of user-level functions is beyond the scope of SPIR, and is subject to coordination among third parties (compiler front end and library implementors).

For names that do require mangling, SPIR adopts and extends the name mangling scheme from Section 5.1 of the Itanium C++ ABI [1]. Extensions are required to support OpenCL concepts absent from ordinary C++. The SPIR mangling scheme is defined in Appendix A.

[1] The LLVM 3.2 bitcode notation is only partly documented by [3]. However, bitcode notation is fully (but implicitly) defined by the behaviour of LLVM 3.2 software release.

2 OpenCL C mapping to SPIR

2.1 Supported Data Types

The following LLVM data types are supported:

2.1.1 Built-in Scalar Data Types

Table 1 describes the mapping from the OpenCL C built-in scalar data types to SPIR built-in scalar data types.

OpenCL C Type	**LLVM Type**
bool	i1
char	i8
unsigned char, uchar	i8
short	i16
unsigned short, ushort	i16
int	i32
unsigned int, uint	i32
long	i64
unsigned long, ulong	i64
float	float
double	double
half	half
void	void

Table 1: Mapping for built-in scalar data types

Notes:

- Signed and unsigned values are sign extended or zero extended based on the deployed operation.
- While LLVM has many more primitive data types, only the ones described above are allowed in SPIR.

2.1.2 Built-in Vector Types

Table 2 describes the mapping from the OpenCL C built-in vector data types to SPIR built-in scalar data types. Supported values of n are 2, 3, 4, 8, and 16 for all vector data types.

OpenCL C Type	**LLVM Type**
char*n*	< n x i8 >
uchar*n*	< n x i8 >
short*n*	< n x i16 >
ushort*n*	< n x i16 >
int*n*	< n x i32 >
uint*n*	< n x i32 >
long*n*	< n x i64 >
ulong*n*	< n x i64 >
half*n*	< n x half >
float*n*	< n x float >
double*n*	< n x double >

Table 2: Mapping for built-in vector types

Note: LLVM supports many more vector data types, however only the ones described above are allowed in SPIR. Specifically, a vector of `i1`'s is disallowed in SPIR.

2.1.3 Other Built-in Data Types

Table 3 defines the mapping of OpenCL images, sampler, events, size_t, ptrdiff_t,uintptr_t,intptr_t data types to LLVM data types

OpenCL C Type	**LLVM Type**	**LLVM Name**
image1d_t	opaque*	%opencl.image1d_t
image1d_array_t	opaque*	%opencl.image1d_array_t
image1d_buffer_t	opaque*	%opencl.image1d_buffer_t
image2d_t	opaque*	%opencl.image2d_t
image2d_array_t	opaque*	%opencl.image2d_array_t
image3d_t	opaque*	%opencl.image3d_t
image2d_msaa_t	opaque*	%opencl.image2d_msaa_t
image2d_array_msaa_t	opaque*	%opencl.image2d_array_msaa_t
image2d_msaa_depth_t	opaque*	%opencl.image2d_msaa_depth_t
image2d_array_msaa_depth_t	opaque*	%opencl.image2d_array_msaa_depth_t
image2d_depth_t	opaque*	%opencl.image2d_depth_t
image2d_array_depth_t	opaque*	%opencl.image2d_array_depth_t
event_t	opaque*	%opencl.event_t
sampler_t	i32	N/A
size_t	i32 or i64	N/A
ptrdiff_t	i32 or i64	N/A
uintptr_t	i32 or i64	N/A
intptr_t	i32 or i64	N/A

Table 3: Mapping for other built-in data types

Notes:

- The size of images and event data types is equal to 32 bits or 64 bits according to the device address width.

- The names given to opaque data types are reserved for SPIR and shall not be used otherwise.
- The OpenCL `size_t`,`ptrdiff_t`,`uintptr_t` and `intptr_t` data types are mapped to LLVM `i32` when the device address width is equal to 32 bits and to LLVM `i64` when the device address width is equal 64 bits
- i32 values that represent `sampler_t` objects, can only be passed as arguments to images built-ins. Any other operation involving these i32 values is implementation defined.

2.1.3.1 Declaring sampler variables

A sampler variable is an i32 constant-qualified module scope variable in the constant address space, initialized with an i32 constant value. The i32 constant value is interpreted as a bit-field specifying the following properties:

Sampler State	**Init Values**
addressing mode	CLK_ADDRESS_NONE=0x0000 CLK_ADDRESS_CLAMP_TO_EDGE=0x0002 CLK_ADDRESS_CLAMP=0x0004 CLK_ADDRESS_REPEAT=0x0006 CLK_ADDRESS_MIRRORED_REPEAT=0x0008
normalized coords	CLK_NORMALIZED_COORDS_FALSE=0x0000 CLK_NORMALIZED_COORDS_TRUE=0x0001
filter mode	CLK_FILTER_NEAREST=0x0010 CLK_FILTER_LINEAR= 0x0020

Table 4: sampler initialization values

2.1.3.2 Image channel data type values

The get_image_channel_data_type() built-in returns an integer value which represents the image channel data type. The following table indicates the valid values:

Channel order	**Value**
CLK_SNORM_INT8	0x10D0
CLK_SNORM_INT16	0x10D1
CLK_UNORM_INT8	0x10D2
CLK_UNORM_INT16	0x10D3
CLK_UNORM_SHORT_565	0x10D4
CLK_UNORM_SHORT_555	0x10D5
CLK_UNORM_INT_101010	0x10D6
CLK_SIGNED_INT8	0x10D7
CLK_SIGNED_INT16	0x10D8
CLK_SIGNED_INT32	0x10D9
CLK_UNSIGNED_INT8	0x10DA
CLK_UNSIGNED_INT16	0x10DB
CLK_UNSIGNED_INT32	0x10DC
CLK_HALF_FLOAT	0x10DD
CLK_FLOAT	0x10DE
CLK_UNORM_INT24	0x10DF

Table 5: image channel data type values

2.1.3.3 Image channel order values

The get_image_channel_order() built-in returns an integer value which represents the image channel order. The following table indicates the valid values:

Channel order	**Value**
CLK_R	0x10B0
CLK_A	0x10B1
CLK_RG	0x10B2
CLK_RA	0x10B3
CLK_RGB	0x10B4
CLK_RGBA	0x10B5
CLK_BGRA	0x10B6
CLK_ARGB	0x10B7
CLK_INTENSITY	0x10B8
CLK_LUMINANCE	0x10B9
CLK_Rx	0x10BA
CLK_RGx	0x10BB
CLK_RGBx	0x10BC
CLK_DEPTH	0x10BD
CLK_DEPTH_STENCIL	0x10BE

Table 6: image channel order values

2.1.3.4 Zero events

Zero events are represented using the LLVM `null` keyword.

2.1.3.5 NULL pointer

NULL pointers are represented using the LLVM `null` keyword.

2.1.4 Alignment of Types

SPIR follows the alignment rules of OpenCL. Therefore:

- Stack allocations and module scope variable declarations must follow the alignment rules defined in OpenCL specification.
- All `load` and `store` operations need to be aligned.

2.1.5 Structs

The alignment of structures data members is the alignment of the SPIR data type. Extra padding is disallowed. The alignment of the structure is the alignment of the member which requires the largest alignment.

When mapping an OpenCL C struct data type to SPIR, the order of members shall be preserved.

2.2 Address space qualifiers

OpenCL C address spaces are mapped to the LLVM `addrspace(n)` qualifier using the following convention:

- 0 – private
- 1 – global
- 2 – constant
- 3 – local

Note: Casts between address spaces is disallowed in SPIR.

Note: Each OpenCL C function-scope local variable is mapped into an LLVM module-level variable in address space 3. They are not allocated using `alloca` instruction. The name of the module-level variable consists of the function name, followed by a period, followed by the the source identifier.

Example OpenCL C program:

```
void foo(void) {
  local float4 lf4;
}
```

A valid SPIR mapping:

```
; Unmangled component names shown here.
; float4 must be 16 bytes aligned.
@foo.lf4 = internal addrspace(3) global <4 x float> zeroinitializer, align 16

define spir_kernel void @foo() nounwind {
entry:
  ret void
}
```

In OpenCL C, a kernel function can call another kernel. However, when the called kernel declares a variable in the __local address space, then the behaviour is implementation defined. SPIR supports a kernel calling another kernel, but does not allow the called kernel to have a variable in the __local address space. For example, the following example is not valid SPIR:

```
@bar.lf4 = internal addrspace(3) global <4 x float> zeroinitializer, align 16

define spir_kernel void @bar() nounwind {
entry:
  ret void
}

define spir_kernel void @callbar() nounwind {
entry:
  call spir_kernel void @bar()  ; This is not supported by SPIR
  ret void
}
```

2.3 Kernel qualifiers

Adding qualifiers and attributes to a kernel and its arguments is achieved by usage of the LLVM metadata infrastructure. Each SPIR module has a `opencl.kernels` named metadata node containing a list of metadata objects. Each metadata object in `opencl.kernels` references a list of metadata objects, each of which represents a single kernel. The first value in a SPIR function metadata object is the SPIR function that represents an OpenCL kernel. The rest of the metadata objects are additional attributes and information which is attached to the SPIR function. The description of each metadata object inside the SPIR function metadata list is described in the other sections.

The following LLVM textual representation shows how SPIR function attributes are represented:

`!opencl.kernels = !{ !0,!1,...,!N }`
`; Note:  The first element is always an LLVM::Function signature`
`!0 = metadata !{` $<$ *function signature* $>$`, ` $!0_1$`, ` $!0_2$`, ..., , ` $!0_i$ `}`
`!1 = metadata !{` $<$ *function signature* $>$`, ` $!1_1$`, ` $!1_2$`, ..., , ` $!1_j$ `}`
...
`!N = metadata !{` $<$ *function signature* $>$`, ` $!N_1$`, ` $!N_2$`, ..., , ` $!N_k$ `}`

2.3.1 Optional attribute qualifiers

2.3.1.1 Work group size information

Attaching `work_group_size_hint` and `reqd_work_group_size` information to kernels is achieved using LLVM metadata infrastructure. Two new metadata object are introduced. The first item in the metadata object is the string `"work_group_size_hint"` or `"reqd_work_group_size"` followed by three `i32` constant values. The three `i32` values specify the `(X,Y,Z)` group dimensions.

```
; work_group_size_hint(128,1,1)
!0 = metadata !{ metadata !"work_group_size_hint", i32 128, i32 1, i32 1}
; reqd_work_group_size(128,1,1)
!1 = metadata !{ metadata !"reqd_work_group_size", i32 128, i32 1, i32 1}
```

Note:

- Attaching the work group size hint to a non-kernel SPIR function is invalid.

2.3.1.2 Vector type hint information

Attaching `vec_type_hint` information to kernels is achieved using LLVM metadata infrastructure. The first argument in each metadata object is the string `"vec_type_hint"` followed by a typed `undef` LLVM value and an additional `i1` value representing the signedness of the value.

```
; vec_type_hint(float)
!0 = metadata !{ metadata !"vec_type_hint", float undef, i1 1}
; vec_type_hint(uint8)
!1 = metadata !{ metadata !"vec_type_hint", <8 x i32> undef, i1 0}
...
; vec_type_hint(<type>)
!H = metadata !{ metadata !"vec_type_hint", <type> undef, i1 isSigned}
```

Note:

- Attaching vector type hint information to a non-kernel SPIR function is invalid.
- The `double` data type is an optional type and using it requires marking the SPIR module as using the `cl_doubles` optional core feature. See Section 2.11.1.

2.4 Kernel Arg Info

Kernel argument specific information is preserved using metadata objects. These objects are generated for every kernel, with an exception for the `kernel_arg_name` metadata, which is generated only when the `-cl-kernel-arg-info` build option is specified for compilation. The metadata nodes describing the kernel argument info are in the form of a string tag, and then a list of the corresponding data for each one of the kernel's arguments.

The following table shows the valid kernel argument information types and values:

ARG Info	**Type**	**Values**
`"kernel_arg_addr_space"`	i32	0 – private 1 – global 2 – constant 3 – local
`"kernel_arg_access_qual"`	string metadata	"read_only" "write_only" "read_write" "none"
`"kernel_arg_type"`	string metadata	The type name specified for the argument. The type name will be the argument type name as it was declared with any whitespace removed. If argument type name is an unsigned scalar type (i.e. unsigned char, unsigned short, unsigned int, unsigned long), uchar, ushort, uint and ulong will be returned. The argument type name returned does not include any type qualifiers.
`"kernel_arg_base_type"`	string metadata	The base type name of the argument. The type name will be identical to the kernel_arg_type metadata, except for types derived from a single OpenCL built-in type (typedef). In this case the name of the OpenCL built-in type will be used.
`"kernel_arg_type_qual"`	string metadata	"const" "restrict" "volatile" or a single space separated combination of these.
`"kernel_arg_name"`	string metadata	the name specified for the argument. Generated only when the -cl-kernel-arg-info build option is specified for compilation.

Table 7: Kernel Arg Info metadata description

Note: Images data types reside in global memory and hence should be marked as such in the `"kernel_arg_addr_space"` metadata.

Example:

```
typedef sampler_t mySampler;
__kernel void helloworld(__global char* in, __global char* out, mySampler s);

!opencl.kernels = !{!0}

!0 = metadata !{void (i8 addrspace(1)*, i8 addrspace(1)*, i32)* @helloworld, metadata !1,
                           metadata !2, metadata !3, metadata !4, metadata !5, metadata !6}
!1 = metadata !{metadata !"kernel_arg_address_space", i32 1, i32 1, i32 0}
!2 = metadata !{metadata !"kernel_arg_access_qual", metadata !"none", metadata !"none",
                           metadata !"none"}
!3 = metadata !{metadata !"kernel_arg_type", metadata !"char*", metadata !"char*",
                           metadata !"mySampler"}
!4 = metadata !{metadata !"kernel_arg_base_type", metadata !"char*", metadata !"char*",
                           metadata !"sampler_t"}
```

```
!5 = metadata !{metadata !"kernel_arg_type_qual", metadata !"", metadata !"", metadata !""}
!6 = metadata !{metadata !"kernel_arg_name", metadata !"in", metadata !"out", metadata !"s"}
```

2.5 Storage class specifier

The OpenCL C `extern` and `static` storage class specifiers map to the LLVM `external` and `internal` linkage types, respectively.

2.6 Type qualifiers

OpenCL C Type Qualifier	**LLVM Mapping**
const	constant
restrict	noalias
volatile	Certain memory accesses, such as loads, stores, and SPIR memcpys may be marked volatile. (See Notes below.)

Table 8: Mapping of type qualifiers

Notes for the `volatile` qualifier:

1. The optimizers must not change the number of volatile operations or change their order of execution relative to other volatile operations.
2. The optimizers may change the order of volatile operations relative to non-volatile operations.

2.7 Attribute Qualifiers

2.7.1 Type Attributes

SPIR provides structure types to describe unions and structures. The layout of structures in SPIR must take into consideration the alignment rules of OpenCL C. Optimizers are not allowed to do any modifications to structures.

2.7.1.1 aligned attribute

SPIR structures can be aligned at declaration time. This applies both to module level structures and stack allocations using the `alloca` instruction.

2.7.1.2 packed attribute

SPIR structures are marked as packed when `__attribute__((packed))` is used in OpenCL C.

`Example:`

<{i8 , i32}> is a packed structure known to be 5 bytes in size.

2.7.2 Variable Attributes

2.7.2.1 aligned attribute

- SPIR variables can be aligned at declaration time. This applies both to module level variables and stack allocations using the `alloca` instruction.

- SPIR does not provide a mechanism to reflect the alignment of structure members. Instead the SPIR generator is expected to create a structure definition taking into consideration this attribute, for example by inserting dummy members to occupy the extra space. Optimizers are not allowed to modify the data layout of structures.

2.8 Compiler Options

Compiler options are represented in SPIR using a named metadata node `opencl.compiler.options`. The named metadata node will contain a single metadata node that holds a list of string metadata objects. Each string metadata object corresponds to a single standard OpenCL compiler option. Preprocessor options are not saved in SPIR and the list of the allowed options are as follows:

- -cl-single-precision-constant
- -cl-denorms-are-zero
- -cl-fp32-correctly-rounded-divide-sqrt
- -cl-opt-disable
- -cl-mad-enable
- -cl-no-signed-zeros
- -cl-unsafe-math-optimizations
- -cl-finite-math-only
- -cl-fast-relaxed-math
- -w
- -Werror
- -cl-kernel-arg-info

Note: The `-cl-std` option is propagated to the `opencl.ocl.version` as defined in Section 2.13, OpenCL Version.

This example indicates that both `-cl-mad-enable` and `-cl-denorms-are-zero` standard compile options were used to compile the module:

```
!opencl.compiler.options = !{!2}
!2 = metadata !{metadata !"-cl-mad-enable", metadata !"-cl-denorms-are-zero"}
```

Compilation options which are not part of the OpenCL specification are stored via the named metadata node `opencl.compiler.ext.options`. The named metadata node contains a single metadata node that holds a list of string metadata objects. Each string metadata object corresponds to a non-standard compile option. Compilation options which appear in `opencl.compiler.ext.options` shall not affect functional portability of the SPIR module.

This example indicates that the (hypothetical) non-standard option `-opt-arch-pdp11` was used to compile the module:

```
!opencl.compiler.ext.options = !{!5}
!5 = metadata !{metadata !"-opt-arch-pdp11"}
```

2.9 Preprocessor Directives and Macros

It is the SPIR generator's responsibility to handle all preprocessor responsibilities including macro substitution.

2.9.1 Floating point contractions

The named metadata `opencl.enable.FP_CONTRACT` can be used to enable contractions at module level. If the named metadata node exists, contractions can be generated by a SPIR optimizer at module level.

Note: This is a case where OpenCL C allows finer grained optimisation than SPIR, since it allows the selective enabling of floating point contraction for only certain calculations within a compilation unit.

SPIR can nevertheless express these programs. Since FP_CONTRACT only relaxes precision requirements, OpenCL C programs that use FP_CONTRACT selectively can still be safely and legally represented as more precise SPIR programs without FP_CONTRACT. However, such a program will not necessarily have the same performance or identical rounding and precision as the original on any particular platform.

2.10 Built-ins

2.10.1 Name Mangling

All of the built-in names described in this document are shown in their unmangled form.

2.10.2 Synchronization Functions

Synchronization functions accept `cl_mem_fence_flags` enumeration as an argument. In SPIR this maps to a constant `i32` value which is a bitwise OR between CLK_LOCAL_MEM_FENCE = 1 and CLK_GLOBAL_MEM_FENCE = 2.

Note: The legal values are 1, 2, and 3

2.10.3 The printf function

The `printf` function is supported, and is mangled according to its prototype as follows:

```
int printf(constant char * restrict fmt, ... )
```

Note that the ellipsis formal argument (`...`) is mangled to argument type specifier `z`.

In SPIR the conversion specifiers e,E,g,G,a,A require a double type argument to be passed to the function printf. Thus a `float` or `half` argument that is a scalar type should be explicitly converted to a `double`. A device that doesn't support the `double` data type shall disregard this explicit conversion, or replace the conversion with a conversion to a `float` data type in the case of a `half` data type argument.

The presence of this conversion alone is not enough to force the listing of `"cl_doubles"` as a "used optional core features" for this SPIR instance.

2.11 KHR Extensions

2.11.1 Declaration of used optional core features

The named metadata object `opencl.used.optional.core.features` contains a single metadata object. The metadata object should contain a list of metadata strings, each of which encodes the name of an optional core feature used by the SPIR module.

This is the list of valid strings and their meaning:

- `"cl_images"` - indicates that images are used
- `"cl_doubles"` - indicates that doubles are used

A device may reject a SPIR module using an unsupported optional core feature.
This example indicates that the module uses both images and doubles.

```
!opencl.used.optional.core.features = !{!0}
!0 = metadata !{metadata !"cl_doubles", metadata !"cl_images"}
```

2.11.2 Declaration of used KHR extensions

A SPIR module using one or more KHR extension, must declare them inside the SPIR module. The named metadata object `opencl.used.extensions` is used to declare this list. The named metadata object contains a metadata object consisting of a list of metadata strings, where each string indicates a usage of a KHR extension inside the SPIR module.

This is the list of extension strings:

- cl_khr_int64_base_atomics
- cl_khr_int64_extended_atomics
- cl_khr_fp16
- cl_khr_gl_sharing
- cl_khr_gl_event
- cl_khr_d3d10_sharing
- cl_khr_media_sharing
- cl_khr_d3d11_sharing
- cl_khr_global_int32_base_atomics
- cl_khr_global_int32_extended_atomics
- cl_khr_local_int32_base_atomics
- cl_khr_local_int32_extended_atomics
- cl_khr_byte_addressable_store
- cl_khr_3d_image_writes
- cl_khr_gl_msaa_sharing
- cl_khr_depth_images
- cl_khr_gl_depth_images

This example shows that `cl_khr_fp16` and `cl_khr_int64_base_atomics` standard extensions are used in the module.

```
!opencl.used.extensions  = !{!6}
!6 = metadata !{metadata !"cl_khr_fp16", metadata !"cl_khr_int64_base_atomics"}
```

Notes:

- A device may reject a SPIR module using an unsupported KHR extension.
- A device using `cl_khr_3d_image_writes` must also declare its use of `cl_images` inside `opencl.used.optional.core.features`.
- **cl_khr_fp64** doesn't exist in SPIR. Instead SPIR generators should use the `cl_doubles` optional core features.

2.12 SPIR Version

The SPIR version used by the module is stored in the `opencl.spir.version` named metadata. The named metadata contains a metadata node consisting of a list of two `i32` constant values denoting the major and minor version numbers.

The following example indicates the module uses SPIR version 1.2:

```
!opencl.spir.version = !{!3}
!3 = metadata !{i32 1, i32 2}
```

2.13 OpenCL Version

The OpenCL version used by the module is stored in the `opencl.ocl.version` named metadata node. The named metadata node contains a metadata node consisting of a list of two `i32` constant values denoting the major and minor version numbers.

This example indicates the module is compiled for OpenCL 1.0:

```
!opencl.ocl.version = !{!4}
!4 = metadata !{i32 1, i32 0}
```

This example indicates the module is compiled for OpenCL 1.1:

```
!opencl.ocl.version = !{!4}
!4 = metadata !{i32 1, i32 1}
```

2.14 memcpy functions

The usage of LLVM memcpy and memset intrinsics is allowed in SPIR

2.15 Restrictions

Restrictions from OpenCL C also apply to programs represented in SPIR.

Also, recall that use of FP_CONTRACT is encoded at the module level. See Section 2.9.1 for a discussion of how this limits what OpenCL programs may be represented in SPIR.

3 SPIR and LLVM IR

3.1 LLVM Triple

SPIR introduces a couple of new LLVM triples called "`spir-unknown-unknown`" and "`spir64-unknown-unknown`"

```
target triple = "spir-unknown-unknown"
target triple = "spir64-unknown-unknown"
```

"`spir`" targets devices with address width of 32 bits. "`spir64`" targets devices with address width of 64 bits.

3.2 LLVM Target data layout

The `spir` triple datalayout is as follows:

```
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
                   f32:32:32-f64:64:64-v16:16:16-v24:32:32-v32:32:32-v48:64:64-
                   v64:64:64-v96:128:128-v128:128:128-v192:256:256-v256:256:256-
                   v512:512:512-v1024:1024:1024"
```

The `spir64` triple datalayout is as follows:

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
                    f32:32:32-f64:64:64-v16:16:16-v24:32:32-v32:32:32-v48:64:64-
                    v64:64:64-v96:128:128-v128:128:128-v192:256:256-v256:256:256-
                    v512:512:512-v1024:1024:1024"
```

3.3 LLVM Supported Instructions

The following tables show which LLVM instructions are may be used in SPIR:

LLVM Instruction Family	**Instruction name**	**Supported**
Terminator	ret	yes
Terminator	br	yes
Terminator	switch	yes
Terminator	indirectbr	no, required for GNU extension (array of pointer of functions)
Terminator	invoke	no, exception handling related
Terminator	unwind	no, exception handling related
Terminator	resume	no, exception handling related
Terminator	unreachable	yes, might be used for switch statements
Binary	add	yes
Binary	fadd	yes
Binary	sub	yes
Binary	fsub	yes
Binary	mul	yes
Binary	fmul	yes
Binary	udiv	yes
Binary	sdiv	yes
Binary	fdiv	yes
Binary	urem	yes
Binary	srem	yes
Binary	frem	yes
Bitwise Binary	shl	yes, left-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value
Bitwise Binary	lshr	yes, right-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value.
Bitwise Binary	ashr	yes, right-shifted by log2(N), where N is the number of bits used to represent the data type of the shifted value. `exact` is disallowed and used for trap values
Bitwise Binary	and	yes
Bitwise Binary	or	yes
Bitwise Binary	xor	yes
Vector	extractelement	yes
Vector	insertelement	yes
Vector	shufflevector	yes
Aggregate	extractvalue	yes
Aggregate	insertvalue	yes
Memory Access & Addressing	alloca	yes
Memory Access & Addressing	load	yes, `atomic` is disallowed
Memory Access & Addressing	store	yes, `atomic` is disallowed
Memory Access & Addressing	fence	no, use built-ins instead
Memory Access & Addressing	cmpxchg	no, use built-ins instead
Memory Access & Addressing	atomicrmw	no, use built-ins instead
Memory Access & Addressing	getelementptr	yes

Table 9: Instructions, part 1

LLVM Instruction Family	**Instruction name**	**Supported**
Conversion Operations	trunc .. to	yes, but only for scalars
Conversion Operations	zext .. to	yes, but only for scalars
Conversion Operations	sext .. to	yes, but only for scalars
Conversion Operations	fptrunc .. to	yes, but only for scalars
Conversion Operations	fpext .. to	yes, but only for scalars
Conversion Operations	fptoui .. to	yes, but only for scalars
Conversion Operations	fptosi .. to	yes, but only for scalars
Conversion Operations	uitofp .. to	yes, but only for scalars
Conversion Operations	sitofp .. to	yes, but only for scalars
Conversion Operations	ptrtoint .. to	yes
Conversion Operations	inttoptr .. to	yes
Conversion Operations	bitcast .. to	yes
Other Operations	icmp	yes
Other Operations	fcmp	yes
Other Operations	phi	yes
Other Operations	select	yes
Other Operations	call	yes, but not to pointers to functions
Other Operations	va_arg	no, not supported by OpenCL
Other Operations	landingpad_arg	no

Table 10: Instructions, part 2

3.4 LLVM Supported Intrinsic Functions

None of the LLVM intrinsics are allowed in SPIR except the memcpy intrinsics.

3.5 SPIR ABI

In this section we define the application binary interface for OpenCL "C" programs in SPIR. The SPIR ABI defines the interfaces between the SPIR program and the OpenCL runtime, built-ins libraries and additional third party SPIR libraries.

Each function argument and return type is classified as follows:

- Any aggregate type is passed as a pointer. Memory allocation (if needed) is the responsibility of the caller function.
- Enumeration types are handled as the underlying integer type.
- If the argument type is a promotable integer type, it will be extended according to the C99 integer promotion rules.
- Any other type, including floating point types, vectors, etc.. will be passed directly as the corresponding LLVM type.

Note: The ABI described in this section is implemented in Clang 3.2 and is called the "default" ABI.

3.6 LLVM Linkage Types

The following table shows the LLVM linkage types allowed in SPIR:

Linkage type	**Supported**
private	yes
linker_private	no
linker_private_weak	no
linker_private_weak_def_auto	no
available_externally	yes (describes C99 inline definition)
linkonce	no
internal	yes (maps to static)
weak	no
common	yes
appending	no
extern_weak	no
linkonce_odr	no
weak_odr	no
external	yes
dllimport	no
dllexport	no

Table 11: Linkage types

In addition, SPIR allows the usage of LLVM `unnamed_addr` optional attribute for both global variables and functions.

3.7 Calling Conventions

SPIR kernels should use `"spir_kernel"` calling convention. Non-kernel functions use `"spir_func"` calling convention. All other calling conventions are disallowed.

3.8 Visibility Styles

Visibility styles are not used in SPIR and should be set to **"default"**. Other values are disallowed.

3.9 Parameter Attributes

The following table defines which parameter attributes are usable in SPIR:

Parameter Attribute	**Supported**
zeroext	yes
signext	yes
inreg	no
byval	yes
sret	yes
nocapture	yes
nest	no

Table 12: Parameter attributes

3.10 Garbage Collection Names

Garbage collection is not part of SPIR, hence functions are not allowed to specify a garbage collector name.

3.11 Function Attributes

Every SPIR function should use the `nounwind` attribute. In addition the following optional attributes could be used: `alwaysinline`, `inlinehint`, `noinline`, `readnone`, `readonly`. The rest of the function attributes are disallowed.

Function Attribute	Supported
alignstack	no
alwaysinline	yes
nonlazybind	no
inlinehint	yes
naked	no
noimplicitfloat	no
noinline	yes
noredzone	no
noreturn	no
nounwind	yes, needs to be always set
optsize	no
readnone	yes
readonly	yes
ssp	no
sspreq	no
uwtable	no
returns_twice	no

Table 13: Function attributes

3.12 Reserved identifiers

All identifiers that begin with `opencl.*` are reserved and shall not be used by SPIR generators (for user source identifiers).

3.13 Module Level Inline Assembly

LLVM module level inline assembly is not allowed in SPIR.

3.14 Pointer Aliasing Rules

SPIR follows the pointer aliasing rules of LLVM.

3.15 Volatile Memory Accesses

SPIR requires use of volatile memory accesses and follows LLVM IR rules for `load`'s, `store`'s, `llvm.memcpy`'s and `llvm.memset`'s.

3.16 Memory Model for Concurrent Operations

SPIR does not use the LLVM atomic intrinsics, because OpenCL has its own set of intrinsics.

3.17 Atomic Memory Ordering Constraints

The LLVM atomic orderings are disallowed in SPIR.

A SPIR name mangling

In order to support cross device compatibility of SPIR, the name mangling scheme must be standardized across vendors. SPIR adopts and extends the name mangling scheme in Section 5.1 of the Itanium C++ ABI [1]. There are three major issues to deal with, along with many minor items. The major items are data types, address spaces, and overloaded 'C' functions.

Normally, 'C' functions require no overloading, and their names are not mangled. When generating SPIR, OpenCL C built-in functions must use this mangling scheme.

A.1 Data types

The following table shows the mapping from OpenCL C data types to the type names used in the mangling scheme:

OpenCL C type	**Mangling scheme type name**
bool	b
unsigned char, char	h
char	c
unsigned short, short	t
short	s
unsigned int, uint	j
int	i
unsigned long, ulong	m
long	l
half	Dh
float	f
double	d
pointer to private address space	P<*mangled-element-type-name*>
pointer to non private address space	PU3AS*N*<*mangled-element-type-name*> (where N is the address space number)
Vector types with N elements	Dv*N*_<*mangled-element-type-name*> (where N is one of 2, 3, 4, 8, 16)
image1d_t	11ocl_image1d
image1d_array_t	16ocl_image1darray
image1d_buffer_t	17ocl_image1dbuffer
image2d_t	11ocl_image2d
image2d_array_t	16ocl_image2darray
image3d_t	11ocl_image3d
image2d_msaa_t	15ocl_image2dmsaa
image2d_array_msaa_t	20ocl_image2darraymsaa
image2d_msaa_depth_t	20ocl_image2dmsaadepth
image2d_array_msaa_depth_t	25ocl_image2darraymsaadepth
image2d_depth_t	16ocl_image2ddepth
image2d_array_depth_t	21ocl_image2darraydepth
event_t	9ocl_event
sampler_t	11ocl_sampler
size_t, uintptr_t	treated as uint or ulong
ptrdiff_t, intptr_t	treated as int or long

Table 14: Mapping of OpenCL C builtin type names to mangled type names

A.2 The restrict qualifier

The Itanium ABI states:

> The restrict qualifier is part of the C99 standard, but is strictly an extension to C++ at this time. There is no standard specification of whether the restrict attribute is part of the type for overloading purposes. An implementation should include its encoding in the mangled name if and only if it also treats it as a distinguishing attribute for overloading purposes. This ABI does not specify that choice."

SPIR encodes the "restrict" qualifier as part of the mangled name using the 'r' token in the CV-qualifiers. Hence SPIR treats the "restrict" qualifier as significant for overloading.

A.3 Summary of changes

The following is a summary of the mangling of builtin types:

```
<builtin-type> ::= v  # void (Maps to OpenCL void)
          ::= w   # wchar_t (*Not valid)
          ::= b   # bool (Maps to OpenCL bool)
          ::= c   # char(Maps to OpenCL char)
          ::= a   # signed char (*Not valid)
          ::= h   # unsigned char (Maps to OpenCL uchar)
          ::= s   # short (Maps to OpenCL short)
          ::= t   # unsigned short (Maps to OpenCL ushort)
          ::= i   # int (Maps to OpenCL int)
          ::= j   # unsigned int (Maps to OpenCL uint)
          ::= l   # long (Maps to OpenCL long)
          ::= m   # unsigned long(Maps to OpenCL ulong)
          ::= x   # long long, __int64(*Not valid)
          ::= y   # unsigned long long, __int64(*Not valid)
          ::= n   # __int128 (*Not valid)
          ::= o   # unsigned __int128(*Not valid)
          ::= f   # float (Maps to OpenCL float)
          ::= d   # double (Maps to OpenCL double)
          ::= e   # long double, __float80(*Not valid)
          ::= g   # __float128 (*Not valid)
          ::= z   # ellipsis (*Valid only for printf*)
          ::= Dd # IEEE 754r decimal floating point (64 bits) (*Not valid)
          ::= De # IEEE 754r decimal floating point (128 bits) (*Not valid)
          ::= Df # IEEE 754r decimal floating point (32 bits) (*Not valid)
          ::= Dh # IEEE 754r half-precision floating point (16 bits) (Maps to OpenCL Half)
          ::= Di # char32_t(*Not valid)
          ::= Ds # char16_t(*Not valid)
          ::= Da # auto (in dependent new-expressions)
          ::= Dn # std::nullptr_t (i.e., decltype(nullptr))
          ::= P<builtin-type> # A pointer to private address space.
          ::= PU3ASN<builtin-type> # A pointer to address space 'N' (non-private).
                                   # Only values of 1, 2 and 3 are valid.
          ::= DvN_<builtin-type> # An OpenCL vector of length 'N' of the specified type.
                                   # Only values of 2, 3, 4, 8 and 16 are valid.
          ::= 11ocl_image1d # A 1d image type
          ::= 16ocl_image1darray # A 1d image array type
          ::= 17ocl_image1dbuffer # A 1d image buffer type
          ::= 11ocl_image2d # A 2d image type
          ::= 16ocl_image2darray # A 2d image array type
          ::= 11ocl_image3d # A 3d image type
          ::= 15ocl_image2dmsaa
          ::= 20ocl_image2darraymsaa
          ::= 20ocl_image2dmsaadepth
          ::= 25ocl_image2darraymsaadepth
          ::= 16ocl_image2ddepth
          ::= 21ocl_image2darraydepth
          ::= 9ocl_event # A event type
          ::= 11ocl_sampler # A sampler type
          ::= u <source-name> # vendor extended type
```

SPIR also uses the CV-qualifier list as follows. All CV-qualifiers are order-insensitive.

```
<CV-qualifiers> ::= [r] [V] [K]   # restrict (C99), volatile, const
```

```
# These are order-insensitive.
```

Note: By default, objects reside in the `private` address space (number 0). No address space qualification is used to indicate the private address space.

References

[1] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, SGI, and others. Itanium C++ ABI. http://mentorembedded.github.com/cxx-abi/abi.html.

[2] Khronos OpenCL Working Group. The OpenCL Specification, version 1.2. http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf, November 2012.

[3] LLVM Team. LLVM Bitcode File Format. http://www.llvm.org/releases/3.2/docs/BitCodeFormat.html, 2012. Version 3.2.

[4] LLVM Team. LLVM Language Reference Manual. http://www.llvm.org/releases/3.2/docs/LangRef.html, 2012. Version 3.2.