moviCompile

Movidius Compiler

00.90.0

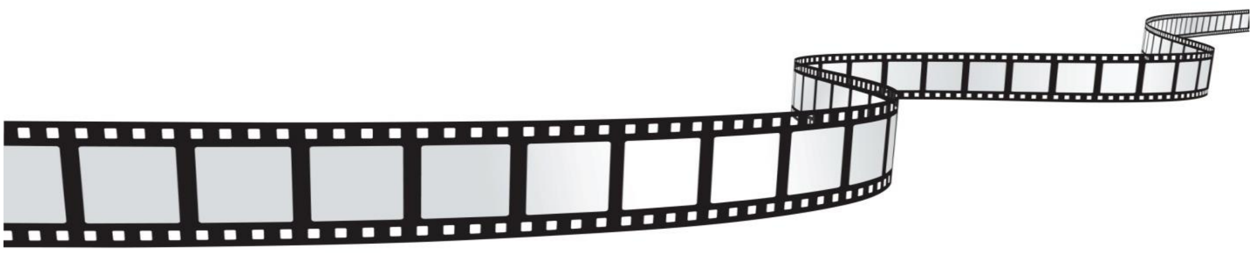

Table of Contents

1 Introduction

1.1 Tool Overview

`moviCompile` is Movidius' compiler for the C and C++ programming languages[1].

`moviCompile` is based on `CLang` and the LLVM v6.0 Open Source compiler infrastructure.

`moviCompile` is the default compiler used to compile code for Movidius' `SHAVE` cores.

The term MDK where referenced by this document, refers to the Movidius Software Development Kit. Generally, the programmer will do all their development for the Myriad platform using the MDK, which in turn uses tools such as `moviCompile`.

1.2 Other Relevant Documents

Movidius documentation:

o moviToolsOverview.pdf	overview of all tools, conventions used, directory structure, etc.

Useful external Links:

o LLVM resources:	http://www.llvm.org

1.3 Instruction Set Architecture (ISA) versus Chip

The Myriad2 architecture is now in its 3rd revision, also known as MyriadX. Each architecture revision comes in one or more chip variants, and each architecture has differences in the `SHAVE` ISA (Instruction Set Architecture), some of which are trivial and others are extensive. However, all of the chips for a particular architecture revision share an identical `SHAVE` ISA.

When generating code for the `SHAVE` processors, the compiler `moviCompile` is only concerned with the ISA and not the specific variant of the chip.

The `SHAVE` ISAs and corresponding chips supported by `moviCompile` are:

1. `SHAVE` ISA version 2.1 is used by the following chip {deprecated}:
   - MA2100 : 1Gb DRAM
2. `SHAVE` ISA version 2.2 is used by the following chips:
   - MA2150 : 1Gb DRAM
   - MA2155 : 1Gb DRAM, with encrypted boot
   - MA2450 : 4Gb DRAM
   - MA2455 : 4Gb DRAM, with encrypted boot
3. `SHAVE` ISA version 2.3 is used by the following chips:
   - MA2080 : No internal DRAM
   - MA2085 : No internal DRAM, with encrypted boot
   - MA2480 : 4Gb DRAM
   - MA2485 : 4Gb DRAM, with encrypted boot

For the purpose of instruction selection, `moviCompile` only needs to know which `SHAVE` ISA is required, and for this the Myriad2 architecture revision is relevant. However, a programmer implementing a program for a

[1] **NOTE:** Although other implementations of Clang and LLVM support additional file types, unless explicitly stated in this document these are not supported by `moviCompile` including LLVM-IR '`.bc`' and '`.ll`' files.

particular Myriad2 chip may need to direct their program compilation according to the actual chip configuration they are using. For this reason `moviCompile` provides a comprehensive set of values for the '`–mcpu=`' option that will introduce appropriate predefined macros for each chip variant to direct conditionally compiled programs; see Table 1 for more details about this option.

1.4 What's New in This Release

1.4.1 The Compiler

1.4.1.1 New pass to eliminate identity copies

Due to the interplay between various optimization passes, at times we end up with redundant identity copy instructions in the assembly output. This release of `moviCompile` adds a new llvm optimization pass to eliminate such copies.

1.4.1.2 Better handling of fp16 and fp64 library functions

It was observed in **B-29023** that some `fp64` math library functions were giving incorrect results. This release of `moviCompile` fixes this issue and also provides a better mapping of `fp64` and `fp16` math operations to the optimized library implementations

1.4.1.3 Discontinued command-line options

As notified in the previous release, the options '`–faligned-allocation`' and '`–fsized-deallocation`' are not supported by `moviCompile` and have been deleted in this release.

1.4.1.4 Many bug fixes

Since the previous Scheduled Release of `moviCompile`, we have addressed many bugs and feature requests. The following lists a short summary of important issues which have been addressed by this full release since the previous full release:

B-29826	Wrong code generation with '`–mstack-overflow-instrumentation`' flag
B-29662	'`vsnprintf`' does not respect the size of the buffer provided
B-29755	CMU.CMVV.I16 instructions generated for comparison of v2i16 values in IRF registers
B-29510	moviCompile interleaving instructions at –O0
B-29023	Incorrect handling of some fp64 math functions
F-28957	Supplement the ISO C math library functions with fp16 variants

Each number has one of the following prefixes:

B	This refers to a bug that has been fixed in this release
F	A feature that was requested and is provided with this release

2 Invocation

This section describes how to invoke `moviCompile`. The following section may also be useful:

- Section 11.1 which describes Simple Usage

2.1 `moviCompile` Command-Line Options

The `moviCompile` tool can be invoked from the command line using the following format:

```
moviCompile [<options>] <fileName> [<fileName>, ...]
```

`moviCompile` generates partially linked ELF files (‘`a.mvlib`’) by default, internally invoking `moviAsm` to convert the internally generated assembly code into ELF format object code, and `sparc-myriad-rtems-ld` to perform the 1st phase linking of the `SHAVE` sources. This ‘`a.mvlib`’ file uses the ‘`_EP_start`’ runtime model (see Section 6.1.5.2).

To compile a simple C file, the user has to enter, for example:

```
moviCompile -c -O1 fileName.c      # Creates 'fileName.o'
```

Or alternatively:

```
moviCompile -S -O1 fileName.c      # Creates 'fileName.s'
moviAsm fileName.s -o:fileName.o   # Creates 'fileName.o'
```

The list of `clang` and `llvm` *<options>* is very large and complex, and has been deliberately simplified in `moviCompile` to the set of options specifically supported by `moviCompile`. The list of supported *<options>* can be obtained by passing `-help` to `moviCompile` thus:

```
moviCompile -help
```

2.1.1 Supported Options

A simplified list of supported <*options*> is presented in the table below:

Switch	**Description**
`-c`	Compile-only – will create a corresponding "`.o`" object file for each source file
`-dD`	Used with ‘`-E`’ this will write the names and values for all the pre-defined macros in addition to the normal output
`-dI`	Used with ‘`-E`’ this will write the ‘`#include`’ directives in addition to the normal output
`-dM`	Used with ‘`-E`’ this will write the names and values for all the pre-defined macros instead of the normal output

Switch	Description

`-D<macro-name>`
`-D<macro-name>=<value>`
	Defines the specified macro, equivalent to '`#define`' in the source code.
	
	The alternative syntax also specifies a value for the macro being defined

`-E`	Emit preprocessed code to the standard output stream

`-f[no-]align-functions`
`-falign-functions=<value>`
	Configures the alignment of all functions to be on the byte boundary specified by *`<value>`* which must be zero or a power of two. The default is to align functions to a 16-byte boundary.
	
	The option:
	    `-fno-align-functions`
	is exactly equivalent to:
	    `-falign-functions=1`
	
	The options (default):
	    `-falign-functions`
	    `-falign-functions=0`
	are exactly equivalent to:
	    `-falign-functions=16`

`-f[no-]align-labels`	Enable or disable the alignment of all branch targets in the generated code. This is disabled by default. When enabled, all branch targets are aligned to a 16-byte boundary if possible (it is not always possible for `moviAsm` to pad instructions to align to the desired 16-byte boundary)
`-f[no-]color-diagnostics`	Enable or disable colorization of diagnostic messages. The default is to colorize when writing to the console
`-f[no-]data-sections`	Place each data object in its own ELF section. The default for `moviCompile` is to place each data object in its own unique section
`-fdiagnostics-absolute-path`	When emitting diagnostics, use the absolute path to the file referred to in the diagnostic. The default is to use the path as stated in the source file, which may be relative
`-f[no-]fast-math`	Enable or disable a set of optimizations that can reduce the overhead of performing math operations involving the ISO C Standard `<math.h>` functions. This is at the expense of strict IEEE compliance. The default is disabled

Switch	Description
`-f[no-]function-sections`	Place each function in its own ELF section. The default for `moviCompile` is to place each function in its own unique section
`-f[no-]inline-functions`	Enable or disable inlining of functions. This defaults to enabled
`-f[no-]inline-hint-functions`	Enable or disable inlining of functions which are defined as inline. This defaults to enabled
`-f[no-]instrument-functions` `-f[no-]instrument-functions-after-inlining`	Enable or disable the automatic insertion of instrumentation at the entry and exit of each function for use with `moviProf`. The default is not to instrument functions
`-f[no-]math-errno`	Instructs the compiler to assume or not to assume that the ISO C math functions may modify '`errno`'. The default for this is '`-fno-math-errno`'. The `SHAVE` implementation of the ISO C math functions in '`mlibm.a`' do not alter '`errno`'
`-f[no-]peephole`	Enable or disable the peephole optimizers. This defaults to enabled for optimization levels '`-O1`' and above
`-frewrite-includes`	When used in conjunction with the '`-E`' option, this will instruct the compiler to also output the un-preprocessed content of each included file into the preprocessed output
`-f[no-]rtti`	Enable or disable compiler support for Run-Time Type Identification (RTTI) in C++. The default for `moviCompile` is to disable RTTI support
`-f[no-]show-column`	Enable or disable the column number when emitting diagnostics. The default is to show the column number
`-fshow-overloads=<`*`option`*`>`	This is used to manage the amount of information produced when emitting diagnostics for C++ template resolution. The values for <*`option`*> are '`all`' and '`best`'. The default is '`all`'
`-f[no-]show-source-location`	Enable or disable the inclusion of the source location when emitting template related diagnostics. The default is to show the source location
`-f[no-]slp-vectorize`	Enable or disable the SLP vectorization optimization. This defaults to disabled for all optimization levels

Switch	Description
`-f[no-]uac-reduction-pass`	Enable or disable the UAC reduction optimization pass. This defaults to disabled for all optimization levels other than '`-O2`' and '`-O3`'
`-f[no-]unroll-loops`	Enable or disable the loop-unrolling optimizations. This defaults to disabled for all optimization levels other than '`-O2`' and '`-O3`'
`-f[no-]vectorize`	Enable or disable the vectorization optimizations. This defaults to disabled for all optimization levels other than '`-O2`' and '`-O3`'
`-f[no-]verbose-asm`	Enable the emission of verbose annotations to the generated assembly code. This defaults to disabled
`-f[no-]wrapv`	Treat signed integer overflows as two's complement. This defaults to disabled
`-g`	Generate source level debug information
`-gline-tables-only`	Generate only line number source level debug information

`-gdwarf-2`
`-gdwarf-3`
`-gdwarf-4`
	When used with '`-g`' this will instruct the compiler to emit source level debug information for the specified version of the Dwarf standard. The default is Dwarf v4

`-help`	Display information about the supported command line options for `moviCompile`
`-H`	Show the header includes and nesting depth
`-I<path>`	Add `<path>` to the set of include directories to be searched

`-MD`
 	
`-M`
	Write a dependency file including user and system headers. This file has the same name as the source file, but with the extension replaced with '`d`'.
	
	As '`-MD`', but write to the standard output stream instead of to a file

`-MDD`
 	
`-MM`
	Write a dependency file including only user headers. This file has the same name as the source file, but with the extension replaced with '`d`'.
	
	As '`-MDD`', but write to the standard output stream instead of to a file

Switch	Description
`-MF <path>`	Used with '`-MD`', '`-M`', '`-MDD`' and '`-MM`' to name the *`<path>`* into which the dependency information should be written
`-MG`	Used with '`-MD`' or '`-MDD`' to add missing headers to the dependency file

`-MP`
 	
`-MQ <path>`
	Creates a '`.PHONY`' target for each dependency other than the '`main`' file.
	
	Identifies the '`main`' file

`-mcpu=<cpu>`	Generates code for the processor specified by *`<cpu>`*. This may be one of:
	`myriad2.1   {deprecated}`
	`  ma2100    {deprecated}`
	`myriad2.2`
	`  ma2150`
	`  ma2155`
	`  ma2450`
	`  ma2455`
	`  ma2x5x`
	`myriad2.3`
	`  ma2080`
	`  ma2085`
	`  ma2480`
	`  ma2485`
	`  ma2x8x`
	
	If unspecified the value defaults to `myriad2.2`. See also section 7.2.1

`-m[no-]enable-fp64-formatting`	When compiling direct to an ELF file (e.g. '`a.mvlib`'), this option instructs the 64-bit Floating-Point formatted output support to be included. The default is not to enable formatted 64-bit Floating-Point output
`-m[no-]floating-point-acc-mac-pipelining`	Enable or disable the use of the `ACC` and `MAC` instruction pipelining for Floating-Point operations. This is disabled by default, unless the optimization level is '`-O2`' or '`-O3`', and '`-ffast-math`' has also been selected
`-m[no-]insert-NaN-checks`	Enable or disable the insertion of explicit NaN checks when generating code for floating-point operations. This is enabled by default unless '`-ffast-math`' is selected

Switch	Description
`-m[no-]integer-acc-mac-pipelining`	Enable or disable the use of the `ACC` and `MAC` instruction pipelining for integer operations. This is enabled by default for optimization levels '`-O2`' and '`-O3`'
`-mlibc`	Use the full ISO C library pair '`mlibc.a`' and '`mlibc_lgpl.a`' when linking
`-mlibc_lite`	Use the Lite ISO C library pair '`mlibc_lite.a`' and '`mlibc_lite_lgpl.a`' when linking. This is the default

`-mlsu-load-policy=<`*`policy-type`*`>`	Specify which LSU should be used for memory loads, where the <*policy-type*> is one of:
	
	`prefer-lsu0`
	`prefer-lsu1`
	`use-only-lsu0`
	`use-only-lsu1`
	
	The default LSU policy for loads is '`prefer-lsu0`'

`-mlsu-store-policy=<`*`policy-type`*`>`	Specify which LSU should be used for memory stores, where the <*policy-type*> is one of:
	
	`prefer-lsu0`
	`prefer-lsu1`
	`use-only-lsu0`
	`use-only-lsu1`
	
	The default LSU policy for stores is '`prefer-lsu1`'

`-mlsu-volatile-policy=<`*`policy-type`*`>`	Specify which LSU should be used for all '`volatile`' memory accesses, where the <*policy-type*> is one of:
	
	`use-only-lsu0`
	`use-only-lsu1`
	
	The default LSU policy for '`volatile`' memory accesses is '`use-only-lsu1`'

`-m[no-]partial-unroll-loops`	Enable or disable the partial unroll loops optimization. This defaults to enabled at '`-O3`' and disabled at all other optimization levels
`-m[no-]replace-jmp-with-bra-peephole`	Enables or disables the peephole optimization that replaces the `LDIL/LDIH/JMP` instruction group with the PC relative `BRA` instruction. This defaults to enabled

Switch	Description
-m[no-]stack-overflow-instrumentation	Enable or disable the insertion of instrumentation to detect if the amount of stack requested by a function would exceed the amount of stack available. This is disabled by default
-m[no-]stack-usage-instrumentation	Enable or disable the insertion of instrumentation to track the maximum amount of stack used. This is disabled by default
-o *<path>*	Specify the name of the output file
-O[<*n*>]	Optimization level; where <*n*> is 0, 1, 2, 3 or 4. '-O4' is present for compatibility with GCC, and is equivalent to '-O2'. If '-O' is specified on its own, it is equivalent to '-O2'
-P	Used with '-E' to disable the emission of line-number information
-pg	Enable mcount function profile instrumentation for use with moviProf
-save-temps -save-temps=<*option*>	When moviCompile is used with the '-c' option, intermediate files are created and assembly files passed to moviAsm. These are normally deleted automatically, but using this option instructs moviCompile to retain the intermediate file(s). By default, when '-save-temps' is selected, it will save the intermediate files in the same directory as the source file; but the alternative form allows the programmer some additional control, where <*option*> is one of: cwd This will save the intermediate files to the current-working-directory. obj This will save the intermediate files to the same directory as the output file is located

Switch	Description

-std=<*standard*>	Tells moviCompile to compile the source file in accordance with the rules for the selected ISO C or ISO C++ language Standard.
	Supported options are:
	`c89    (ANSI C89)`
	`c90    (ISO C90, same as C89)`
	`c99    (ISO C99)`
	`c11    (ISO C11)`
	`gnu89`
	`gnu90`
	`gnu99  (default for C)`
	`gnu11`
	`c++98  (ISO C++98)`
	`c++11  (ISO C++11)`
	`c++14  (ISO C++14)`
	`c++17  (ISO C++17)`
	`gnu++98  (default for C++)`
	`gnu++11`
	`gnu++14`
	`gnu++17`
	If unspecified the value defaults to C99 if the source is identified as a C file, or C++98 if the source is identified as a C++ file.
	The 'gnu' variants are the same as the ISO equivalent, but with GCC compatible extensions enabled

-S	Create assembly file only – will create a corresponding '.s' assembly file for each source file
-U<*macro-name*>	Undefines the specified macro, equivalent to '#undef' in the source code
-v	Enable verbose mode which will display additional internal information when moviCompile is invoked
--version	Show summary version information and exit
-Wa,<*options*>	Specifies a comma-separated list of <*options*> to be passed to the assembler moviAsm
-Wall	Enables all normal warnings
-Werror	Makes all warning become errors
-Wextra	Enables additional pedantic warnings not enabled with '-Wall'
-Wl,<*options*>	Specifies a comma-separated list of <*options*> to be passed to the 1st Phase linker sparc-myriad-rtems-ld

Switch	Description
`-Xassembler <option>`	Passes a single `<option>` to the assembler `moviAsm`
`-Xlinker <option>`	Passes a single `<option>` to the 1st Phase linker `sparc-myriad-rtems-ld`
`@<filename>`	This will instruct `moviCompile` to read options from the file named `<filename>` before continuing to process additional command-line options

Table 1. `moviCompile` command line options

The full set of supported options will also be displayed by the following command:

```
moviCompile -help
```

2.1.2 Unsupported Options

Any other options that may be provided by other `clang` and `llvm` implementations or which are discovered in another fashion and which are not listed in Table 1 above or by using the '`-help`' option, are considered internal to the `moviCompile` implementation. These are specifically **not** supported and are subject to change in future releases without notice or explanation.

2.1.3 File Types

The default behavior for `moviCompile` is to produce an ELF object file for each of the input source files. This is achieved by invoking the assembler `moviAsm` after all other translations have completed. By default, `moviCompile` accepts the following input file-extensions:

`.c` The file is expected to contain ISO C source, and is compiled by default for the C99 Standard with GCC extensions enabled. The compiler first compiles the source to `SHAVE` assembly code, and then passes this through `moviAsm` to create the object code file

`.cc`
`.cpp`
`.cxx` The file is expected to contain ISO C++ source, and is compiled by default for the C++98 Standard with GCC extensions enabled. The compiler first compiles the source to `SHAVE` assembly code, and then passes this through `moviAsm` to create the object code file

`.o` The file is expected to contain the SHAVE object code for a previously compiled or assembled source

`.s` The file is expected to contain `SHAVE` assembly source, and is passed directly to `moviAsm`

`.S` The file is expected to contain `SHAVE` assembly source, but with C pre-processor directives present. In this case `moviCompile` will first pass the input source through the inbuilt C pre-processor, and the resulting pre-processed assembly is passed on to `moviAsm`

NOTE: Files with any extension not listed above (including '`.bc`' and '`.ll`') are specifically **not** supported.

While `moviCompile` will internally create an assembly file, this file is temporary and will be deleted automatically upon completion, and only the output file of the same name as the input file with the extension '`.o`' is kept. Sometimes however, the programmer may want to retain the intermediate assembly file for inspection or curiosity, and in this case they can use the option '`-save-temps`' which will retain both the intended output object file, plus the intermediate assembly file with the same name as the input file, but the extension '`.s`'.

For example:

```
moviCompile -c file1.c file2.cpp file3.s file4.S
```

will result in 4 output files[2]:

- '`file1.c`' is compiled from C to assembly by `moviCompile`, and then passed to `moviAsm` to create the final output '`file1.o`'
- '`file2.cpp`' is compiled from C++ to assembly by `moviCompile`, and then passed to `moviAsm` to create the final output '`file2.o`'
- '`file3.s`' is passed unchanged to `moviAsm` which creates the final output '`file3.o`'
- '`file4.S`' is pre-processed by `moviCompile` using the in-built C pre-processor, and then passed to `moviAsm` to create the final output '`file4.o`'

but:

```
moviCompile -save-temps -c file1.c file2.cpp file3.s file4.S
```

will result in 3 additional output files:

- '`file1.s`' which is the intermediate assembly file created by `moviCompile` from the input C source
- '`file2.s`' which is the intermediate assembly file created by `moviCompile` from the input C++ source
- '`file4.s`' which is the intermediate pre-processed assembly file created by `moviCompile` from the input assembly source[3]

Since '`file3.s`' is passed directly to `moviAsm`, there is no intermediate file produced.

Unlike a traditional compiler for the host system, `moviCompile` is a cross-compiler for `SHAVE` and is unable to link multiple files to create an executable image.

[2] Not strictly true, it will also save other intermediate files such as the pre-processed files ('`.i`' and '`.ii`') and the LLVM Byte-Code files ('`.bc`')

[3] **NOTE:** On Windows the file system is "case preserving", but not "case sensitive", so the automatically generated file named '`file4.s`' will overwrite the existing file named '`file4.S`'. In this case, use '`moviCompile -S file4.S -o file4.asm`' or some other unique name

3 C Language Support

3.1 Supported Language set

`moviCompile` supports ISO C90 (ANSI C89), ISO C99 and ISO C11 with the following limitations:

- The data types '`float`' and '`double`' are both 32-bit IEEE Single Precision floating-point
  - Use '`long double`' for emulated 64-bit IEEE Double-Precision floating-point
- The ISO C Standard library functionality provided by '`<setjmp.h>`' is not supported
- `moviCompile` does not provide support for Unicode or wide-characters
- Library support for C11 may not be complete
- The GCC extension allowing the address of a label to be taken is not supported by `moviCompile`

This is not a complete list; if you find an unsupported feature missing from this list, please report this on `movidius.org`.

The default for compiling C programs is C99 with GCC Extensions enabled.

4 C++ Language Support

4.1 Supported Language set

`moviCompile` supports ISO C++98, ISO C++11 and ISO C++14 with the following limitations[4]:

Not all functionality is supported on the `SHAVE` processor, specifically `moviCompile` does not support the following features of the C++ Language:

- **Clocks** - `SHAVE` has no real-time clock or suitable counters
- **Exception Handling** - supporting EH has a very high space and performance penalty, even for programs not using EH, making it unsuitable for the `SHAVE` embedded processor
- **Locales** - supporting locales has a very high space and performance penalty, even for programs not using locales, making it unsuitable for the `SHAVE` embedded processor
  - **'`ctype<char>::classic_table()`'** - this table is related to the implementation of **Locales**, and is consequently not supported
- **Threads** - `SHAVE` does not support multi-tasking

Support is provided for RTTI (Run-Time Type Identification), but this is disabled by default. Enabling RTTI will increase the size of the generated program even when the program does not use RTTI, and may also degrade its performance. The C++ library has been built with RTTI enabled so that programs that do require the use of RTTI may do so when RTTI is explicitly enabled by the programmer.

This is not a complete list; if you find an unsupported feature missing from this list, please report this on `movidius.org`.

The default for compiling C++ programs is C++98 with GCC Extensions enabled.

[4] It also supports '`-std=c++03`'. This is not a formal Standard, but a Technical Corrigenda to ISO C98.

5 OpenCL Language Support

OpenCL is not supported by `moviCompile`. The option may be selected as it is a property of `CLang` from which `moviCompile` is derived, but its behavior is unspecified and unsupported.

Instead `moviCompile` provides a Vector Utility Library (Section 6.3) which provides support for explicit vectorization similar to that offered by the OpenCL C language bindings.

6 Library Support

Like many compilers, `moviCompile` provides a set of Standard and custom libraries and headers.

The `common/moviCompile/include/` directory contains the header files which provide the declarations for the functions, types, constants, macros and objects for each of the provided libraries.

The `common/moviCompile/include/c++/` directory contains the header files for the C++ library.

The directory `common/moviCompile/lib/ma2100/` contains the provided libraries for the `SHAVE` ISA v2.1 instruction set (‘`ma2100`’ chip) – MA2100 is no longer formally supported.

The directory `common/moviCompile/lib/ma2x5x/` contains the provided libraries for the `SHAVE` ISA v2.2 instruction set (‘`ma2150`’, ‘`ma2155`’, ‘`ma2450`’ and ‘`ma2455`’ chips).

The directory `common/moviCompile/lib/ma2x8x/` contains the provided libraries for the `SHAVE` ISA v2.3 instruction set (‘`ma2080`’, ‘`ma2085`,’ ‘`ma2480`’ and ‘`ma2485`’ chips).

6.1 ISO C Standard Libraries

`moviCompile` provides a C library implementation which is compliant with the ISO C90 (ANSI C89), ISO C99 and ISO C11 Standards. The default for compiling a C program is to select the ISO C99 Standard with GCC extensions, but the programmer can choose any of the other Standards using the option:

```
-std=c90
```

or:

```
-std=c11
```

To allow the programmer to decide between solutions that have different objectives with respect to performance versus accuracy and completeness, `moviCompile` offers two implementations of the C libraries. The two libraries are binary compatible, and can be chosen when the program is being linked without the need to recompile the sources for the program.

6.1.1 `mlibc_lite`

This is the minimalist “Lite” version of the non-math components of the ISO C Standard library. ‘`mlibc_lite`’ is not a comprehensive implementation of the ISO C Standard library and provides only a subset of commonly used C functions. However, this subset has been implemented to provide a smaller and faster implementation than the more complete and comprehensive Newlib implementation (section 6.1.2), and is generally adequate for most `SHAVE` programs.

The library actually consists of a pair of library archives:

```
mlibc_lite.a
mlibc_lite_lgpl.a
```

The library archive named ‘`mlibc_lite.a`’ contains object code from sources that are proprietary to Movidius and `moviCompile`, while the library archive named ‘`mlibc_lite_lgpl.a`’ contains a small number of supplementary object code files from sources that are distributed under the LGPL license for the OpenSource project Newlib v2.5.0-20170922.

6.1.1.1 Limitations – ‘`printf`’ and ‘`mlibc_lite_lgpl.a`’

The implementation of the ‘`*printf*`’ family of formatting functions is not complete, and in particular the formatting of Floating-Point numbers has some significant limitations:

- Scientific notation is not supported, so the ‘`%e`’ and ‘`%E`’ options will not work as expected, which in turn means that ‘`%g`’ and ‘`%G`’ will never select scientific notation.
- The precision of formatting a Floating-Point number is limited to 18 characters.

- The value following the decimal point in a Floating-Point number is accurate to 18 characters after which ‘0’ will be used for all subsequent characters in the requested precision.

The full ISO C library `mlibc.a` does not have these limitations (section 6.1.2).

This group of functions does support the format specifier ‘`%Lf`’ for printing 64-bit Floating-Point numbers, however, this too has limitations.

The 64-bit Floating-Point library is emulated, as there is no native support for FP64 on `SHAVE`. The emulation library incurs a significant performance penalty as would be expected; but less obvious is that there is a significant space penalty.

This space penalty is not normally an issue, because the build process does not link unreferenced functions into the application unless it is explicitly referenced in the program – for example, the function:

```
long double sinl(long double)
```

will not be part of the program image if the function ‘`sinl`’ is not referenced by the program.

Unfortunately, because the ‘`*printf*`’ family of functions has to support the ‘`%Lf`’ format specifier, even if no use of ‘`long double`’ occurs in the program, it would consequently cause a large and unnecessary space penalty for programs not using 64-bit Floating-Point, but which use any of these formatted output functions.

To mitigate against this penalty, the implementation of these functions in ‘`mlibc_lite.a`’ use a “Weak Extern” reference to a function named:

```
__ldbl2stri
```

The definition of this function will not be automatically included when the program is linked, and if the ‘`%Lf`’ specifier is used, then the expected 64-bit floating-point value will be replaced with the sub-string:

```
<<!NOFP64!>>
```

However, `moviCompile` does provide the implementation of this function as a discrete object file which is located in the library directory corresponding to the intended `SHAVE` ISA:

```
common/moviCompile/lib/ma2100/ldbl2stri.o
common/moviCompile/lib/ma2x5x/ldbl2stri.o
common/moviCompile/lib/ma2x8x/ldbl2stri.o
```

To use this functionality, the program must link with this object file explicitly provided on the linker command-line.

6.1.2 mlibc

The library ‘`mlibc`’ fully supports all of the supported ISO C Standards (except for the math functions) without any special action by the programmer. This library is mostly derived from the OpenSource project Newlib v2.5.0-20170922.

The library actually consists of a pair of library archives:

```
mlibc.a
mlibc_lgpl.a
```

The library archive named ‘`mlibc.a`’ contains object code from sources that are proprietary to Movidius and `moviCompile`, while the library archive named ‘`mlibc_lgpl.a`’ contains the object code for the OpenSource project Newlib v2.5.0-20170922 which are distributed under the LGPL license.

6.1.2.1 Limitations – ‘`printf`’ and ‘`mlibc.a`’

The implementation of the ‘`*printf*`’ family of formatting functions provided with ‘`mlibc.a`’ does not support the printing of 64-bit Floating-Point values.

6.1.2.2 Why Two Versions?

Earlier versions of the `moviCompile` libraries were neither complete nor ISO C compliant. However, they were far more lightweight than a full ISO C compliant implementation, and for many applications this provided a “good enough” implementation.

The more complete ISO Compliant version of ‘`mlibc.a`’ on the other-hand trades performance and size for accuracy and completeness.

By providing both sets of libraries, the programmer can decide which objective is the more important for their particular application. For some applications, speed or program size will be the most important trait, while for others accuracy and completeness will be essential. Since this is a link-time decision, the programmer can select between one or the other of this library pair without being required to recompile their code.

6.1.3 The ISO C Math Library

The library ‘`mlibm.a`’ has been optimized specifically for `SHAVE` to provide enhanced performance and precision. It contains a full implementation of the ISO C Math functions for ‘`float`’ and for ‘`double`’ – both of which use 32-bit IEEE Single Precision Floating-Point – and for ‘`long double`’ which uses 64-bit IEEE Double Precision Floating-Point. It is also supplemented by an equivalent set of functions for the ‘`half`’ or ‘`__fp16`’ 16-bit IEEE Half Precision Floating-Point type See section 13 for detailed performance and precision information.

Full support for 64-bit floating-point is provided by emulation using the data-type ‘`long double`’. The full set of ISO C11 functions for ‘`long double`’ are provided by ‘`mlibm.a`’ in addition to the 32-bit and 16-bit versions. The ‘`long double`’ versions of these functions have the same name as the ‘`double`’ versions, but suffixed by the letter ‘`l`’.

6.1.3.1 The 16-bit Floating-Point Library

In addition to the `SHAVE` optimized implementation for the 32-bit Floating-Point math library, `moviCompile` supplements the ISO math library ‘`mlibm.a`’ with full support for 16-bit Floating-Point. Access to this library is exactly the same as for the normal ISO C functions using the Standard header ‘`<math.h>`’.

Since there are no ISO C names for these, we have given these functions the same name as their ‘`double`’ equivalents in the ISO C11 math library, but prefixed with ‘`__`’[5] and suffixed with ‘`s`’. The `moviCompile` math libraries provide this additional variant. For example, the function ‘`sin`’ which has the following three ISO forms:

```
float       sinf(float);        // 32-bit FP
double      sin (double);       // 32-bit FP
long double sinl(long double); // 64-bit FP
```

‘`mlibm.a`’ has the following four forms:

```
__fp16    __sins(__fp16);       // 16-bit FP
float       sinf(float);        // 32-bit FP
double      sin (double);       // 32-bit FP
long double sinl(long double); // 64-bit FP
```

NOTE: The 16-bit floating-point data type is always available using the keyword ‘`__fp16`’.
NOTE: `moviCompile` also supports the extended type ‘`short float`’ as a synonym for ‘`__fp16`’.

[5] Use of the double-underscore prefix is reserved to the implementation by ISO C, in this case `moviCompile`

NOTE: The commonly used name ‘`half`’ is implemented as a ‘`typedef`’ which is defined when the header file ‘`<moviVectorUtils.h>`’ is included.

Additionally, if the pre-processing symbol ‘`__STRICT_ANSI__`’ is not defined, these functions are also available using the same name but without the ‘`__`’ prefix, for example ‘`sqrts`’ instead of ‘`__sqrts`’.

NOTE: ‘`__STRICT_ANSI__`’ is not defined by default, or whenever any of the ‘`-std=gnu*`’ options are used.

6.1.3.2 Rounding Modes

The ISO C Standard header ‘`<fenv.h>`’ provides the interface for setting the rounding modes for expressions involving only floating-point values.

However, the `SHAVE` processor also supports hardware rounding modes for conversions from floating-point to integer, and ‘`<fenv.h>`’ for `moviCompile` extends this interface for these additional `SHAVE` rounding-modes with the following two functions:

`int __getfptointround(void)`

Which will return the FP-to-INT rounding mode currently selected. The return values are exactly as for the usual ISO C FP-to-FP rounding modes, that is, one of the following values:

```
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
FE_DOWNWARD
```

`int __setfptointround(int round)`

This function allows the programmer to set the FP-to-INT rounding mode to one of the four values above. This will return the value zero if setting the rounding-mode was successful, and non-zero otherwise.

If the macro ‘`__STRICT_ANSI__`’ is not defined, these names are supplemented by macros which use the same names, but without the leading double-underscore, thus:

```
getfptointround
setfptointround
```

6.1.4 The System-Call Stubs

In addition to the ISO C functions, the libraries ‘`mlibc.a`‘ and ‘`mlibc_lite.a`’ contain a set of “System Call” stub functions.

Tthe ‘`mlibc_lgpl.a`’ library is built from the OpenSource project Newlib which is a very portable implementation of the ISO C library, but it is not possible to make a library such as the C library 100% portable because eventually it has to engage with the actual system itself for which there is no portable definition.

This layer is often referred to as the “System Calls” or `syscalls` layer, and consists of a small set of discrete functions that bind the portable implementation of the C library to the final non-portable part that binds to the actual system. On a typical hosted system, Newlib provides this low level binding using the ‘`libgloss`’ sub-library, but it is not feasible for this sub-library to be ported to `SHAVE` in any meaningful way, so instead `SHAVE` provides its own very lightweight system call implementation.

Since no Operating System is running on `SHAVE`, many of these calls have no meaning and are implemented as “dummies”. These dummy functions provide a valid response to the caller, but the valid response generally informs the caller that the requested operation is not available.

For example, there is no meaningful definition for opening a file since there is no file system available to `SHAVE` (by default) so one of the system calls – '`_open`' – simply returns an error state (in '`errno`') that informs the caller that it was not able to open the file.

All of these functions with the exception of '`_write`' are implemented as dummies in `SHAVE`'s '`mlibc.a`' and '`mlibc_lite.a`' libraries, and '`_write`' is implemented to send the requested characters to the UART; this is mostly used for debugging purposes.

However, to allow the system implementer provide their own custom implementations of some or all of these system-call stubs, they can provide their own definition during linking ahead of the reference to the library.

The sources for the stubs used by this library are located in:

```
common/moviCompile/src/syscalls
```

6.1.5 Supplementary Functions

The C libraries are also supplemented with some additional functions that are particular to the programming requirements of the Myriad platform. These supplementary functions are declared in the following `SHAVE` system header:

```
sys/shave_system.h
```

Some of these supplementary functions have two forms:

```
  __<function-name>
_EP_<function-name>
```

The first form is a `SHAVE` callable function, and will return to the `SHAVE` caller in the normal fashion. The second form with the '`_EP_`' prefix is an "Entry-Point" that is invoked from the `Leon`. On completion this form will use the '`BRU.SWIH`' instruction to yield control to the `Leon`.

By providing both forms, the programmer is able to choose whether the associated task is best carried out from within their `SHAVE` code, or invoked by the `Leon` executive program.

6.1.5.1 Heap Management

By default, the heap used by `SHAVE` programs will be set to the address `0x1F000000` and assumes a size of 1Mbyte. This is the `WIN_D` windowed address space, but requires that the `WIN_D` register is initialized appropriately.

However, many programs do not initialize this register, or intend to use it for another purpose. To allow the programmer to specify an alternative heap, the libraries are supplemented with the following two functions:

```
void      __setheap(void* heapAddress, size_t heapSize)
int32_t _EP_setheap(void* heapAddress, size_t heapSize)
```

These two function perform exactly the same task, and that is to set the address and size of the heap to be used by the program. The function '`__setheap`' is a `SHAVE` callable function, and will return to the `SHAVE` caller after initializing the heap information; while the function '`_EP_setheap`' is implemented as an Entry-Point intended to be started from the `Leon`, and which will return control to the `Leon` upon completion.

These functions require that the heap is aligned to an 8-Byte boundary, and that the size is a minimum of 1KByte.

On returning control to the `Leon`, the '`_EP_setheap`' Entry-Point will return '`0`' in IRF '`I18`' if it completed normally, otherwise the value will be as described in section 6.1.5.3 for abnormal termination.

In the event of an error in any of the heap functions, the programmer can provide an error handler that will be invoked in the event of one of the following error conditions occurring (these are declared in '`<sys/shave_system.h>`' in the enumeration '`_HeapErrorType_t`'):

`_HET_setheap_invalid_args`

> If '`__setheap`' or '`_EP_setheap`' is called with invalid arguments (i.e. the address is not 8-Byte aligned, or the minimum size is less than 1KByte), the custom error handler is called with this value.
>
> If no custom error handler is registered, the default action is to accept the invalid arguments and continue, though the heap address is adjusted until it is 8-Byte aligned.
>
> If the custom error handler returns, the default action will take place.

`_HET_malloc_out_of_memory`

> If '`malloc`' is unable to allocate the amount of memory requested, and the programmer has registered an error handler, then the error handler is called with this value.
>
> If no custom error handler is registered, the default action is for '`malloc`' to return `NULL`.
>
> If the custom error handler returns, the default action will take place.

`_HET_free_unallocated_block`

> If '`free`' is called with the address of a valid memory block, but that block has already been freed, and the programmer has registered an error handler, then the custom error handler is called with this value.
>
> If no custom error handler is registered, the default action is to ignore the error and return without making any changes to the heap.
>
> If the custom error handler returns, the default action will take place.

`_HET_free_invalid_block`

> If '`free`' is called with the address of an invalid memory block, and the programmer has registered an error handler, then the custom error handler is called with this value.
>
> If no custom error handler is registered, the default action is to ignore the error and return without making any changes to the heap.
>
> If the custom error handler returns, the default action will take place.

The programmer can register a custom heap error handler[6] of their own choosing by calling either of the following functions:

```
_HeapErrorHandler_t
      __set_heap_error_handler(_HeapErrorHandler_t newErrorHandler)
_HeapErrorHandler_t
    _EP_set_heap_error_handler(_HeapErrorHandler_t newErrorHandler)
```

> In either case, the function will return the previously registered heap error handler in IRF register '`I18`' (the default error handler is `NULL`).
>
> The error handler must have the following type, and must be `SHAVE` callable[7]:

```
    void myHandler(_HeapErrorHandler_t);
```

[6] This can be useful for debugging heap related issues, but be wary of calling functions that may require the heap from within the handler which may result in a recursive call to the error handler.
[7] That is, it cannot be an "Entry-Point".

6.1.5.2 Program Execution, Initialization and Finalization

To support alternative program execution models, the C libraries are supplemented with functions to facilitate writing programs which correctly handle initialization and finalization of the execution context. For more information about program execution models see section 10.

```
int32_t _EP_start(int32_t argc, const char* argv[])
```

This function is only provided as an Entry-Point that should be invoked from the `Leon`. This provides the most straight-forward conventional execution model. Programs that use this model must contain a `SHAVE` callable function called '`main`'. This Entry-Point will ensure that all initialization operations have completed before invoking '`main`', and following return from '`main`' will ensure that all finalization operations are completed before yielding control to the `Leon` using '`BRU.SWIH`'. The parameters '`argc`' and '`argv`' are passed to the programmer's '`main`'.

```
void      __crtinit(void)
int32_t _EP_crtinit(void)
```

These functions provide support for alternative execution models, and should not be used in conjunction with the conventional execution model supported by '`_EP_start`'. These will perform the initialization operations that would usually occur before invoking '`main`' in the conventional execution model, and ensure that the execution context is valid for the remainder of the program's lifetime.

Only one of this pair of functions should be called in the lifetime of a single program, and must be called exactly ONCE prior to the execution of any other tasks associated with the program.

On returning control to the `Leon`, the '`_EP_crtinit`' Entry-Point will return '`0`' in IRF '`I18`' if it completed normally, otherwise the value will be as described in section 6.1.5.3 for abnormal termination.

```
void      __crtfini(int32_t exit_code)
int32_t _EP_crtfini(int32_t exit_code)
```

These functions provide support for the finalization phase for alternative execution models, and should not be used in conjunction with the conventional execution model supported by '`_EP_start`'. These will perform the program finalization operations that would normally be performed by calling '`exit`' or returning from '`main`' in the conventional execution model.

Only one of this pair of functions should be called in the lifetime of a single program, and must be called exactly ONCE following the execution of all other tasks associated with the program.

The function '`__crtfini`' will end with the instruction '`BRU.SWIH`' and yield control to the `Leon`, passing the value '`exit_code`' in IRF '`I18`' This is because after finalization has completed, the execution context is no longer considered valid and allowing it to return to a `SHAVE` C calling function is not meaningful or safe.

On completion, both of these functions return control to the `Leon` using '`BRU.SWIH`' with the value which was passed as it's argument, or alternatively, if '`exit`' or '`_Exit`' are called explicitly, this will be the value the programmer passed to those functions.

NOTE: Calling the function '`exit`' will initiate the finalization of the execution context regardless of whether the program uses the conventional execution model, or an alternative execution model. It is not recommended that the programmer calls '`exit`' in a program that is not using the conventional execution model, and are instead advised to use ISO C99's '`_Exit`' function[8] which will skip the program finalization and yield control immediately to the `Leon`.

NOTE: The function '`__crtinit`' and the Entry-Points '`_EP_start`' and '`_EP_crtini`' explicitly set the `SHAVE` rounding modes to the documented defaults for the chip. This is necessary to ensure that any

[8] The `moviCompile` libraries also provides the common extension '`_exit`'.

program started with these CRT Entry-Points do so with the Floating-Point hardware configured in a well-known state. The precision and correctness of the ISO C math library functions depends on the machine to be in the default state; though the programmer is free to change these using the interface described by '`<fenv.h>`', doing so will change the behavior of the math library.

6.1.5.3 SHAVE System Exit Codes

When a `SHAVE` function is defined with the '`dllexport`' attribute it returns control to the `Leon` using the '`BRU.SWIH`' instruction instead of a normal "return" using the '`BRU.JMP I30`' instruction (section 9.11.1). The '`BRU.SWIH`' instruction has a "tag field" which allows a value to be passed to the `Leon` at the same time as halting the `SHAVE`. For the Myriad2 v1 and v2 architectures, this is a 5-bit field, and although that value has not been traditionally inspected by the Myriad Software Development framework, it has by convention always been given the value '`0x1F`' (all ones in binary).

With the MyriadX (Myriad2 v3) architecture the tag field is widened to 13-bits.

To facilitate the development of more robust program development, `moviCompile` defines a standard set of tags that will allow applications to make better decisions depending on "why" a `SHAVE` Entry-Point ended and returned control to the `Leon`. This way the `Leon` code which invokes the `SHAVE` Entry-Point can inspect whether the `SHAVE` application exited in a normal way, or if it was a result of some abnormal event such as '`abort`' being called.

The first 16 values in the tag field are reserved to `moviCompile` and the Myriad Software Development system, and of these 4 have presently got a formal assigned purpose. These values are defined in the header file:

```
sys/shave_exitcodes.h
```

The remaining 16 values[9] for the Myriad2 v1 and v2 architectures are available for the application developer to use for their own purpose, or the remaining 8,176 values for MyriadX architecture.

The values currently defined for system use are as follows:

`SHAVEExitNormal` – the TAG field will be set to this value when a `SHAVE` Entry-Point exits normally. If the Entry-Point is a user defined function, then the registers representing the return value are set as normal for the `SHAVE` calling convention. If the functions '`exit`' or '`_Exit`' are called, then the register '`I18`' is also set to have the '`int`' value that the programmer passed when calling these functions. Returning from the function '`main`'[10] is equivalent to calling '`exit`' with the value provided in the return statement.

`SHAVEExitAbortCalled` – when the C function '`abort`' is called, the TAG field will be set to this value, and the value in the register '`I18`' will be set to '`-1`'.

`SHAVEExitAssertCalled` – when the C function '`assert`' is called, the TAG field will be set to this value, and the value in the register '`I18`' will be set to '`-2`'.

`SHAVEExitStackOverflow` – when stack-overflow instrumentation is enabled (see section 10.5.3), and the instrumented code detects that the stack usage would exceed the limit specified by register '`I20`' when the most recent '`dllexport`' Entry-Point function was invoked (section 9.11.1.1), then the TAG field will be set to this value, and the value in the register '`I18`' will be set to '`-3`'.

The next 12 values are reserved for future system requirements.

[9] Actually, the value '`0x1F`' is only partially reserved. It is specifically provided for by this header, but only to allow legacy code use a more symbolic form of the usual "Magic Number".
[10] The function '`main`' is special to both C and to C++ and should never be defined or used as an Entry-Point.

_LegacyMDKExit - to allow legacy applications to use the conventional value '0x1F', this exit-code is also provided.

In addition to the formalized exit-code and TAG field values, the following MACRO interface is provided to allow the programmer to relinquish control to the Leon with a custom TAG field value. A custom tag field is any value which is not reserved by the system, i.e. values between 0 and 15, and the value 31 ('0x1F'):

```
shave_sysexit(exitcode)
```

This interface is implemented as a macro because it uses '__builtin_shave_bru_swih_i', and this particular intrinsic requires that the parameter must be a literal integer constant and not a variable. Passing a variable to this intrinsic will result in a compile time error (see also section 8.1.1).

6.2 ISO C++ Standard Library

moviCompile provides a C++ library implementation which is compliant with the ISO C++98, ISO C++11 and ISO C++14 Standards. The default for compiling a C++ program is to select the ISO C++98 Standard with GCC extensions enabled, but the programmer can choose any of the more recent Standards using the option:

```
-std=c++11
-stc=c++14
```

or:

```
-std=c++17
```

Except for the explicitly unsupported features of C++ (see section 4.1), the library 'mlibcxx.a' fully supports all of these Standards without any special action required by the programmer. This library is derived from the LLVM C++ library source LibC++ which can be found on the LLVM website.

Although moviCompile will default to having RTTI (RunTime-Type Identification) disabled, the libraries have been built with full RTTI support enabled and may be used by the programmer in an RTTI safe and compliant way as required.

6.3 Vector Utility Library

This section describes the vector extension support available in moviCompile, along with the support utility library 'mlibVecUtils.a'. The interface for this library is inspired by the C interface bindings for the OpenCL v1.2 specification.

6.3.1 CLang Vector Extensions

The CLang vector extensions are always available by including '<moviVectorUtils.h>', the moviCompile support library for explicit vectorization. The CLang vector extensions follow the GCC vector extensions for the most part, but diverge on some aspects (notably '__builtin_shufflevector'). A full description of the GCC vector extensions can be found at the following URL:

```
http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html
```

And a full description of the CLang vector extensions can be found at:

```
http://clang.llvm.org/docs/LanguageExtensions.html
```

The moviCompile vector extensions follow the CLang extensions with some exceptions and also provides some additional functionality.

6.3.2 Supported Vector Data-Types

6.3.2.1 Native vector data-types

All native `SHAVE` vector types are 128-bits wide. The `SHAVE` instruction set has native support for the following C equivalent vector data types:

- 4 x signed 32-bit signed integer (`long` and `int`)
- 4 x unsigned 32-bit integer (`unsigned long` and `unsigned int`)
- 4 x 32-bit floating-point (`float`)
- 8 x signed 16-bit integer (`short`)
- 8 x unsigned 16-bit integer (`unsigned short`)
- 8 x 16-bit floating-point (`half`, `short float` or `__fp16`)
- 16 x 8-bit signed integer (`signed char`)
- 16 x 8-bit unsigned integer (`unsigned char` and `byte`)

For programmer convenience, these native vector types are represented by the following '`typedef`'s when the '`<moviVectorUtils.h>`' header file is included by the programmer:

Type	Description	Aliases
`long4`	`4 x 32-bit signed integer`	
`ulong4`	`4 x 32-bit unsigned integer`	
`int4`	`4 x 32-bit signed integer`	
`uint4`	`4 x 32-bit unsigned integer`	
`float4`	`4 x 32-bit floating-point`	
`short8`	`8 x 16-bit signed integer`	
`ushort8`	`8 x 16-bit unsigned integer`	
`half8`	`8 x 16-bit floating-point`	
`schar16`	`16 x 8-bit signed integer`	
`char16`	`16 x 8-bit integer`	`See note`
`uchar16`	`16 x 8-bit unsigned integer`	`byte16`

Table 2. Native Vector Data Types

Operations involving the natively supported data types are the most optimal in performance as in many cases there is an exact correspondence between the operation and the `SHAVE` instruction set.

NOTE: The '`char`*N*' data types require special consideration. In the C Programming Language, the actual type of '`char`' is either '`signed char`' or '`unsigned char`', so it is in effect an alias.

In the C++ Programming Language '`char`' is a different type to '`signed char`' and '`unsigned char`', but it may be implemented as either a '`signed char`' or '`unsigned char`'.

The default implementation of '`char`' for `moviCompile` is '`signed char`'.

6.3.2.2 16-bit vector data-types

In addition to the 128-bit native vector types, `moviCompile` also has partial support for 16-bit vector data types. Many but not all operations on 16-bit vector data-types have optimized native implementations, and not all functions are natively supported for 16-bit vector data-types. These types are as follows:

Type	Description	Aliases
`schar2`	`2 x 8-bit signed integer`	
`char2`	`2 x 8-bit integer`	
`uchar2`	`2 x 8-bit unsigned integer`	`byte2`

Table 3. Additional 16-bit Vector Data Types

6.3.2.3 32-bit vector data-types

`moviCompile` also has partial support for 32-bit vector data types. Many but not all operations on 32-bit vector data-types have optimized native implementations, and not all functions are natively supported for 32-bit vector data-types. These types are as follows:

Type	Description	Aliases
short2	2 x 16-bit signed integer	
ushort2	2 x 16-bit unsigned integer	
half2	2 x 16-bit floating-point	
schar4	4 x 8-bit signed integer	
char4	4 x 8-bit integer	
uchar4	4 x 8-bit unsigned integer	byte4

Table 4. Additional 32-bit Vector Data Types

6.3.2.4 64-bit vector data-types

`moviCompile` also has partial support for 64-bit vector data types. Many but not all operations on 64-bit vector data-types have optimized native implementations, and not all functions are natively supported for 64-bit vector data-types. These types are as follows:

Type	Description	Aliases
long2	2 x 32-bit signed integer	
ulong2	2 x 32-bit unsigned integer	
int2	2 x 32-bit signed integer	
uint2	2 x 32-bit unsigned integer	
float2	2 x 32-bit floating-point	
short4	4 x 16-bit signed integer	
ushort4	4 x 16-bit unsigned integer	
half4	4 x 16-bit floating-point	
schar8	8 x 8-bit signed integer	
char8	8 x 8-bit integer	
uchar8	8 x 8-bit unsigned integer	byte8

Table 5. Additional 64-bit Vector Data Types

6.3.2.5 Additional vector data-types

In addition to the natively supported types, and the limited 16-bit, 32-bit and 64-bit vector types; the vector extensions implement the following list of additional vector types and aliases, which are also available when the '`<moviVectorUtils.h>`' header file is included:

Type	Description	Aliases
longlong2	2 x 64-bit signed integer	
ulonglong2	2 x 64-bit unsigned integer	

Type	Description	Aliases
longlong3	3 x 64-bit signed integer	
ulonglong3	3 x 64-bit unsigned integer	
long3	3 x 32-bit signed integer	
ulong3	3 x 32-bit unsigned integer	
int3	3 x 32-bit signed integer	
uint3	3 x 32-bit unsigned integer	
float3	3 x 32-bit floating-point	
short3	3 x 16-bit signed integer	
ushort3	3 x 16-bit unsigned integer	
half3	3 x 16-bit floating-point	
schar3	3 x 8-bit signed integer	
char3	3 x 8-bit integer	
uchar3	3 x 8-bit unsigned integer	byte3
longlong4	4 x 64-bit signed integer	
ulonglong4	4 x 64-bit unsigned integer	
longlong8	8 x 64-bit signed integer	
ulonglong8	8 x 64-bit unsigned integer	
long8	8 x 32-bit signed integer	
ulong8	8 x 32-bit unsigned integer	
int8	8 x 32-bit signed integer	
uint8	8 x 32-bit unsigned integer	
float8	8 x 32-bit floating-point	
longlong16	16 x 64-bit signed integer	
ulonglong16	16 x 64-bit unsigned integer	
long16	16 x 32-bit signed integer	
ulong16	16 x 32-bit unsigned integer	
int16	16 x 32-bit signed integer	
uint16	16 x 32-bit unsigned integer	
float16	16 x 32-bit floating-point	
short16	16 x 16-bit signed integer	
ushort16	16 x 16-bit unsigned integer	
half16	16 x 16-bit floating-point	

Table 6. Additional Supported Vector Data Types

These additional types are implemented using the underlying natively supported types, so performance is not necessarily improved because of smaller size, and may indeed be reduced (store to memory for example might be affected).

While the '<moviVectorUtils.h>' header file does not support vector data types of other sizes, the CLang vector extensions syntax does allow the programmer to specify vector types with an arbitrary number of elements. For example:

```
typedef int int42 __attribute__((ext_vector_type(42)));
```

If specified, moviCompile attempts to honor the semantics of these extended non-native types, but as it has to map them to the set of natively supported types, there may be some unspecified and undetermined performance degradation which may not be linear. That is, an algorithm tuned to work with 'int4' does not necessarily have equivalent performance if expressed using 'int42', as moviCompile has to determine how to represent the non-native data type in terms of the available native data types, and may make less optimal choices than the programmer might expect. Best performance always results from tuning the algorithm to the set of natively implemented vector data types.

NOTE: Any vector types other than those specifically identified in the tables above are not supported by moviCompile and their behavior if used is unspecified.

NOTE: moviCompile does not currently support vectors of 'bool', 'double', 'long double' or 'wchar_t'.

6.3.2.5.1 *Special Note About 3-Element Vectors*

The 'moviVectorUtils.h' header declares a set of vectors which have 3-elements. These have been added to provide a range of vector types that are compatible with those defined by the popular OpenCL Standard.

However, CLang and hence moviCompile treats these as-if they are 4-element vectors. This affects the size of the vector, its alignment and more importantly how much data is read and written from and to memory.

This is intended behavior in CLang which is the OpenSource C and C++ compiler frontend on which moviCompile is derived, and is implemented this way to support the OpenCL v1.2 Standard which specifies that this is the required behavior.

This note is to alert programmers to this unintuitive implementation, and the type declarations for these 3-element vectors will produce a “deprecated” warning when they are used in your program, cross-referencing this section.

While moviCompile only provides declared vectors with 2, 3, 4, 8 and 16 elements, in more general terms CLang will actually handle all vector types as-if the number of elements is equal to the number specified rounded up to the nearest power of 2.

6.3.2.6 *Matrix data-types*

The vector data types represent a set of one-dimensional matrix data types, and the Vector Utility Library also supports a limited set of two-dimensional Matrix data types. These matrix data types are used by transpose and rotate.

The Matrix data types are implemented as C struct aggregates containing an array of 4 or 8 vector elements. These vectors form the rows, so access to the elements of a matrix uses a combination of member selection and array subscripting. For example:

```
#include <moviVectorUtils.h>

int4x4 matrix  = {{{ 1,   2,   3,   4 },
                   { 5,   6,   7,   8 },
                   { 9,  10,  11,  12 },
                   { 13, 14,  15,  16 }}};
int4   row     = matrix.rows[2];     // Selects '{9, 10, 11, 12}'
int    element = matrix.rows[2][1];  // Selects '10'
```

The following is the complete set of supported matrix data-types:

Type	Description
long4x4	4 x 4 x 32-bit signed integer
ulong4x4	4 x 4 x 32-bit unsigned integer
int4x4	4 x 4 x 32-bit signed integer
uint4x4	4 x 4 x 32-bit unsigned integer
float4x4	4 x 4 x 32-bit floating-point
short8x8	8 x 8 x 16-bit signed integer
ushort8x8	8 x 8 x 16-bit unsigned integer
half8x8	8 x 8 x 16-bit floating-point

Table 7. Matrix Data Types

6.3.2.7 *Supported operations*

For each of the data types supported, moviCompile also supports the following infix operations on the vector types, where the operands to the operation are vectors with the same number of elements, and whose underlying C type is compatible with the operation:

- Arithmetic Binary: +, -, *, / and % (any compatible arithmetic scalar or vector type)

• Arithmetic Unary:	`+`, `-` (any compatible arithmetic scalar or vector type)
• Bitwise Binary:	`|`, `&` and `^` (integer scalar and vector types only)
• Bitwise Unary:	`~` (integer scalar and vector types only)
• Assignment:	`=` (any compatible arithmetic scalar or vector type)
• Compound Assignment	`@=` (any compatible arithmetic scalar or vector type)
• Shifting:	`>>` and `<<` (integer scalar and vector types only)

The increment (++) and decrement (--) operators are not supported for vectors.

Vectors can be subscripted using the normal C subscripting operator `[]` as if it were a regular C array. However, the C equivalence between '`vec[index]`' and '`*(vec+index)`' is not supported:

- Subscripting: `[]` (as if the vector was a normal C array)

Vectors of compatible types with the same number of elements can be compared, and the result of the comparison is a vector of a signed integer with the same size and number of elements as the vectors which are being compared; `schar`*N*, `short`*N* or `int`*N*. The values in this resulting vector are '`0`' for false, and '`-1`' for true:

- Relational: `==`, `!=`, `<`, `<=`, `>` and `>=` (any compatible arithmetic vector type)

The address of a vector object can be taken in the usual way, and results in a pointer-to a vector:

- Unary Address-of: `&` (any vector type)[11]

The logical operators `&&`, `||` and `!` are not supported as there is no commonly accepted meaningful scalar Boolean value for a vector. However, the programmer may use the functions `mvuAll` (see section 6.3.5.8) or `mvuAny` (see section 6.3.5.9) to derive Boolean meaning from the resulting comparison vector.

The ternary selection operator `?:` is supported with the constraint that the types of the two outcomes are either the same vector-type or a vector-type which can be implicitly converted to the type of the other. The conditional-expression must be a scalar Boolean expression.

The `sizeof` operator is supported for all vector types (See also Section 6.3.2.5.1).

Conversions between vectors with the same number of elements is supported, some implicitly and others requiring a cast. Normal C cast notation can be used when a cast is required. Implicit conversions[12] are supported for the following data types:

- `int`*N* to `long`*N*
- `uint`*N* to `ulong`*N*

All other conversions between vector types require an explicit cast or the use of the '`mvuConvert_`*toType*' set of functions (see section 6.3.4.1).

6.3.2.8 Supported built-ins

In addition to the supported types and operations, the vector extensions provide limited function-style support for shuffling (or swizzling) vectors, specifically the `CLang` extension (see also section 8.3):

- `__builtin_shufflevector(vec1, vec2, index0, index1, … index`*N*`)`

[11] **NOTE**: It is not possible to take the address of an element of a vector type. This is related to the lack of equivalence between the subscript operator '`[]`' for vectors and pointer addition.

[12] While implicit conversions are very limited, at some time in the future `moviCompile` will support implicit conversions between vector types that mirror the Usual Arithmetic Conversions of their scalar counterparts; but at this time `CLang` on which `moviCompile` is based, does not yet implement this functionality.

This corresponds to the OpenCL functions `shuffle` and `shuffle2` but differs in the arguments. The arguments `vec1` and `vec2` must be of the same vector type, and this function returns a vector of the same type.

The result vector is a “shuffled” combination of the elements of the two source vectors, where the values of the indexed elements are copied from the source vectors to the destination vector. If there are `N` elements in the vector type, then there are `2N` indices. Each `indexM` corresponds to the `M`th element in the destination vector, while the value of the index corresponds to an element in the source vectors. The values `0` thru `N-1` correspond to the `0`th through `(N-1)`th element of `vec1`, while the value `N` thru `2N-1` correspond to the `0`th through `(N-1)`th element of `vec2`.

For example:

```
int4 vec1   = { 1, 2, 3, 4 };
int4 vec2   = { 100, 200, 300, 400 };
int4 result = __builtin_shufflevector(vec1, vec2, 0, 4, 1, 5);
```

will set `result` to the values `{ 1, 100, 2, 200 }`. A more complete `CLang` description of this function can be found at:

http://clang.llvm.org/docs/LanguageExtensions.html#langext-builtin-shufflevector

Although the built-in may be used in this way, the programmer should be aware that the same outcome can be achieved using the vector extensions to the C language provided by `moviCompile`. The example above can be written using these extensions and without requiring the use of the built-in, thus:

```
int4 vec1   = { 1, 2, 3, 4 };
int4 vec2   = { 100, 200, 300, 400 };
int4 result = { vec1.s0, vec2.s0, vec1.s1, vec2.s1 };
```

this will produce identical code to the example using ‘`__builtin_shufflevector`’ directly, and is generally more portable.

6.3.3 Enhanced Vector Extensions

6.3.3.1 Swizzle support

Related to the shuffle built-in previously described, `moviCompile` provides an “infix” language extension that provides the programmer with fine-grained control over how a vector may be “swizzled”; that is, how its elements may be selectively copied and moved in a single expression. This is limited to vectors that have a maximum of four elements of a 32-bit data type (`int`, `unsigned int`, `long`, `unsigned long`, `float` and `double`.

This uses the common “`xyzw`” notation, for example:

```
int4 a, b, c;
a.w  = 1;       // Just set the 4th element of ‘a’ to ‘1’
a = b + c.yyyy; // Add the 2nd element of ‘c’ to each element of ‘b’ and
                   store the result in ‘a’
c = a.wzyx;     // Reverse the vector elements
```

Alternatively the more general “`s012..F`” notation can be used, especially for vectors with up to 16 elements, for example:

```
short8 a, b, c;
a.s2  = 1;           // Just set the 3rd element of ‘a’ to ‘1’
a = b + c.s44444444; // Add the 5th element of ‘c’ to each element of ‘b’ and
                        store the result in ‘a’
c = a.s76543210;     // Reverse the vector elements
```

For vectors with an even number of elements, the upper and lower halves of these vectors can be accessed using the ”hi” or “lo” notation, for example:

```
short8 a;
short4 b, c;

b = a.lo;         // Same as ‘a.s0123’
c = a.hi;         // Same as ‘a.s4567’
```

For vectors with an even number of elements, the alternate odd or even elements of these vectors can be accessed using the ”odd” or “even” notation, for example:

```
short8 a;
short4 b, c;

b = a.odd;        // Same as ‘a.s1357’
c = a.even;       // Same as ‘a.s0246’
```

6.3.4 Conversions

In addition to supporting a limited set of implicit conversions between compatible vector types, and the restricted set of supported conversions using explicit cast notation, the support library for explicit conversions also provides a function-style conversion interface. Where appropriate and possible, these conversions are implemented as high-efficiency intrinsics, but may be implemented using a traditional function provided in the support library where an intrinsic form is not feasible.

6.3.4.1 Standard conversions

All the standard conversions previously described (other than the reinterpret casts) can also be expressed using the function-style equivalent. For each *toType* and *fromType* pair from the set of standard conversions, a function of the following form is provided:

```
mvuConvert_toType(fromType)
```

for example:

```
mvuConvert_uint4(int4)
```

This function-style interface is also available for the underlying scalar types, i.e. `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `half`, `float` and `double`.

If the *toType* and *fromType* are vectors, they must have the same number of elements.

The *toType* and *fromType* may be the same.

6.3.4.2 Converting between vectors with different numbers of elements

Conversions between vectors with differing numbers of elements is not supported by `moviCompile`. However, it is possible to use the usual element access syntax to do this explicitly, for example:

```
int4 bigVec      = { 0, 1, 2, 3 };
int3 littleVec1 = { bigVec[0], bigVec[1], bigVec[2]};
int3 littleVec2 = bigVec.s012;
int3 littleVec3 = bigVec.xyz;
```

6.3.4.3 Saturated conversions

Saturated conversions are not currently supported, but may be supported in a future release of the Vector Utility Library.

6.3.4.4 Reinterpret conversions

All types (*fromType*) may be reinterpreted as being of another type (*toType*) provided the two types have the same size (`sizeof`). No conversion takes place and the bits representing the value are unchanged. The result of a reinterpret conversion is unspecified (as with Standard C). Reinterpret conversions are provided for all supported vector and scalar types using functions of the form:

```
mvuAs_toType(fromType)
```

Other than size, there are no restrictions on the types involved.

6.3.5 Vector Functions

The explicit vectorization extensions provide a limited number of additional functions that may be used by the programmer. Where possible these are implemented optimally using intrinsics, but may be provided in a conventional library where an intrinsic implementation is not presently possible or provides no advantage.

These functions are inspired by the corresponding functions provided by OpenCL v1.2, and wherever possible their use and behavior is identical to the use and behavior of the corresponding OpenCL function. For example, the function '`mvuCross`' provided by the Movidius Vector Utility Library, is intended to mirror the purpose and use of the OpenCL function '`cross`'.

6.3.5.1 Cross product '`mvuCross()`'

The following function is provided for the vector type `float4`:

```
float4 mvuCross(float4 a, float4 b)
```

6.3.5.2 Dot product '`mvuDot()`'

The following set of functions are provided for all supported native vector types[13] `vecN`:

```
vecN mvuDot(vecN a, vecN b)
```

6.3.5.3 Minimum '`mvuMin()`'

The following set of functions are provided for all supported native vector types `vecN`:

```
vecN mvuMin(vecN a, vecN b)
```

6.3.5.4 Maximum '`mvuMax()`'

The following set of functions are provided for all supported native vector types `vecN`:

```
vecN mvuMax(vecN a, vecN b)
```

6.3.5.5 Absolute '`mvuAbs()`' and '`mvuAdiff()`'

The following set of functions are provided for all supported native vector types `vecN`:

```
uvecN mvuAbs(vecN a)
uvecN mvuAdiff(vecN a, vecN b)
```

When the argument is an integer vector type, then '`uvecN`' is the unsigned variant of the argument vector type '`vecN`'. For floating-point vector arguments, '`uvecN`' is the same as the argument vector type '`vecN`'.

[13] See section 6.3.2.1 for the list of supported native vector types.

6.3.5.6 Transpose '`mvuTranspose()`'

The matrix transpose function performs the normal matrix transpose operation, but is limited to vectors of 4x4 elements of a 32-bit data type and 8x8 elements of a 16-bit data type.

The following functions are provided:

```
int4x4    mvuTranspose(int4x4)
uint4x4   mvuTranspose(uint4x4)
long4x4   mvuTranspose(long4x4)
ulong4x4  mvuTranspose(ulong4x4)
float4x4  mvuTranspose(float4x4)

short8x8  mvuTranspose(short8x8)
ushort8x8 mvuTranspose(ushort8x8)
half8x8   mvuTranspose(half8x8)
```

6.3.5.7 Rotate '`mvuRotate_left()`' and '`mvuRotate_right()`'

The matrix rotate function performs the normal matrix transpose operation, but is limited to vectors of 4x4 elements of a 32-bit data.

The following functions are provided:

```
int4x4    mvuRotate_left(int4x4)
uint4x4   mvuRotate_left(uint4x4)
long4x4   mvuRotate_left(long4x4)
ulong4x4  mvuRotate_left(ulong4x4)
float4x4  mvuRotate_left(float4x4)

int4x4    mvuRotate_right(int4x4)
uint4x4   mvuRotate_right(uint4x4)
long4x4   mvuRotate_right(long4x4)
ulong4x4  mvuRotate_right(ulong4x4)
float4x4  mvuRotate_right(float4x4)
```

Where '`mvuRotate_left`' means anti-clockwise (-90° or +270°) and '`mvuRotate_right`' means clockwise (+90°).

6.3.5.8 Relational '`mvuAll()`'

The following set of functions are provided for all supported native integer vector types `vec`*N*:

```
int mvuAll(vecN a)
```

The '`mvuAll`' function checks to see if the Most Significant bit (MSb) of each of the elements of a vector of an integral data type is set to one. The value returned is of type '`int`' and has the value '`1`' if all of the elements in the vector have their MSb set to '`1`', otherwise it has the value '`0`'.

This definition and the use of the term MSb is for consistency with the OpenCL function '`all`'.

6.3.5.9 Relational '`mvuAny()`'

The following set of functions are provided for all supported native integer vector types `vec`*N*:

```
int mvuAny(vecN a)
```

The '`mvuAny`' function checks to see if the Most Significant bit (MSb) of any of the elements of a vector of an integral data type is set to one. The value returned is of type '`int`' and has the value '`1`' if any of the elements in the vector have their MSb set to '`1`', otherwise it has the value '`0`'.

This definition and the use of the term MSb is for consistency with the OpenCL function '`any`'.

6.4 ARM NEON Intrinsic Bindings Library ‘`mlibneon.a`’

This library provides `SHAVE` bindings to the intrinsics for ARM® NEON™ SIMD technology[14].

The ARM processor supports a large range of specialized instructions for 64-bit and 128-bit vectors. These are called the NEON instructions, and native compilers for the ARM processor implement a suite of intrinsics that allow the programmer to explicitly use these instruction from C. These intrinsics are described in section 18 of the ARM document “DUI0472K ARM CC User Guide” which is located at the following location:

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0472k/chr1359125040250.html

The library ‘`mlibneon.a`’ provides an implementation of these intrinsics as `SHAVE` functions. The programmer may use this binding library by adding the following to their code:

```
#include <moviNeonBindings.h>
```

And linking their program with the library ‘`mlibneon.a`’ in addition to the other libraries.

Some of the NEON intrinsics have a direct correspondence to a `SHAVE` instruction and these will have the most optimal implementation. However, there are many NEON intrinsics for which there is no `SHAVE` equivalent, and these are emulated with varying levels of performance.

The primary intention of these bindings however is not performance, but to allow programs that currently use the ARM NEON intrinsics interface to be compiled and run on `SHAVE`. It is not intended as a primary or optimal approach for writing performant `SHAVE` applications.

The intrinsics for the ‘`poly_t`’ data type and for Fixed-Point data types are not supported.

NOTE: The ARM NEON Bindings library requires that the program is compiled using ISO C99 or ISO C++98 or later. This is because the 64-bit integer data types are ‘`long long`’, and this data type did not exist in the earlier version of the ISO Standard.

[14] ARM is a registered trademark of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. NEON is a trademark of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

7 Attributes and Macros

7.1 Attributes

`moviCompile` supports many `GCC` style attributes for variables, functions and type definitions. These can be specified with the keyword '`__attribute__`'.

Attributes not specifically mentioned here are not supported, even if they appear to be accepted by `moviCompile`.

Currently, the following attributes are specifically supported:

- `aligned(`*`number`*`)` – tells the compiler to align the function/variable to `number` boundary. `number` must be a power of 2. **NOTE:** `moviCompile` does not support this attribute for non-static local variables
- `always_inline` – tells the compiler always to inline this function (this is overridden by the command-line option '`-fno-inline-functions`')
- `const` –tells the compiler that the function does not alter memory or have other side-effects, and that if it is called with the same parameters, that it will yield identical results. This allows the compiler to optimize multiple calls to the same function with the same parameters, to a single call to the function and reuse the result from the first call when possible
- `constructor` and `constructor(`*`number`*`)` – functions that will be called prior to the execution of '`main`' during program initialization (see section 10.5.1)
- `destructor` and `destructor(`*`number`*`)` – functions that will be called after the execution of '`main`' during program finalization (see section 10.5.1)
- `dllexport` – tells the compiler to place the address of this function in the IAT (Indirect Address Table). This is intended to facilitate the implementation of "Entry-Points" which are `SHAVE` functions intended to be started from the `Leon` code – see section 9.11 for more details
- `ext_vector_type(`*`n`*`)` – extended vector type, `n` is the number of elements. It can only be used with `typedef`s. This is preferred over using `vector_size(`*`size`*`)`
- `noinline` – tells the compiler never to inline this function
- `noreturn` – tells the compiler that this function will not return to the caller (e.g. `abort`, `exit`). A function with this attribute, but which does return, has undefined behavior
- `optnone` – tells the compiler to disable the optimization of this function
- `overloadable` – allows the function to be overloaded in C. Such functions will have their names mangled as for C++ functions (see section 9.10)
- `packed` – tells the compiler to use the minimum required memory to represent a type, but can result in less performant code
- `section("`*`name`*`")` – tells the compiler which section to place the current function or variable during linking, overriding the default sections the compiler would usually choose (the section "*`name`*" may only be composed of alphanumeric characters, '`_`' and '`.`')
- `vector_size(`*`size`*`)` – tells the compiler the size of the vector type measured in Bytes
- `weak` – allows a definition or declaration to be identified as having weak external linkage

One, two or more attributes can be specified for the same declaration, either as separate attributes, or as a comma-separated list of attributes; for example:

```
void func() __attribute__((noreturn)) __attribute__((always_inline))
```

is exactly equivalent to:

```
void func() __attribute__((noreturn,always_inline))
```

All attributes can be written as above, or optionally with a double-underscore prefix and suffix, thus:

```
void func() __attribute__((__noreturn__,__always_inline__))
```

Some examples of these are:

```
// An attribute specified in the function declaration
void function1(void) __attribute__((section("section1")));
// or
void __attribute__((section("section1"))) function1(void);

// For function definitions, the attribute cannot be specified
// after the function name as it can be for declarations
void __attribute__((aligned(32))) function2(void) {...}

// A function with 2 attributes
void __attribute__((noinline,noreturn)) function3() {...}
// or
void __attribute__((noinline) __attribute__((noreturn)) function3() {...}

// An 'int' variable with an attribute
int x __attribute__((aligned(16)));
// or
int __attribute__((aligned(16))) x;

// This structure is represented in 5-Bytes and has an alignment of 1-Byte
// without the packed attribute it would be represented in 8-Bytes with a 4-
// Byte alignment
struct __attribute((packed)) S {
  char s;
  int x;
};
```

7.2 moviCompile Predefined Macros

The usual ISO C and C++ predefined macros are provided and also the standard set of macros predefined by CLang, but in addition moviCompile also predefines a large number of additional macros to help the programmer to write more portable multi-platform and version independent code:

Pre-Defined Macro	Description

`__LITTLE_ENDIAN`
`__LITTLE_ENDIAN__`
 	
`__ENDIAN_LITTLE`
`__ENDIAN_LITTLE__`
	These are always defined and tell the program that the SHAVE is a little-endian architecture. The first pair are compliant with GCC conventions, while the additional pair satisfy an alternative convention expected by some 3rd party libraries

`__MOVICOMPILE__`	This has a value which is the version of the compiler being used, for example: `00.90.0 Build 3000`
`__MOVICOMPILE_MAJOR__`	This is the major release number and has the same value as the first element of the version quartet – e.g. '00'

Pre-Defined Macro	Description
`__MOVICOMPILE_MINOR__`	This is the minor release number and has the same value as the second element of the version quartet – e.g. ‘`90`’
`__MOVICOMPILE_PATCH__`	This is the patch number for the release and has the same value as the third element of the version quartet – e.g. ‘`0`’
`__MOVICOMPILE_BUILD__`	This is the build number for the release and has the same value as the fourth element of the version quartet. This value is derived from the version control system used by `moviCompile` for the set of sources used to build this release – e.g. ‘`3000`’
`__shave` `__shave__`	The target processor is `SHAVE`

`__myriad2`
`__myriad2__`
	The target instruction set is for `SHAVE` ISA v2. These are present when any of the options:
	`-mcpu=myriad2.1   {deprecated}`
	`-mcpu=myriad2.2`
	`-mcpu=myriad2.3`
	`-mcpu=ma2100      {deprecated}`
	`-mcpu=ma2150`
	`-mcpu=ma2155`
	`-mcpu=ma2450`
	`-mcpu=ma2455`
	`-mcpu=ma2x5x`
	`-mcpu=ma2080`
	`-mcpu=ma2085`
	`-mcpu=ma2480`
	`-mcpu=ma2485`
	`-mcpu=ma2x8x`
	are specified, or by default if no ‘`-mcpu`’ option is specified.
	If ‘`myriad2.1`’ or ‘`ma2100`’ is chosen, these will have the value ‘`1`’. The MA2100 chip is no longer formally supported.
	If no ‘`-mcpu`’ option is specified or when ‘`myriad2.2`’, ‘`ma2150`’, ‘`ma2155`’, ‘`ma2450`’, ‘`ma2455`’ or ‘`ma2x5x`’ is chosen these will have the value ‘`2`’.
	If ‘`myriad2.3`’, ‘`ma2080`’, ‘`ma2085`’, ‘`ma2480`’, ‘`ma2485`’ or ‘`ma2x8x`’ is chosen these will have the value ‘`3`’.

Pre-Defined Macro	Description

`__ma2100` and `__ma2100__`
`__ma2150` and `__ma2150__`
`__ma2155` and `__ma2155__`
`__ma2450` and `__ma2450__`
`__ma2455` and `__ma2455__`
`__ma2080` and `__ma2080__`
`__ma2085` and `__ma2085__`
`__ma2480` and `__ma2480__`
`__ma2485` and `__ma2485__`
	When '`-mcpu=ma2100`' is specified[15].
	When '`-mcpu=ma2150`' is specified.
	When '`-mcpu=ma2155`' is specified.
	When '`-mcpu=ma2450`' is specified.
	When '`-mcpu=ma2455`' is specified.
	When '`-mcpu=ma2080`' is specified.
	When '`-mcpu=ma2085`' is specified.
	When '`-mcpu=ma2480`' is specified.
	When '`-mcpu=ma2485`' is specified.

`__ma2x5x` and `__ma2x5x__`
 	
`__ma2x8x` and `__ma2x8x__`
	When no '`-mcpu`' option is specified', or whenever any of the following are specified:
	`-mcpu=ma2150`
	`-mcpu=ma2155`
	`-mcpu=ma2450`
	`-mcpu=ma2455`
	`-mcpu=ma2x5x`
	`-mcpu=myriad2.2`
	Whenever any of the following are specified:
	`-mcpu=ma2080`
	`-mcpu=ma2085`
	`-mcpu=ma2480`
	`-mcpu=ma2485`
	`-mcpu=ma2x8x`
	`-mcpu=myriad2.3`

`__SHAVE_BRANCH_DELAY_SLOTS__`	This defines a number which is equal to the number of delay slots following a branch instruction. This defaults to '`6`'
`__SHAVE_MEMORY_LOAD_LATENCY__`	This defines a number which is equal to the number of cycles following a memory load before the destination register contains the loaded value. This defaults to '`6`'

Table 8. `moviCompile` Predefined Macros

7.2.1 Predefined MACROS and GCC for `Leon` Compatibility

Although this document is not about the `GCC` Tool-Chain used for development of the `Leon` parts of a Myriad application, we endeavour to keep a high degree of compatibility regarding the CPU information and predefined macros.

Both `moviCompile` and `sparc-myriad-rtems-gcc` support the same values for the '`-mcpu=<`*`target`*`>`' option, and will predefine the following set of macros in the same way:

- '`__maXXXX__`' and '`__maXXXX`'
- '`__myriad2__`' and '`__myriad2`'

This makes it possible for source code to be shared between the `SHAVE` and `Leon` parts of a program, which can be conditionally compiled as appropriate to the particular chip or ISA selected (see Section 1.3).

[15] This is also defined if the selected CPU is '`myriad2.1`' – this is because there is only one chip variant using the `SHAVE` ISA v2.1.

To differentiate between the CPUs, the moviCompile compiler for SHAVE predefines the macros '__shave__' and '__shave', while the sparc-myriad-rtems-gcc compiler for Leon predefines the macros '__sparc__', '__sparc' and '__leon__'.

In addition, the GCC sparc-myriad-rtems-gcc compiler for Leon also predefines the following set of version information (the actual values depend on the specific version of sparc-myriad-rtems-gcc):

- '__MOVITOOLS__' with the value '00.90.0.146412'
- '__MOVITOOLS_MAJOR__' with the value '00'
- '__MOVITOOLS_MINOR__' with the value '90'
- '__MOVITOOLS_PATCH__' with the value '0'
- '__MOVITOOLS_BUILD__' with the value '146412'

These are analogous to the '__MOVICOMPILE*' group of predefined macros provided by the moviCompile compiler for the SHAVE CPU.

8 Built-ins / Intrinsics

8.1 Built-in "Instruction-Set" Intrinsics

`moviCompile` provides a large set of intrinsics that expose a significant fraction of the rich `SHAVE` instruction-set (ISA) to the programmer. These represent a very high fidelity binding between the instructions they represent and the C or C++ program.

Unlike inline assembler, these intrinsics are tightly coupled to the compiler's code generation, which allows it to perform a large range of normal optimizations and instruction scheduling, as it has detailed knowledge of how they interact with the scheduler and register allocator.

While these are automatically present (no header file is included to make them available), a "pseudo header file" is provided to illustrate the implicit function prototype interface they use. This pseudo header file is provided in the following location:

```
common/moviCompile/include/shave-builtin-pseudo-declarations.h
```

It is not a true header file in that it does not expose any actual interface, and is intended for documentary assistance. Each intrinsic "declaration" is identified by the corresponding instruction from the instruction set.

The set of provided instruction-set intrinsics consists of several hundred built-ins that bind to discrete instructions from the `BRU`, `CMU`, `IAU`, `LSU`, `SAU` and `VAU` functional-units. With each release of the compiler, the set of provided built-ins is typically extended to provide access to additional instructions.

The instruction-set intrinsics follow a fixed naming pattern[16] which can be verbose, but the programmer can use a macro or inline function to "wrap" these long-names should they wish to do so. The general naming convention is as follows (all lowercase):

```
__builtin_shave_<fu>_<opcode>[_<modifier>]_<operands>[_<other>]
```

Where the fields are:

• *<fu>*	is the name of the functional-unit, e.g. '`vau`'
• *<opcode>*	is the name of the opcode for the *<fu>*, e.g. '`add`'
• *<modifier>*	if present, usually the type or conversion type, e.g. '`f32`' for '`float`'
• *<operands>*	an encoded set of argument types, e.g. '`rr`'
• *<other>*	occasionally required when the name cannot be otherwise discriminated

So the Vector Unit `ADD` instruction for a vector of 4 x 32-bit floating-point (`float4`):

```
VAU.ADD.f32 rDst, rSrc1, rSrc2
```

is represented by the following built-in ISA intrinsic:

```
float4 __builtin_shave_vau_add_f32_rr(float4 rSrc1, float4 rSrc2)
```

NOTE: When the '*<operand>*' is the letter '`i`', the corresponding built-in's argument "must be" a compile-time integer constant. See Section 8.1.1 for more information.

8.1.1 Limitations

These intrinsics are NOT functions, and have some limitations on use. Most important are the "immediate" arguments for instructions such as `VAU.SWZBYTE` which takes an input vector argument and returns the result as another vector. In addition to the vector arguments it has four "immediate" arguments which "must be" provided as literal constants and not as variables. These are the "scalar" '`char`' arguments in the declaration. The reason for this is that there is no possible method for these values to be provided using a

[16] This is not true for the Meta ISA intrinsics described in section 8.1.2 as these do not correspond to true instructions

register, as they must be actual hard coded values at the time the code is being passed through the assembler. Thus given the following intrinsic:

```
int4 __builtin_shave_vau_swzbyte_i32_riiii(int4 variable_1,
                                           char immediate_1,
                                           char immediate_2,
                                           char immediate_3,
                                           char immediate_4);
```

The following fragment works:

```
int4 input = { 1, 2, 3, 4 };
int4 output = __builtin_shave_vau_swzbyte_i32_riiii (input, 3, 3, 1, 0);
// Corresponds to:
//   VAU.SWZBYTE.i32 output, input [0133]
```

while the following does not:

```
int4 function(char arg) {
  int4 input = { 1, 2, 3, 4 };
  int4 output = __builtin_shave_vau_swzbyte_i32_riiii (input, 3, arg, 1, 0);
  return output;
}
```

The pseudo-declaration header file names the arguments in the prototype using '`variable_`*N*' for each argument which can be any variable or value of the appropriate type, and uses the name '`immediate_`*N*' for each argument which must be a constant-literal.

Using a variable where an immediate constant-literal is expected will result in a compile-time error.

It is not possible to take the address of a built-in intrinsic.

8.1.2 64-bit Vector Meta Instruction Set Intrinsics

Although `SHAVE` has no 64-bit vector instructions, it is possible to use the lower half of a VRF with a VAU instruction to perform many 64-bit vector operations. However, the C type system does not make it easy to do this in conjunction with the usual 128-bit ISA intrinsics described above. For example:

```
int4 a, b, c;
c = a + b;  // Implicitly selects VAU.VADD.i32
c = __builtin_shave_vau_add_i32_rr(a, b);  // Explicitly select
                                           // VAU.ADD.i32
```

this will work perfectly as expected, but the following will fail:

```
int2 x, y, z;
z = x + y;  // Implicitly selects VAU.VADD.i32, this is okay
z = __builtin_shave_vau_add_i32_rr(x, y);  // But this fails
```

To make access to the VAU intrinsics[17] more useful to programmers using 64-bit vectors, many of the VAU instruction intrinsics have been supplemented with a set of corresponding "Meta ISA" intrinsics, but which are designed for type compatibility with 64-bit vectors occupying the low 64-bits of a VRF.

This allows the previous desired use case to be written as:

```
int2 x, y, z;
z = x + y;  // Implicitly selects VAU.VADD.i32
z = __builtin_shave_vau_add_v2i32_rr(x, y);  // Explicitly select
```

[17] While we generally try to preserve the names of the intrinsics across releases, the names of these "Meta ISA" intrinsics may need to be changed in a future release as they currently do not follow a regular naming convention.

```
                                                     // VAU.ADD.i32
```

8.2 Special Built-ins

In addition to supporting intrinsics that map directly to a `SHAVE` instruction, `moviCompile` also provides some additional built-ins to facilitate commonly used programming idioms for `SHAVE`.

8.2.1 Get the SHAVE CPU ID

The following built-in will retrieve the processor number for the `SHAVE` that the program is executing on:

```
int __builtin_shave_getcpuid(void);
```

This is a commonly required feature and is often implemented using inline-assembly code. The inline-assembler has a number of disadvantages, the main one being that it defeats the optimizer which has to assume worst case schedules; and inline-assembly code cannot be scheduled by the compiler with other compiler generated instructions.

This built-in is also specially attributed to tell the compiler that if it is called more than once, that it is guaranteed to always yield the same value, so the optimizer may be able to optimize better if it is called more than once in the programmer's code.

8.2.2 Exclusive Loads

There are occasions when the programmer wants to load a value from memory, but needs to be assured that the other LSU is not performing a memory load at the same time. The following built-ins will guarantee that the compiler schedules a memory load on only one LSU in a single cycle, and will suppress loading from occurring simultaneously on the other LSU:

```
unsigned long long
    __builtin_shave_lsu_ld64_r_exclusive(
        const volatile unsigned long long *);  /* LSU.LD64 */
unsigned int
    __builtin_shave_lsu_ld32_r_exclusive(
        const volatile unsigned int *);        /* LSU.LD32 */
unsigned short
    __builtin_shave_lsu_ld16_r_exclusive
        (const volatile unsigned short *);     /* LSU.LD16 */
unsigned char
    __builtin_shave_lsu_ld8_r_exclusive(
        const volatile unsigned char *);       /* LSU.LD8 */
```

Because of how the code-generator selects which LSU to use, this will in most cases select LSU0 (the default LSU load policy). But regardless of whether it selects LSU0 or LSU1, it guarantees that the other LSU is not also being used to perform a memory load.

NOTE: Although the signature accepts pointers to '`const volatile`' integers, this is simply to allow the C type system to permit any CV-qualified pointer to the corresponding integer type as an argument, and does not infer any guarantee on the ordering of the evaluation of the intrinsic which is not itself a '`volatile`' access. To ensure '`volatile`' semantic sequencing, please use normal pointers to '`volatile`' objects.

8.3 Built-Ins provided by CLang

In addition to the built-ins specifically provided by moviCompile for SHAVE, the LLVM and CLang Open Source projects (on which moviCompile is based) provide a very large range of built-ins.

Generally it is not recommended that the programmer uses these built-ins directly, and they are present to facilitate the implementation of internal operations, or the libraries provided with the compiler. An example is the built-in '__builtin_va_start' which is used by LLVM to implement the internal mechanisms for variadic functions, but the programmer is never supposed to use this directly, and instead should use the ISO C interface described in the header '<stdarg.h>'.

Similarly there are built-ins which are internal and required to implement functionality such as C++ Atomics, and these should never be used directly by the programmer.

The CLang documentation is very specific about programmers using built-ins directly, and you should also read:

http://clang.llvm.org/docs/LanguageExtensions.html#id14

Also, the existence, structure and purpose of these built-ins are subject to change between releases, and using the C interfaces provided where applicable is the only portable and safe means of using them. And just as with the SHAVE specific built-ins, built-ins in general are not portable from target to target, or from compiler to compiler, and should be used with caution.

While the direct use of these built-ins is not recommended, and in general their direct use by the programmer is not supported by moviCompile, there are a small number of LLVM provided built-ins that may be useful, and their direct use is supported by moviCompile.

The pre-processor test '__has_builtin' should be used to check if a built-in is present, thus:

```
#ifdef __has_builtin(__builtin_shufflevector)
use '__builtin_shufflevector'
#else
use some other method
#endif
```

But even if a built-in is present does not mean that it is supported by moviCompile and the use of an unsupported built-in has unspecified behavior.

The remainder of this section describes the CLang built-ins that are specifically supported by moviCompile. The direct use by the programmer of any other built-ins not explicitly mentioned here is specifically not supported, even though they may appear in the headers and sources provided with moviCompile.

8.3.1 CLang Built-In '__builtin_shufflevector'

This is a very useful general purpose built-in for manipulating vectors. The full definition of this built-in can be found at:

http://clang.llvm.org/docs/LanguageExtensions.html#langext-builtin-shufflevector

An example of its use can be found in section 6.3.2.8. It is important to note that all of the capabilities of '__builtin_shufflevector' can be achieved using the C language extensions for vectors without having to use built-ins, but sometimes it is more convenient or clear to use this built-in directly.

8.3.2 CLang Built-In '`__builtin_expect`'

When the compiler is generating code for conditional expressions and control-flow, it uses a complex series of heuristics to decide the most probable outcome of a test in order to produce code that favors the most probable outcome. For example:

```
if (expression)
  do this code if true
else
  do this code if false
```

If the compiler's analysis indicates that '`expression`' is most likely to be TRUE, then the compiler will ensure that branching is minimized and the '`do this code if true`' is most likely to be present in cache. Alternatively, if its analysis indicates that it is more likely to be FALSE, then it would invert the code generation to favor '`do this code if false`'.

Most of the time it does not matter a lot, but sometimes when tuning the performance of some time-critical code, the compiler's choice may not be accurate, and the programmer may have more detailed knowledge about the probable truth of '`expression`', and would like to direct the compiler with this information.

This is where '`__builtin_expect`' can be used to provide a hint to the compiler. So in the example above, assume that the compiler's analysis indicates that '`expression`' is most likely to be TRUE, but the programmer has determined that it is most likely to be FALSE, then they can inform the compiler as follows:

```
if (__builtin_expect(expression, 0))
  do this code if true
else
  do this code if false
```

This built-in is not just for Boolean checks, and its pseudo interface is effectively:

```
long __builtin_expect(long test_expression, long most_probable_value)
```

so it may be used anywhere an expression is used that will affect the control-flow of the program, for example:

```
switch (__builtin_expect(expression), 5) {
case ...
case 5:
case ...
}
```

In this example, the programmer is telling the compiler that the most probable value for '`expression`' is '`5`' and that the code generation should favor that outcome if there is a performance advantage to doing so.

This should be used carefully and with great caution. A program that is tuned with such an explicit statement of the probable outcome of a test, may not be valid in the future after the program has been edited or when the structure of the real world data it acts on changes; and such tuning hints may now direct the compiler inaccurately. Also, when using profile-directed optimization[18], these hints will take precedence over the actual data gathered by profiling the application.

[18] `moviCompile` does not currently support profile-directed optimizations, but it is a feature that we plan on providing in some as yet to be determined future release.

9 Calling Convention and ABI

The `moviCompile` calling convention is presented in this section. This is of use when debugging, when calling `SHAVE` entry points from another (e.g. `Leon`) processor, or when writing assembly functions to directly call functions written in C, or to be callable from C.

When combined with an exact statement of how data is laid out in memory, this is commonly referred to as the Abstract Binary Interface, or ABI.

9.1 General Description of the Calling Convention

Calling conventions allow functions in a computer program to successfully call each other, even if they are defined in separate files, written in different languages, or compiled by different compilers. A calling convention describes precisely the way in which parameters to functions should be passed, where return values should go, what the assembly label associated with an object or function should look like, etc.

Additionally, a calling convention specifies certain assumptions which can, or must, be made by the function caller and the called function (the callee), for example the maximum latency of instructions which may have been scheduled before a function is called.

The main purpose of these conventions is to allow programs compiled with `moviCompile` to call functions written in `SHAVE` assembly, and vice-versa, as described earlier in this document. The calling convention has been designed to use registers where possible for maximum efficiency.

Failure to respect the calling convention may result in serious program failure.

9.2 Label Names Associated With Functions

Since the C language does not allow function overloading[19], every function in a C program with external linkage has an associated assembly label identical to its name. So, for a function called ‘`my_func`’, `moviCompile` will output an assembly implementation for the function using the label ‘`my_func`’:

For the following C function prototype;

```
<any return type> my_func(<any parameter list>);
```

`moviCompile` outputs the label:

```
my_func:
```

Overloading and C++ makes the label names more complex as it requires that the names are “mangled” in a systematic way. This is discussed in Section 9.10.

Names which have internal linkage (for example ‘`static`’ functions in C), the name is the same as for names with external linkage, but prefixed with the characters ‘`.I`’.

In addition, there are compiler generated symbols which have “Private” linkage such as the names of internal branch targets, and which are not published to the symbol table in the generated object file; these symbols are prefixed with the characters ‘`.L`’.

9.3 Stack Pointer

IRF register ‘`I19`’ is reserved as a dedicated stack pointer (SP). If a function requires `N` bytes of stack memory to store the results of intermediate computations then, upon invocation, the function should decrement the stack pointer by `X`, where `X` is the value `N` rounded up to the next multiple of 8, and increment the SP by `X`

[19] The attribute ‘`overloadable`’ may be used to allow functions to be overloaded in C, see Section 7.1

before returning. Between this decrement and increment, the function may freely use the memory range `*i19 ... (*i19) + (X – 1)` for its own purpose.

Example:

```
my_func:
                          ; Setup code for function
IAU.SUB    i19, i19, 8   ; Reserve 8 Bytes of stack
LSU0.LDIL  i0, 5         ; Load literal value into temporary register
LSU0.ST.32 i0, i19       ; Store the loaded value to the allocated stack
                          ; position
                          ; Do some computation
IAU.ADD    i19, i19, 8   ; Relinquish the allocated stack space before
...                       ; returning to the caller
```

The SP is key to maintaining the structure of memory, and should never be modified except to allocate and free space for stack variables. This means that it is not necessary to save the SP before a function call; by adhering to the calling convention, the called method promises to restore the SP to its original value before returning.

Also, to ensure memory alignment integrity, it is very important to ensure that the stack is always increased or reduced in multiples of 8-bytes.

9.4 Link Register (Return Address)

IRF register '`I30`' is used both to store the address to which control flow should jump when a function is called, and the return address to which control flow should jump after a function has completed execution. This is referred to as the link register (LR).

Example:

```
my_func:

LSU.LDIL i30, my_callee  ; Load low 16-bits of the address of the function
                          ; to be called
LSU.LDIH i30, my_callee  ; Load high 16-bits of the address of the function
                          ; to be called
BRU.SWP  i30              ; Set program counter to the value in i30, and
                          ; assign i30 to instruction following the BRU.SWP,
                          ; this is the return address

my_callee:
                          ; Instructions to implement my_callee must ensure
                          ; that the return address in i30 is preserved
BRU.jmp i30               ; Resume execution in the calling function
```

All execution paths through a function adhering to the calling conventions must finish with an unconditional jump to the address held in the LR and must ensure that at this point, the address in the LR is the same as at the return point for the function invocation.

If use of the LR is required during function execution (typically to call another function) then the register should be saved and restored appropriately.

9.5 Scalar Parameters

In this section, the scalar is used to denote a value of type `bool`, `long long`, `unsigned long long`, `long`, `unsigned long`, `int`, `unsigned int`, `short`, `unsigned short`, `char`, `unsigned char`, `signed char`, any enumerate type ('`enum`') or a pointer `T*` where `T` is any type. It also includes the 32-bit floating-point values of types ‘float’ and ‘double’, and the 16-bit floating-point type ‘short float’ (‘half’ or ‘__fp16’). In C++ the scalar type ‘bool’ is passed as if it is an ‘int’.

Up to 8 scalar parameters may be passed via the IRF, using registers i18, i17, ..., i11. The first (leftmost) scalar parameter goes in register i18, the second in register i17, etc.

64-bit integers are passed in a register pair, with the low numbered register containing the low 32-bits of the 64-bit integer, and the high numbered register contains the high 32-bits of the 64-bit integer. This also means that the number of scalar parameters that can be passed in registers is reduced by 1 for each 64-bit integer passed to the function.

Register i18, the first scalar parameter register, is also used to store the return value for a function call, if the return value is one of the above scalar types. For 64-bit integers, the value is returned in the register pair (i17 and i18).

Furthermore, if the return value cannot be passed in registers (non-POD values), then register i18 is used to pass the address where the return value is to be saved (see also section 9.5.3) and implicitly reduces the number of IRF registers available for the declared arguments.

Example:

Consider the following function prototype:

```
int average(int i, short s, char c, int* p);
```

The intended behavior of this function is that it should compute the integer average of integers i, s, c and the integer pointed to by p.

```
caller:
                                ; Assume at the point of the call, i, s, c
                                ; and p are in registers i1, i2, i3 and i4
                                ; respectively
        CMU.CPII  i18, i1       ; Copy the parameters to their registers, as
        CMU.CPII  i17, i2       ; defined by the calling convention
        CMU.CPII  i16, i3
        CMU.CPII  i15, i4
        LSU0.LDIL i30, average  ; Load the address of the function to be
        LSU0.LDIH i30, average  ; called
        BRU.SWP   i30           ; Call the function 'average'
        NOP <branch delay slots> ; On return, i18 should contain the value
                                ; returned by 'average'
    ..........

 average:
        IAU.ADD     i18, i18, i17 ; Accumulate the result into return register
                                  ; i18
        LSU0.LD.32  i0, i15       ; Value pointed to by i15 is fetched into
                                  ; a temporary register i0
        IAU.ADD     i18, i18, i16
        LSU0.LDIL   i1, 4
        NOP 4                     ; Wait for the load to complete
        IAU.ADD     i18, i18, i0
        SAU.DIV.i32 i18, i18, i1
        NOP 14                    ; Wait for the divide to complete
        BRU.JMP     i30           ; Return
        NOP <branch delay slots>
```

If a function accepts more than 8 scalar parameters[20], then scalar parameters from 9 and onwards are passed via the stack. Parameter 9 is assigned to the lowest address in memory, parameter 10 to the next lowest address, etc.

When parameters are passed via the stack, care must be taken to ensure that the size of each parameter is respected. For example, an ‘`int`’ parameter requires 4 bytes on the stack and is aligned on a 4-byte boundary; while a vector parameter requires 16 bytes and is aligned on an 8-byte boundary. For reasons of efficiency, the stack must be padded so that the address of a parameter falls on an address boundary which is aligned to the appropriate size of that parameter up to a maximum alignment of 8. For example, when passing an ‘`int`’ parameter, the stack should be padded to the nearest multiple of 4. Parameters of type ‘`bool`’, ‘`char`’, ‘`unsigned char`’, ‘`signed char`’, ‘`short`’ and ‘`unsigned short`’ are first promoted to ‘`int`’ by the ISO C arithmetic promotion rules, and it is an ‘`int`’ that is actually passed on the stack. Similarly ‘`float`’ and ‘`__fp16`’ are first promoted to ‘`double`’, and it is a ‘`double`’ that is actually passed on the stack. With `moviCompile`, since the representation of a ‘`float`’ and a ‘`double`’ are identical, this is purely a semantic change not an actual change.

Example:

Consider another version of the ‘`average`’ example:

```
      // Parameters s1 and s2 must be passed via the stack
      int average(int r1, int r2, int r3, int r4, int r5, int r6, int r7,
                  int r8, int s1, short s2);

; Assembly code

caller:
      ...                       ; Assume at the point of the call, that r1 is in
                                ; i1, r2 in i2 ... r8; in i8, s1 in i9 and s2
                                ; in i10.
      CMU.CPII    i18, i1       ; Copy parameters to their registers, as
                                ; defined by the calling convention
      CMU.CPII    i17, i2
      ...
      CMU.CPII    i11, i8
      IAU.SUB     i19, i19, 6   ; Allocate space on the stack for an int and
                                ; a short
      LSU0.ST.32  i9, i19       ; Store parameter s1 to stack
      LSU0.STO.16 i10, i19, 4   ; Store parameter s2 to stack + sizeof(s1)
      LSU0.LDIL   i30, average  ; Load the address of the function to be called
      LSU0.LDIH   i30, average
      BRU.SWP     i30           ; Call the function
      NOP         <branch delay slots>
      IAU.ADD     i19, i19, 6   ; Restore the stack. At this point i18 should
                                ; contain the value returned by ‘average’
     ..........

average:
      ; Accumulate sum of parameters into the return register.
      ; Meanwhile, load non-register parameters from stack into registers.
      IAU.ADD     i18, i18, i17 || LSU0.LD.32  i0, i19
      IAU.ADD     i18, i18, i16 || LSU0.LDO.16 i1, i19, 4
      IAU.ADD     i18, i18, i15
      IAU.ADD     i18, i18, i14
      IAU.ADD     i18, i18, i13
      IAU.ADD     i18, i18, i12
      IAU.ADD     i18, i18, i11
      IAU.ADD     i18, i18, i0
```

[20] Adjusted accordingly for 64-bit integers scalars

```
IAU.ADD      i18, i18, i1
LSU0.LDIL    i1, 11
SAU.DIV.i32 i18, i18, i1
NOP          14
BRU.JMP      i30           ; Return
NOP          <branch delay slots>
```

9.5.1 64-bit Integers

The 64-bit integer types ‘`long long`’ and ‘`unsigned long long`’ are too large to fit in a single 32-bit register, so instead these are passed in register “pairs”. These are always organized as a pair of consecutive IRF registers, e.g. `I4` and `I5`, or `I13` and `I14`. The low 32-bits of the 64-bit value are always placed in the register with the lower ordinal of the pair, and the high 32-bits of the 64-bit value are placed in the register with the higher ordinal of the pair.

These register pairs compete with the other scalars for register allocation on a first-come, first-served basis.

9.5.2 64-bit Floating-Point

While 64-bit floating-point scalars are now supported by `moviCompile`, this is achieved using emulation. An actual 64-bit Floating-Point value is represented as a pair of IRF registers. However, unlike 64-bit Integers, these are not paired as a 64-bit super-register, but rather as a simple pair of discrete 32-bit IRF registers with the low-32-bits of the 64-bit Floating-Point value being passed in the “High Order” IRF numbered register, and the high 32-bits of the 64-bit Floating-Point value being passed in the “Low-Order” IRF numbered register.

While this is counter-intuitive, it is a consequence of the register allocation ordering for discrete 32-bit values as function arguments which goes from IRF18 to IRF11. For a number of technical reasons, it is not currently possible to implement FP64 values using a 64-bit super-register as we have for 64-bit integers, though this may change in a future implementation which would change this aspect ABI.

9.5.3 C++ Member Functions

In most regards C++ and C are identical regarding how arguments are passed to functions, but for non-static member functions an additional hidden argument called the ‘`this`’ pointer is passed. For such functions, this is passed “as if” it was explicitly provided as the first argument to the function, and so all scalar arguments in the function declaration are passed as if they are the 2nd, 3rd and so on arguments instead of the 1st, 2nd and so on.

In addition, if the non-static member function returns a ‘`struct`’ or ‘`class`’ by value, then the address of that value is passed into the non-static member function using the register `I17` and not `I18` as would usually be the case because the ‘`this`’ pointer takes precedence and uses `I18`.

9.6 Vector Parameters

Up to 8 vector parameters may be passed via the `VRF`[21] using registers `v23`, `v22`, `...`, `v16`. The first (leftmost) vector parameter goes in register `v23`, the second in register `v22`, etc.

If a method accepts more than 8 vector parameters, then vector parameters from 9 and onwards must be passed via the stack. As with scalar parameters, the 9th vector parameter is assigned the lowest address in memory, the 10th vector parameter the next lowest address, etc.

Register `v23`, the first vector parameter register, is also used to store the return value for a function call if the return value is of a vector type.

[21] Even though `moviCompile` supports native 32-bit vectors in the IRF, the calling convention predates this support and 32-bit vectors are automatically widened to a 128-bit VRF when a function is called, and narrowed back to a 32-bit IRF on return. The more optimal use of a 32-bit IRF is not possible without breaking the ABI contract with existing code.

The stack must be padded so that the address of a vector parameter is aligned to a multiple of 8 bytes.

9.7 Matrix Parameters

Limited support is provided for matrix data types. These are described by the Vector Utility Library (Section 6.3) and are currently implemented as '`struct`' aggregates.

9.8 Non-POD Parameters

Parameters of basic types (`int`, `float`, `short`, `char`, pointer types, etc.) are known as "plain old data" types (`POD`), and are referred to as `POD`-parameters. See the ISO C and C++ Standards for the complete definition of a `POD`.

Example:

Given the following structure declaration:

```
struct S {
      int A[10];
      float x;
};
```

and the following function prototype:

```
S f(int p1, S p2, float p3, float4 p4, S* p5);
```

parameters `p1`, `p3`, `p4` and `p5` are `POD`, `p2` is `non-POD`, and the return type is `non-POD`.

Note in particular that parameter `p5` is a `POD` because it is simply a pointer to a value of type `S`, and thus can be stored in an integer register. Also, parameter `p4` is regarded as a `POD` because `SHAVE` uses vector registers for these (in a traditional architecture vectors would be regarded as a `non-POD` parameter). `moviCompile` implements all vector types up to size 16 Bytes as a `POD`.

In conclusion, the calling convention for `non-POD` types:

1. `Non-POD` parameters of type '`T`' where '`sizeof(T) <= 4`' are passed via integer registers, as if they were integer parameters, as long as one of the 8 integer parameter registers is available. Similarly, if a function returns a `non-POD` type where '`sizeof(T) <= 4`' this is returned via integer return register `i18`.
2. Exception: a `struct`, `class` or `union` containing a single `float` or `half` is passed as a parameter/returned from a method as if it were a `POD float`.
3. All other `Non-POD` parameters and `Non-POD` return values are passed via the stack.

NOTE: If a `non-POD` type is returned via the stack then the address of the space allocated for the return value (by the caller) is passed to the callee as an implicit additional parameter, inserted at the beginning of the parameter list, e.g. in register `i18`. All subsequent scalar parameters are passed in registers `i17` downwards.

9.9 Variable Number of Arguments

A `C/C++` method can be defined to accept a variable number of arguments by specifying a number of required parameters, followed by '...'. The classic example of this is `printf`, which has the signature:

```
int printf(const char*, ...);
```

All parameters to a function accepting a variable number of arguments (including the explicit parameters) are passed via the stack (see also section 11.3.7).

9.10 C++ Name Mangling

For C++ programs to be able to use overloaded functions, the assembly label associated with a C++ function or method needs to be mangled (or decorated) so that it is distinct from labels associated with different functions which share the same name but which have different parameter types. This is essential to implementing ISO C++'s requirement for "Type Safe Linkage" as well as overloading[22].

The mangled name of a function is derived from the function's declared name and its arguments (its signature). Name mangling schemes are part of the calling convention because, when calling a C++ function defined in a different file, it is necessary to know how that function's name is mangled.

`moviCompile` mangles `C++` names using the `GCC` name mangling scheme. For example, consider these two function declarations:

```
void myFunc(int f, volatile float* x);
void myFunc(short t);
```

According to the `GCC` scheme, the first function gets the following mangled name:

```
_Z6myFunciPVf
```

This mangled naming scheme is not described in this document, but in essence the `iPVf` portion of the name says 'this function takes an `int (i)` and a pointer `(P)` to a `volatile (V) float (f)` as parameters'.

The second function gets this mangled name:

```
_Z6myFuncs
```

since the function takes a `short` (`s`) parameter.

When integrating C++ programs with assembly routines it can be difficult to work explicitly with mangled names. To make things easier, the user can define a function as '`extern "C"`' to tell the compiler the function's name should be mangled according to the C rules.

```
extern "C" void myFunc(int f, volatile float* x);
```

This method now gets the C-style name:

```
myFunc
```

which is easier to work with from an assembly routine, and for inter-operating with C.

NOTE: You cannot use '`extern "C"`' and also overload a function.

9.10.1 De-Mangling

It is often useful to determine what the original C++ name for a mangled name is. Since the mangling scheme is the same as that used by `GCC`, the Linux and Cygwin command-line utility '`c++filt`' may be used for this purpose.

The Movidius tools also come with a pre-built version of this utility called `sparc-myriad-rtems-c++filt`.

You will need to add an extra leading underscore to the name to use '`c++filt`', so the function named '`_Z6myFunciPVf`' can be de-mangled as follows:

```
c++filt __Z6myFunciPVf
```

[22] C++ names are always mangled, unless they are declared with '`extern "C"`'

And it will display:

```
        myFunc(int, float volatile*)
```

9.11 Formalized Leon/SHAVE Entry-Points

When a program running on the Leon needs to start a function on the SHAVE, it does so using an "Entry-Point". Entry-Points have specialized ABI requirements to ensure that communication between the Leon and the SHAVE is valid and as efficient as possible.

To facilitate this, moviCompile provides a special function ABI using the attribute:

```
        __attribute__((dllexport))
```

Some non-SHAVE targets (notably X86) support the attribute 'dllexport', and moviCompile "borrows" this attribute to implement functions[23] with a specialized calling convention for use as "Entry-Points" which are SHAVE functions specially adapted to be started from the Leon executive program.

This ABI also supports the construction of "Indirect Address Tables" (IAT) which are conceptually similar to the structure of the same name used to implement DLLs (Dynamic Loaded Libraries) for Microsoft Windows, and although dynamic loading is not currently supported by the SHAVE programming environment, the use of indirect access tables provides the ability to find Entry-Points at runtime by name, and to dispatch them without having to explicitly declare them symbolically in the code that starts them[24].

9.11.1 How Does This Change The Calling Convention?

A normal SHAVE callable function has to save registers that are altered during the execution of that function, but which the calling convention requires must be preserved on return from the function. Also, a normal SHAVE function will assume that the IRF register 'I30' (the Link-Register) contains the return address to the SHAVE function that called it, and will "return" from the function by issuing the instruction:

```
        BRU.JMP I30
```

When the function is defined using the attribute 'dllexport', it is intended to be used as an Entry-Point that is started by the Leon and not to be called by SHAVE code. The compiler will minimize the number of registers that it saves at the start of the function (the function prologue) resulting in a very simple light-weight sequence of instructions before the main body of the function executes.

And when the function "returns", it will restore very few if any registers as the Leon does not need them, and will also generate the 'BRU.SWIH' instruction for the return since it is relinquishing control to the Leon and not to a location in the SHAVE code.

The combination of the prologue and epilogue changes are such that the overhead to these 'dllexport' functions is a lot lower than for a regular function. But they are also not callable from SHAVE code.

If the function takes arguments, the Leon can set up the argument registers as per the normal calling convention; and if it returns a value, the usual calling convention registers are set in the usual way. For example, given:

```
        __attribute__((dllexport)) int foo() {
            return 42;
        }
```

[23] At this time moviCompile only supports 'dllexport' on functions and not on data objects

[24] These will generally be started from the Leon as SHAVE "Entry-Points"

Then when ‘foo’ is compiled as a normal SHAVE function, it produces the following code:

```
.code       .text.foo
.L.text.foo:
    .align      16
    .type       foo,@function

    foo:
        BRU.JMP i30
        NOP 5
        LSU0.LDIL i18 0x002a
.Lfunc_end0:
    .size       foo, .-foo
```

and with the ‘dllexport’ attribute it generates:

```
.code       .text.foo
.L.text.foo:
    .align      16
    .type    foo,@function

    foo:
        BRU.SWIH 0x00 ; Entry-Point Return
            || LSU0.LDIL i18 0x002a
.Lfunc_end0:
    .size    foo, .-foo
```

With more elaborate examples, you would see a more significant difference, in particular regarding stack maintenance, and register saving and restoring.

9.11.1.1 Additional Input Register Requirements

The calling convention for a ‘dllexport’ function accepts input parameters and returns values in exactly the same way as ordinary SHAVE functions; but it does not preserve any registers as it would for ordinary SHAVE function calls.

It also requires that three “Special Registers” are set-up on invocation by the Leon. These are:

- IRF ‘I19’ –the SP or Stack Pointer. This is automatically maintained when an ordinary SHAVE function is called by another, but when an Entry-Point is started from the Leon, it is the responsibility of the Leon to initialize this to point to the location of the stack to be used for the invocation. This has always been the case for Myriad application development, and this initialization is hidden by the Myriad software development framework. See section 10 for more details about program execution flow.
- IRF ‘I20’ – this register must be initialized to the same value as used for initializing the SP (‘I19’), but minus the number of Bytes available in the stack. This is necessary to facilitate the instrumentation of the stack to detect stack overflow and to monitor stack usage (see section 10.5.3). Neither moviCompile nor its libraries are aware of where the stack is located nor how large it is. By providing this information via the ‘dllexport’ calling convention, it is possible for moviCompile to instrument the stack using this additional information.
- IRF ‘I21’ –the Myriad software development framework – supported by moviCompile’s C Run-Time (CRT) Library – implements support for dynamic “Execution Context” information shared across multiple Entry-Point invocations. This register should be initialized with the pointer to this context information, or to NULL if no execution context is applicable.

On entry to the ‘dllexport’ functions, the compiler will insert a small number of instructions to capture the values provided in the three registers ‘I19’, ‘I20’ and ‘I21’ for later use. This “execution context capture” will not happen for regular SHAVE functions.

However, legacy programs that use ordinary SHAVE functions as Entry-Points, supplemented by explicit use of 'BRU.SWIH' should transition to defining these functions using '__attribute__((dllexport))' instead, and replace the existing explicit use of 'BRU.SWIH' with a normal C return statement. For example, replace the following:

```
void myEntryPoint(<arg-list>) {
  ...
  SHAVE_HALT;
}
```

with:

```
__attribute__((dllexport))
void myEntryPoint(<arg-list>) {
  ...
  return;
}
```

Alternatively, the program can include the moviCompile system header file '<sys/__moviconfig.h>' which defines the macro '_MV_ENTRYPOINT_DEFN' for this attribute, so the code could be written as follows:

```
#include <sys/__moviconfig.h>
...
_MV_ENTRYPOINT_DEFN
void myEntryPoint(<arg-list>) {
  ...
  return;
}
```

NOTE: The function 'main' is special to C and to C++ and should never be defined using 'dllexport', nor should it be used as an Entry-Point.

9.11.2 What is The IAT?

The IAT consists of an ordered matching pair of arrays, one which contains the addresses of each of the functions defined using the 'dllexport' attribute, and the other contains the addresses of NUL-terminated 'const char*' strings that contains the names of those functions. The ordinal positions are guaranteed to be the same, so if the name of a particular function is found at index *N* in the "names" array, then the address of that function is contained in the function "address" array (the IAT itself) at the same ordinal index *N*.

The "names" array is terminated with a NULL pointer to indicate that there are no more entries in the IAT.

These structures are automatically generated by the compiler when code is being generated for each function defined with the 'dllexport' attribute, and they are placed in specially named, and implementation reserved sections. For example, the following function:

```
__attribute__((dllexport)) int foo() {
    return 42;
}
```

will result in the following three IAT support sections:

- '..iat.foo' - which contains the address of the function
- '..iatnames.foo' - which contains a pointer to a string containing the name of the function
- '..str..iatstrings' - which contains the NUL terminated string with the name of the function 'foo'

Since none of these sections are critical to programs that use the traditional dispatch mechanism for calling `SHAVE` Entry-Points, these sections can be safely discarded during linking for a backward compatible solution; however they need to be retained during final linking of the complete program if the programmer intends to dispatch `SHAVE` Entry-Points using this mechanism.

The supplementary information for IAT support for this function is as follows:

```
.data       ..str..iatstrings
    .L.iatname_0:
    .string     "foo"

..iat.foo
    .align      4
    .word       foo

.data       ..iatnames.foo
    .align      4
    .word       .L.iatname_0

.data       .gnu.linkonce..iatnamesend
    .linkonce
    .align      4
    .word       0
```

NOTE: It is recommended that these additional data structures for supporting IATs are placed in DDR. They are not intended to be used from the `SHAVE` code, and there is no advantage to storing them in CMX.

9.11.3 Retention of Symbols

Even if the programmer is not using the IAT support capability of the '`dllexport`' functions, functions which are defined using this attribute will be retained during the garbage collection phase of linking ('`--gc-sections`'), even if there are no explicit references to them. This means that it is not necessary to use the '`-u`' option for these functions which reduces the complexity of the link command-line.

This is specially intended to facilitate the construction of programs with multiple Entry-Points to be dispatched from the `Leon`.

It is not necessary to retain the IAT support data structures during the final link phase to ensure that these symbols are retained.

9.12 Summary of Calling Convention

9.12.1 Register Allocation

Table 9 below summarizes the registers used by the calling convention (CC) and the registers which need to be preserved across function calls in order to conform to the calling convention.

When interfacing C and handwritten assembly functions, the programmer is free to use all of the registers. However, they must ensure that the SP, LR and all of the preserved registers have the same values at the return of the handwritten assembly function as they had when the function was called. If the assembly code is calling a function written in C, then it has to assume that the registers identified as parameters, return value or "Not preserved" will be altered by the called function, and save them if their values are still required following the call.

The recommended way of doing this is by pushing any preserved registers that the programmer intends to use inside the function onto the stack as the first thing done inside the function, and popping them off right before returning from the function.

No.	IRF – Normal CC	VRF – Normal CC	IRF – DllExport CC	VRF – DllExport CC
0	Not preserved	Not preserved	Not preserved	Not preserved
1	Not preserved	Not preserved	Not preserved	Not preserved
2	Not preserved	Not preserved	Not preserved	Not preserved
3	Not preserved	Not preserved	Not preserved	Not preserved
4	Not preserved	Not preserved	Not preserved	Not preserved
5	Not preserved	Not preserved	Not preserved	Not preserved
6	Not preserved	Not preserved	Not preserved	Not preserved
7	Not preserved	Not preserved	Not preserved	Not preserved
8	Not preserved	Not preserved	Not preserved	Not preserved
9	Not preserved	Not preserved	Not preserved	Not preserved
10	Not preserved	Not preserved	Not preserved	Not preserved
11	Scalar Param 7	Not preserved	Scalar Param 7	Not preserved
12	Scalar Param 6	Not preserved	Scalar Param 6	Not preserved
13	Scalar Param 5	Not preserved	Scalar Param 5	Not preserved
14	Scalar Param 4	Not preserved	Scalar Param 4	Not preserved
15	Scalar Param 3	Not preserved	Scalar Param 3	Not preserved
16	Scalar Param 2	Vector Param 7	Scalar Param 2	Vector Param 7
17	Scalar Param 1/Return[25]	Vector Param 6	Scalar Param 1/Return[26]	Vector Param 6
18	Scalar Param 0/Return	Vector Param 5	Scalar Param 0/Return	Vector Param 5
19	**Stack Pointer (SP)**	Vector Param 4	**Stack Pointer (SP)**	Vector Param 4
20	**Preserved**	Vector Param 3	**Stack Top Address**	Vector Param 3
21	**Preserved**	Vector Param 2	**Context Pointer**	Vector Param 2
22	**Preserved**	Vector Param 1	Not preserved	Vector Param 1
23	**Preserved**	Vector Param 0/Return	Not preserved	Vector Param 0/Return
24	**Preserved**	**Preserved**	Not preserved	Not preserved
25	**Preserved**	**Preserved**	Not preserved	Not preserved
26	**Preserved**	**Preserved**	Not preserved	Not preserved
27	**Preserved**	**Preserved**	Not preserved	Not preserved
28	**Preserved**	**Preserved**	Not preserved	Not preserved
29	**Preserved**	**Preserved**	Not preserved	Not preserved
30	**Link Register (LR)**	**Preserved**	Not preserved	Not preserved
31	**Preserved**	**Preserved**	Not preserved	Not preserved

Table 9. `moviCompile` Calling Convention

NOTE: For the 64-bit integer scalars, the arguments are allocated as IRF register pairs:

(`I17,I18`), (`I16,I17`), (`I15,I16`) … (`I11,I12`)

and are always returned in the register pair (`I17,I18`)

The entries in Table 9 have been color coded to differentiate their purpose and how they are treated. The following legend explains each of these:

Color Coding	Explanation
Not preserved	No expected input value. Contents are not preserved on return
Preserved	No expected input value. Contents are unchanged on return
Param	Input parameter. Contents are not preserved on return
Param/Return	Input parameter or return value from function, otherwise not preserved on return

[25] If the value is a 64-bit integer
[26] If the value is a 64-bit integer

Color Coding	Explanation
Special	These have special meaning to the program described in the Calling-Convention

Table 10. Calling Convention Color Legend

9.12.2 Parameter Passing and Return for `varargs` Functions

All parameters to a function with `variadic` arguments are passed via the stack.

The return value for a `variadic` function is handled as for `non-variadic` functions.

10 Runtime Execution Models

This section describes the several phases involved in preparing a program for execution, and then carrying out that execution. On a hosted system such as Android, Windows or Linux, the host-platform takes care of many of the details required to place a program in memory, initialize its execution context, and then run the program you wrote.

However, on an embedded platform such as the Myriad VPU, the programmer needs to be aware of these details to properly ensure that their program runs as expected. Most of these details are managed by the Movidius MDK, but it is still useful to understand what is going on.

This is particularly important when running C++ programs, as the execution context for a C++ program is a lot more complex than that of a C program.

10.1 The C Program Execution Model

10.1.1 How Does a Conventional C Program Execute?

On a convention hosted platform, several things happen when a C program is loaded into memory and executed. I will skip the memory loading phase as it is not directly relevant to this section, but the execution context is very important.

Typically the program will start with some pre-execution steps commonly referred to as the '`crtinit`' phase, meaning "C Run-Time Initialization". This phase will:

1. Initialize the stack pointer
2. Zero the COMMON or BSS memory regions
3. Initialize the heap
4. Call the function '`main`'
5. When '`main`' returns, it will call the function '`exit`' with the return value from '`main`' as the argument for '`exit`'
6. The function '`exit`' will execute any routines registered using the '`atexit`' function in a Last-In/First-Out fashion (a stack)
7. The function '`exit`' will then call the function '`_Exit`' with the same argument value it received
8. The function '`_Exit`' ends the execution, and reports the exit code it received to the host

10.1.2 How Does a `SHAVE` C Program Execute? The Task Programming Model

The Task Programming Model is a very useful abstraction, and it allows the executive program running on the `Leon` to decide when to run tasks on the `SHAVE`; but it does depart from the conventional execution model. This has more significant implications when executing C++ programs which have more complex execution requirements (see section 10.2). Also, since finalization is not automatic, functions registered with '`atexit`' will not be run and finalization will have to be explicitly managed by the programmer.

1. Initialise the stack pointer (IRF I19)
    a. this is done by the `Leon`
2. Zero the COMMON or BSS memory regions
    a. this is done by the `Leon`

3. Initialise the heap
    a. this is defaulted in the 'mlibc.a' and 'mlibc_lite.a' libraries to 1Mbyte beginning at the address 0x1F000000 – this is the WIN_D windowed address space
    b. the programmer can explicitly initialize the heap to another location and size of their choosing as follows:
        i. by calling '__setheap' from their SHAVE code prior to any use of the heap ('malloc' or 'free')
        ii. by starting the '_EP_setheap' Entry-Point from the Leon program prior to starting any other Entry-Points. This is used in the same way as the SHAVE callable function '__setheap', but instead of returning to the SHAVE caller, it is implemented using the 'dllexport' attribute which will yield control back to the Leon using the 'BRU.SWIH' instruction
        iii. the heap reservation can be passed in the execution context to the Entry-Points '_EP_start' and '_EP_crtinit' – this is the preferred approach
4. Call the function 'main'
    a. this does not happen[27].

       Instead the Leon program sequences the execution a series of Entry-Points on the SHAVE to carry out the tasks it requires. Each task concludes by using the 'BRU.SWIH' instruction to yield control back to the Leon
5. When 'main' returns, it will call the function 'exit' with the return value from 'main' as the argument for 'exit'
    a. this does not happen since 'main' is not automatically called
6. The function 'exit' will execute any routines registered using the 'atexit' function in a Last-In/First-Out fashions (a stack)
    a. this does not happen automatically since 'main' is not automatically called. It is also not recommended that the programmer calls 'exit' from their code if they are using the Task Programming Model, as this will "finalize" the execution context for the program, and there is no guarantee that the execution context is valid after finalization
7. The function 'exit' will then call the function '_Exit' with the same argument value it received
    a. this does not happen automatically since 'exit' is not automatically called
8. The function '_Exit' ends the execution, and reports the exit code it received to the host.
    a. this does not happen, although a typical legacy[28] SHAVE Entry-Point function will use 'BRU.SWIH' directly using 'SHAVE_HALT', or calls '_Exit' directly which will execute the instruction 'BRU.SWIH', or by returning from a function defined with the 'dllexport' attribute

[27] Although programmers do sometimes have a function called 'main' which is invoked as an Entry-Point, it is not recommended as 'main' has special meaning to ISO C and ISO C++. The programmer should rename such Entry-Points to some other non-ISO reserved name
[28] These should be fixed to use the 'dllexport' ABI described in section 9.11

10.1.2.1 Execution Context – Initialization and Finalization

To support initialization and finalization for the Task Programming Model, the `SHAVE` runtime libraries provide two important Entry-Points:

```
int32_t _EP_crtinit(void)
```

This is a special Entry-Point that ensures that code critical to establishing the execution context has been run prior to starting the sequence of Entry-Points representing the program itself. For C this will generally perform no additional operations, but see also section 10.2.2.1.1.

This special Entry-Point should be started ONCE, and before any other Entry-Points are started. On completion, this will yield control back to the `Leon` by issuing a '`BRU.SWIH`' instruction, and will return the value zero unless some error occurred during initialization.

```
int32_t _EP_crtfini(int)
```

This is another special Entry-Point, and should be started ONCE, and only after all other Entry-Points have been started and completed. As with the special Entry-Point '`_EP_crtinit`', this will yield control to the `Leon` using '`BRU.SWIH`' upon completion.

This Entry-Point will perform all the finalization operations for the execution context, including execution of any functions registered with '`atexit`'. The value passed to this Entry-Point will be passed to the function '`exit`' which performs the actual finalization operations.

After '`_EP_crtfini`' has been invoked, the execution context on the `SHAVE` should no longer be considered valid, so a new execution context needs to be established before the `SHAVE` is used again.

10.1.3 How Does a `SHAVE` C Program Execute? The Standard Model

The `SHAVE` runtime libraries provide an alternative method for running programs on the `SHAVE` that better follows the conventional hosted program execution model. In order to use the `SHAVE` in this way, the `Leon` program need only start one special Entry-Point called '`_EP_start`'.

```
int32_t _EP_start(int32_t, const char*[])
```

This Entry-Point allows the Standard model to be more easily accommodated as follows:

1. Initialise the stack pointer (IRF I19)
   a. this is done by the `Leon`
2. Zero the COMMON or BSS memory regions
   a. this is done by the `Leon`
3. Initialise the heap
   a. this is defaulted in the '`mlibc.a`' and '`mlibc_lite.a`' libraries to 1Mbyte beginning at the address `0x1F000000` – this is the `WIN_D` windowed address space
   b. the programmer can explicitly initialize the heap to another location and size of their choosing as follows:
      i. by calling '`__setheap`' from their `SHAVE` code prior to any use of the heap ('`malloc`' or '`free`')
      ii. by starting the '`_EP_setheap`' Entry-Point from the `Leon` program prior to starting any other Entry-Points. This is used in the same way as the `SHAVE` callable function '`__setheap`', but instead of returning to the `SHAVE` caller. it is implemented using the '`dllexport`' attribute which will yield control back to the `Leon` using the '`BRU.SWIH`' instruction

iii. the heap reservation can be passed in the execution context to the Entry-Points ‘`_EP_start`’ – this is the preferred approach

4. Call the function ‘`main`’
   a. the arguments provided to this Entry-Point are passed on as the arguments ‘`argc`’ and ‘`argv`’ to ‘`main`’
5. When ‘`main`’ returns, it will call the function ‘`exit`’ with the return value from ‘`main`’ as the argument for ‘`exit`’
   a. since ‘`main`’ is automatically called, the value it returns is automatically passed to ‘`exit`’
6. The function ‘`exit`’ will execute any routines registered using the ‘`atexit`’ function in a Last-In/First-Out fashions (a stack)
   a. this will perform the finalization tasks including the invocation of any functions registered with ‘`atexit`’
7. The function ‘`exit`’ will then call the function ‘`_Exit`’ with the same argument value it received
   a. when ‘`exit`’ completes, it automatically calls ‘`_Exit`’
8. The function ‘`_Exit`’ ends the execution, and reports the exit code it received to the host.
   a. this will yield control to the `Leon` by executing the instruction ‘`BRU.SWIH`’

So in summary, this method of invocation provides the closest equivalent to the conventional program execution model.

10.2 The C++ Execution Model

10.2.1 How Does a Conventional C++ Program Execute?

Since C++ has evolved from C, there are many elements in common, but there are some notable differences.

On a conventional hosted platform, several things happen when a C++ program is loaded into memory and executed. Again I will skip the memory loading phase as it is not directly relevant to this document.

As for C, the program will start with the ‘`crtinit`’ phase, though this phase is somewhat more complex and more important than for C. For C++ this phase will:

1. Initialise the stack pointer
2. Zero the COMMON or BSS memory regions
3. Initialise the heap
4. [`crtinit`] Call the constructors for all global objects which have a constructor; for example:

   ```
   struct X { X(); } anX;
   int main() { return 0; }
   ```

   Even though nothing else appears to be present, the constructor ‘`X::X()`’ will be called to initialize ‘`anX`’ before ‘`main`’ executes.

   It will also initialize all other objects whose initializers are not statically determined; for example:

   ```
   extern int bar();
   int myVar = bar();

   int main() { return 0; }
   ```

This initialization of ‘myVar’ is not legal in C, but it is valid in C++. Just as with constructors, this will be dynamically initialized “before” the function ‘main’ is executed by calling the function ‘bar’.

5. Call the function ‘main’
6. When ‘main’ returns, it will call the function ‘exit’ with the return value from ‘main’ as the argument for ‘exit’
7. The function ‘exit’ will execute any routines registered using the ‘atexit’ function in a Last-In/First-Out fashions (a stack), AND the destructors associated with any objects with “static extent (see section 10.2.2)” which have a destructor; for example:

   ```
   struct X {
     X();
     ~X();
   } anX;

   int main() { return 0; }
   ```

   Since the object ‘anX’ has an associated destructor, it must be called during finalization.

   As it happens, the mechanism for ‘atexit’ is reused for these destructors, and the order of execution of functions registered with ‘atexit’ and objects which need to have their destructors invoked are interleaved. An object will “end its lifetime” (i.e. have its destructor called) in the inverse order of its initialization with respect to other objects with destructors; for example:

   ```
   struct X {
     X();
     ~X();
   };

   X anX_1;
   X anX_2;

   int main() { return 0; }
   ```

   In this case the constructor for ‘anX_1’ will happen first, followed by the constructor for ‘anX_2’, then ‘main’ will execute, followed by the destructor for ‘anX_2’ and followed by the destructor for ‘anX_1’.
8. The function ‘exit’ will then call the function ‘_Exit’ with the same argument value it received
9. The function ‘_Exit’ ends the execution, and reports the exit code it received to the host.

This is almost identical to the execution flow for a conventional C program, largely differing in regard to the additional requirement for dynamic initialization of objects prior to the execution of ‘main’, and the additional work to be performed during finalization.

10.2.2 So, What is ‘Static Extent’?

It is easy to think of “static extent” as being the same thing as “static objects”, but there are subtle and important differences. Consider the following example:

```
extern int foo();

struct T {
  static int value;
  T();
```

```
    ~T();
    void function1 ();
    void function2 ();
};

T globalExtern;
static T globalStatic;

namespace {
    T anonymous;
}

T& getSingleton () {
    static T localStatic;
    return localStaticT;
}

int T::staticMember = foo();

int main() {
    getSingleton().function1();
    getSingleton().function2();
    return 0;
}
```

All of the objects declared here have "static extent", but only "`globalStatic`", "`localStatic`" and "`anonymous`" are "static objects". Static extent means that the storage for the object is determined and reserved prior to program execution, and does not reside on the heap or stack.

10.2.2.1 Programming Implications

So what does this mean to programming for the Myriad platform?

The most important elements are that "initialization" and "finalization" MUST happen. These can be done in exactly the same way for C and C++ programs, so while this section discusses C++, it applies equally to C.

10.2.2.1.1 Initialization – and special attention to the Heap

This is a very important aspect to be aware of. A constructor or an initializer for a static extent object may use the heap, so it is VITAL that the heap has been initialized before this happens.

This has great significance for programs which currently call '`__setheap`' from '`main`' or from any Entry-Point – because it may already be too late by the time this happens.

Furthermore, the heap cannot be changed after it has started being used, or a program crash is very likely to happen when objects which have already been allocated are released; for example:

```
char* myBuffer = new char[20];

int main () {
    __setheap((void*)0x12345678, 2048);
    delete[] myBuffer;
    myBuffer = 0;
    return 0;
}
```

This WILL break, because the heap has been moved between the time the memory for ‘myBuffer’ was allocated and when it is deallocated.

The model of execution is essentially as follows:

- Execute the SHAVE initialization phase (crtinit phase)
- Execute ‘main’, or each of the task Entry-Points until the purpose of the program is complete
- Execute the SHAVE finalization phase (crtfini phase)

But where can the heap be custom set so as to avoid the problem described in the previous example?

One approach is to ensure that the Leon starts the ‘_EP_setheap’ Entry-Point prior to starting ‘_EP_start’, ‘_EP_crtinit’ or any other Entry-Point.

An alternative and more flexible approach would be to place a call to ‘__setheap’ inside a “prioritized constructor” (see section 10.5.1 for more details about this mechanism), so in this example it might be achieved as follows:

```
__attribute__((constructor(200))) void initMyHeap(void) {
    __setheap(addressOfMyHeap, myHeapSize);
}
```

But the preferred and safest approach is to provide the heap initialisation requirements in the Execution Context passed to the ‘_EP_crtinit’ and ‘_EP_start’ Entry-Points. This is the recommended mechanism for ensuring that the heap is correctly initialised.

You might wonder why you care about initialization and finalization; can’t you just tell people not to do dynamic initialization involving the heap? But there is no way of policing this, and there are hidden places that you might not think of. For example:

```
#include <iostream>

int main() {
    std::cout << “Hello” << std::endl;
}
```

This innocent looking program introduces three global static extent objects that you know very well – ‘std::cout’, ‘std::cin’ and ‘std::cerr’. Each of these is initialized using a constructor, and each internally allocates other objects and buffers on the heap.

10.2.2.1.2 Finalization

When all of the Entry-Points representing the program have completed, the objects that have dynamic termination requirements must have their finalization code executed. In the conventional C or C++ execution model, this happens normally when ‘main’ returns, or when ‘exit’ is called.

In the previous example, the three Standard objects also have destructors that need to be run upon completion of the program, so the finalization cannot be omitted either; ‘std::cout’ in particular is typically buffered, and the final flushing of that buffer may not take place if its destructor is not executed.

10.3 Execution Flow – Summary

When the standard programming model is chosen and the Entry-Point ‘_EP_start’ is used, the conventional execution flow for both C and C++ occurs. The Leon is responsible for initializing the stack-pointer and clearing the BSS/COMMON regions of memory; but the remainder of the program's execution context is automatic:

- the CRT initialization phase is run, which will run the constructors
- '`main`' is called
- the CRT finalization phase is run, which will run the destructors and functions registered with '`atexit`'.

For the Task Programming Model the programmer needs to take a little more care and ensure that they perform the following duties:

- optionally initialize the heap directly by starting the Entry-Point '`_EP_setheap`'
- explicitly run the CRT initialization phase by starting '`_EP_crtinit`' before invoking any other Entry-Points
- start each of the Entry-Points associated with the tasks for their program
- explicitly run the CRT finalization phase by starting '`_EP_crtfini`' after all other Entry-Points have completed

10.4 Prioritized Constructors and Destructors

`moviCompile` supports the attributes '`constructor`', '`constructor(`*number*`)`', '`destructor`' and '`destructor(`*number*`)`'.

Despite the names of these attributes, they can be used in C programs. All functions defined with these attributes must have the type:

```
void myFunction(void)
```

10.4.1 Initialization

Each function defined with '`__attribute__((constructor))`' will be executed during the same phase as the actual constructors and dynamic initializers for C++ objects with static extent (see section 10.2.2), but there is no specified relative ordering for these, so simple things like the order of symbol resolution during linking could change the actual order of execution.

To grant the programmer greater control over the order of these initialization operations, a function can be defined instead with the "prioritized" variant '`__attribute__((constructor(`*number*`)))`' where '*number*' is a decimal value between 0 and 65535.

The smaller the number, the higher its priority, and the higher a constructor's priority the earlier it will execute. So functions that have for example, the prioritized constructor number 200, will execute before those with the number 205.

The values 0 through 100 are reserved to the implementation, and their use by the programmer could result in undefined behavior.

Prioritized constructors with the same number are executed after those with a higher priority number and before those with a lower priority number, but the relative ordering of execution of prioritized constructors with the same number is unspecified.

All prioritized constructors are executed before functions defined with the more general '`__attribute__((constructor))`' and before all compiler generated constructor calls and initializers for static extent objects.

10.4.2 Finalization

Each function defined with '`__attribute__((destructor))`' will be executed during the program finalization phase, and during the same phase as the destructors for C++ static extent objects requiring dynamic finalization and functions registered with '`atexit`'. However, there is no specified relative ordering for these with respect to execution of the C++ object destructors and functions registered with '`atexit`'.

Just as with the constructors, the programmer can have greater control over the ordering of the destructors by using the prioritized variant '`__attribute__((destructor(`*`number`*`)))`' where '*`number`*' is a decimal value between 0 and 65535.

The smaller the number, the higher its priority, and the higher a destructor's priority the <u>later</u> it will execute. So functions that have for example, the prioritized destructor number 200, will execute <u>after</u> those with the number 205.

The values 0 through 100 are reserved to the implementation, and their use by the programmer could result in undefined behavior.

Prioritized destructors with the same number are executed before those with a higher priority number and after those with a lower priority number, but the relative ordering of execution of prioritized destructors with the same number is unspecified.

All prioritized destructors are executed after functions defined with the more general '`__attribute__((destructor))`' and after all destructors for static extent objects.

10.5 Dynamic Loading and Execution

To facilitate an alternative and flexible model of execution, the C Runtime [CRT] Entry-Points provide support for another execution model sometimes referred to as the Dynamic Loading model.

With this model there is a delineation between the application and the runtime support libraries used by the application. This allows for greater sharing of code and data between multiple applications than can be easily achieved with the previous execution models.

With this model, an additional parameter is passed to the CRT provided Entry-Points that provides the information necessary to support the Dynamic Loading model. This parameter is passed to '`dllexport`' Entry-Point functions in IRF '`I21`' and is described in section 9.11.

If this value is the `NULL` pointer, then the behavior of the CRT libraries is as previously described. However, this may also be the address of an object of type '`_ExecutionContext_t`' which is defined in the header file:

```
        sys/shave_system.h
```

When the execution context pointer is not `NULL`, then the CRT entry-points will use this information present in the execution context to perform a different series of initialization and finalization actions, and these differences are described in the following sections.

10.5.1 Dynamic Loading and C-Runtime Initialization

When the Entry-Points '`_EP_start`' or '`_EP_crtinit`' are started, or when the function '`__crtinit`' is called, the CRT performs the program initialization tasks as previously described. This is unchanged if a `NULL` execution context pointer is passed to most recently started Entry-Point.

The execution context pointer provided by an Entry-Point persists until the next time an Entry-Point is started with a new execution context pointer.

This can be retrieved at any time during the execution of the `SHAVE` program by calling the function '`__get_executioncontext`' – see section 10.5.3.

But when a non-NULL execution context is provided to the CRT by using these Entry-Points, or another Entry-Point followed by a call to '__crtinit', then the sequence of initialization is as follows:

- If the '_ExecutionContext_t' object specifies a location and size for the heap, then the function '__setheap' is called automatically with this information. This information is specified using the members 'heap_address' and 'heap_size'.
- The constructors and prioritized constructors for all objects in the library are executed as normal.
- If the '_ExecutionContext_t' object provides non-NULL values for the members 'ctors_start' and 'ctors_end', then the CRT will iterate through this array, executing each of the application constructors and prioritized constructors.

This 2-phase initialization model is essential for the correct operation of the Dynamic Loading execution model.

The execution context support for heap initialization provides a safer approach than requiring that the programmer calls '_EP_setheap' prior to execution and if the preferred mechanism for initializing the heap.

NOTE: The member 'stack_size' is not currently used by the CRT initialization routines.

10.5.2 Dynamic Loading and C Runtime Finalization

When the SHAVE program exits (return from 'main' or by calling 'exit'), or when the Entry-Point '_EP_crtfini' is started, or when the function '__crtfini' is called, then the CRT will perform the program finalization tasks as previously described. This is unchanged if a NULL execution context pointer is passed to the most recently started entry-point.

But when a non-NULL execution context is provided to the CRT, then the sequence of finalization is as follows:

- If the '_ExecutionContext_t' object provides non-NULL values for the members 'dtors_start' and 'dtors_end', then the CRT will iterate through this array, executing each of the application destructors and prioritized destructors.
- The destructors and prioritized destructors for all objects in the library are executed as normal.

Thus the 2-phase finalization follows the inverse sequence to the 2-phase initialization process.

NOTE: Interaction with 'atexit' and destruction of static extent objects are executed in the inverse order to the corresponding call to 'atexit' or the construction of the static extent objects. This is essential to ensure the correct semantic lifetime management as required by the ISO C++ Standards.

10.5.3 Retrieving the Execution Context

Although the CRT usage of the execution context is implicit, the SHAVE program may need to interact with the execution context directly, and the value passed in IRF 'I21' by the most recent invocation of an Entry-Point can be retrieved at any-time by calling the function:

```
const _ExecutionContext_t* __get_executioncontext(void)
```

which is declared in the header file:

```
sys/shave_system.h
```

10.6 Stack Instrumentation

When the Leon starts an Entry-Point on the SHAVE (see section 9.11.1) the location of the stack to be used is passed in register IRF 'I19', and the address of the maximum extent for the stack is passed in IRF 'I20'.

Because the SHAVE stack grows toward zero, the maximum extent of the stack is the lowest address that may be used for the stack, and is exactly equal to the address of the stack minus the number of Bytes it is permitted to use.

There are two kinds of stack instrumentation provided by `moviCompile`: stack-overflow instrumentation and stack usage instrumentation. The correct use of the Entry-Point ABI between the `Leon` and the `SHAVE` is critical to the valid use of these instrumentation hooks, as the values passed in IRF '`I19`' and '`I20`' provide the information required by this instrumentation.

The programmer may enable either or both forms of stack instrumentation, and if they enable both, then the usage instrumentation is evaluated before the overflow instrumentation. This means that in the event that the overflow instrumentation detects an overflow and aborts, the usage information will record the extent of the stack that was requested, allowing the programmer to make a more informed decision about how much stack they need to allocate to the program.

10.6.1 Stack-Overflow Instrumentation

One source of bugs is when the stack usage exceeds the amount of space intended for the stack. When this happens the data that precedes the space allocated for the stack is overwritten by the local objects allocated on the stack.

The stack is used to store explicit data requested by the programmer such as the local variables of a function, or they may be implicit data such as the arguments passed to variadic functions, and the values of registers that must be "spilled" to ensure that the preserved registers are restored to their original values on return from the function. See Table 9 in section 9.12 for details about the preserved registers.

If the amount of stack required by the function exceeds the remaining space available in the stack, then the content of the memory that occurs before the allocated space for the stack will be overwritten or "corrupted". If the program later depends on the validity and integrity of this overwritten memory, then it may result in unexpected execution of the program.

The use of the option '`-mstack-overflow-instrumentation`' instructs the compiler to insert instrumentation into the function that checks if the amount of stack requested by the function would exceed the amount available, and if so will cause an immediate '`abort`' of the program, yielding control back to the `Leon` without allowing the execution of the following code which might corrupt memory.

When the program aborts in this way, it will set the value of IRF '`I18`' to '`-3`' and terminate the `SHAVE` code using '`BRU.SWIH`' with the tag value '`SHAVEExitStackOverflow`' (defined in the header file '`<sys/shave_exitcodes.h>`').

10.6.2 Stack Usage Instrumentation

Stack-overflow detection is an important tool for tracking down unexpected failures in a program; however, it is also important for the programmer to know how much stack to allocate. If they allocate too little, then an overflow can occur, but if they allocate too much, then they will have unused space in the allocated stack that may be more usefully allocated to another purpose.

The stack usage instrumentation allows the programmer to observe how much stack a program actually uses[29], and this is enabled using the option '`-mstack-usage-instrumentation`'.

When selected, this option instructs the compiler to insert instrumentation that will check if the amount of stack requested by the function would exceed the previously known maximum usage. In practice this is the lowest address used by the stack since the stack grows towards zero. If the instrumentation indicates that the function causes the stack usage to increase since the previously noted "High Watermark", it updates this information with the new value.

---

[29] This is not completely true, as the provided libraries are not instrumented for stack usage information

When the SHAVE code returns control to the Leon code that invoked it, the Leon code can query this value by calling the Entry-Point:

```
uint32_t _EP_get_stackhighwater(void)
```

Which returns the address of the value determined by this instrumentation.

NOTE: Unlike other 'dllexport' Entry-Points, this one does not update the stack information or execution context information described in section 9.11.1.

10.6.3 Known Limitations

As with all forms of instrumentation, stack instrumentation involves altering the code generated versus the un-instrumented program. This means that the values and checks determined by the instrumentation may not be exactly the same as would be implied if the program was not instrumented. In particular, the program will be slightly larger when instrumentation is present, and the instrumentation itself may cause changes in the register utilization, which in turn may require additional register spills and consequently slightly greater amount of stack used than would be used by the un-instrumented version.

This is an unavoidable consequence of instrumentation.

11 Using moviCompile

Using `moviCompile` is similar to any C compiler. This section provides greater detail on platform specific aspects of using the compiler.

11.1 Simple usage

From a terminal window, type:

```
<path-to>/moviCompile -S -O1 test_struct_complex.c
```

Example C source file:

```
// 4 x 32-bit
#include <moviVectorUtils.h>

float4 v1, v2, v3;

float4 foo(float4 *v1, float4 *v2)
{
  return (*v1 + *v2);
}
int main()
{
  v3 = foo(&v1, &v2);
  return (v3.x + v3.y);
}
```

After compilation, the obtained file is:

```
;--------------------------------------------------------------------------
;
.version 00.51.05
;
; Generated by 'moviCompile v00.50.57.2
;--------------------------------------------------------------------------

.code .text.foo
.salign 16
.ealign 16

    foo:
        LSU1.LD.64.l v15 i17
            || LSU0.LD.64.l v14 i18
        LSU1.LDO.64.h v15 i17 8
            || LSU0.LDO.64.h v14 i18 8
        NOP 2
        BRU.JMP i30
        NOP 3
        VAU.ADD.f32 v23 v14 v15
        NOP 2

.code .text.main
.salign 16
.ealign 16

    main:
        LSU0.LDIL i18 v1
        IAU.SUB i19 i19 8
            || LSU1.LDIL i17 v2
            || LSU0.LDIH i18 v1
        LSU1.STO.32 i30 i19 4
            || LSU0.LDIH i17 v2
        LSU1.LDIL i30 foo
            || LSU0.LDIH i30 foo
        BRU.SWP i30 ; Function call
        NOP 6
        LSU1.LDO.32 i30 i19 4
        LSU1.LDIL i17 v3
        LSU0.CP i9 v23.1
        NOP 4
        BRU.JMP i30
        CMU.CPVI.x32 i10 v23.0
        SAU.ADD.f32 i10 i10 i9
        NOP
        LSU0.LDIH i17 v3
        CMU.CPII.f32.i32s i18 i10
            || LSU1.ST.64.l v23 i17
        IAU.ADD i19 i19 8
            || LSU1.STO.64.h v23 i17 8


.data .data.v1
.align 8
v1:
        .fill 16, 1

.data .data.v2
.align 8
v2:
        .fill 16, 1

.data .data.v3
.align 8
v3:
        .fill 16, 1


.end
```

11.2 Inline Assembly Usage

`moviCompile` supports the GCC style of inline assembly.

The format is as follows (the keywords "`__asm__`" and "`__asm`", instead of "`asm`", are also supported):

```
asm ( assembler template
          : output operands                 /* optional */
          : input operands                  /* optional */
          : list of clobbered registers     /* optional */
          );
```

Here are a few simple examples:

```
// A basic example with no register usage
asm ( "BRU.SWIH 0x1F\n" );

// An example with input and output registers
int z, y, z;
asm ( "IAU.ADD %0, %1, %2\n" : "=r"(z) : "r"(x), "r"(y) : );

// An example with input, output, and clobbered registers
int z, y, z;
asm ( "IAU.ADD I0, %1, %1\n\t"
      "IAU.ADD %0, I0, %2\n\t"
    : "=r"(z)
    : "r"(x), "r"(y)
    : "I0");
```

11.2.1 Inline Assembly and Scheduling

Because the compiler is unaware of the content of the inline-assembly block, it cannot perform its usual job of scheduling to avoid contention with other instructions. Prior to and following each inline-assembly block there may be instructions that the compiler has generated in response to the C or C++ code that contains the inline-assembly fragment. The compiler will automatically ensure that there are enough NOP cycles preceding the inline-assembly block so as to ensure that all side-effects and register write-backs have completed before the inline-assembly code starts.

However, because it is unaware of the content of the inline-assembly block, it has no way of knowing how many NOPs cycles should be placed FOLLOWING the inline-assembly block in order to ensure that instructions in the inline-assembly block do not conflict with or corrupt the expectations of compiler generated instructions following the inline-assembly block.

It is up the programmer of the inline-assembly block to ensure that there are enough cycles following their inline-assembly code so that all side-effects and register write-backs have completed before the compiler generated code continues. For example:

```
asm ( "IAU.ADD %0, %1, %2" : "=r"(z) : "r"(x), "r"(y) : );
```

This is OK for integers, because the latency of ‘`IAU.ADD`’ is zero and all side-effects and write-backs are completed before the next instruction. However:

```
asm ( "SAU.ADD.i32 %0, %1, %2" : "=r"(z) : "r"(x), "r"(y) : );
```

This is not OK for integers because the ‘`SAU.ADD`’ instruction has a latency of 1 for integers, and a latency of 2 for floating-point, so the programmer should write:

```
asm ( "SAU.ADD.i32 %0, %1, %2\n"
      "NOP"
      : "=r"(z) : "r"(x), "r"(y) : );
```

if they know the values are integers, or:

```
asm ( "SAU.ADD.f32 %0, %1, %2\n"
      "NOP 2"
      : "=r"(z) : "r"(x), "r"(y) : );
```

if they know the values are floating-point.

11.2.1.1 Using `NOP SYNC`

However, the compiler will emit the ‘`NOP SYNC`’ assembly pseudo instruction following the inline-assembly block. If the assembler `moviAsm` has it’s analyser enabled (the default), then it will calculate the exact minimum number of `NOP` cycles to insert (which may be zero) needed to ensure that the programmer’s inline-assembly code cannot accidentally conflict with the compiler generated code following their inline-assembly.

Consequently, unless the programmer disables `moviAsm`’s analyser, the problems described above are not a problem and the programmer does not need to explicitly provide addition `NOP` cycles following their inline-assembly code.

11.3 Writing Optimizer Friendly C Code

11.3.1 Introduction

The purpose of this document is to describe some practices and techniques that will help to better ensure that the C code you write can be transformed into more optimal assembly code representing the program. It uses an example-led approach, and explains at each stage “why” the compiler makes the decision it makes, and “how” you can reform your code to achieve better optimizations.

The techniques are generally applicable to any compiler and any processor target, but some targets will benefit more from the techniques described than others, and in particular the revised examples are generally going to produce better code for the `SHAVE` vector processing capabilities than for another non-vectorizing processor.

The document primarily targets C, but all of the techniques are equally applicable to C++.

The programming language C is a high level abstraction that allows a program to be expressed in a portable manner. Adhering to the ISO C Standard provides a very high degree of portability and platform independence.

Unfortunately, a high-level abstraction often comes at the expense of providing an “optimal” solution for any particular platform.

But there are things that you can do to ensure that your programs are better suited to optimization, and for the `SHAVE` processor in particular, which are better suited to auto-vectorization optimizations.

NOTE: You will observe that this is more or less “Tutorial Like” and walks through the process of optimizing a program written in C so that it better approximates the performance of a corresponding hand optimized equivalent written in `SHAVE` Assembler. Keeping this in mind should help clarify aspects of this document that are not otherwise clearly described. Indeed, the chapter was inspired by an exercise of doing precisely such a task with existing code in the Movidius MvCV kernel code where C code is provided to illustrate the semantic operation of the less easily understood hand optimized assembly code.

11.3.2 Iteration and Dataset Size Information

Consider the following example:

```
void copyFrame(size_t size, unsigned char* dst,
               unsigned char* src) {
  size_t i;
  for (i = 0; i < size; ++i)
    *dst++ = *src++;
}
```

This seems like a straightforward simple copy of ‘`size`’ number of elements from the memory pointed to by ‘`src`’ to the memory pointed to by ‘`dst`’, and indeed the high-level abstraction is valid though a production version might check for `NULL` and other prerequisite safety checks.

Here is a hypothetical `SHAVE` assembly version of the programmer’s intended function:

```
copyFrame_asm:
  IAU.SUB i18, i18, 16
    || LSU0.LDIL i14, 16
  PEU.PC1I NEQ
    || LSU1.LDO.64 v23.h, i16, 8
    || LSU0.LDINC.64 v23.l, i16, i14
  PEU.PC1I EQ
    || BRU.JMP i30
  NOP 5
  PEU.PC1I NEQ
    || BRU.BRA copyFrame_asm
    || LSU1.STO.64 v23.h, i17, 8
    || LSU0.STINC.64 v23.l, i17, i14
  NOP 6
```

Sounds reasonable? Yes, it “sounds” reasonable, but the programmer of this hand-tuned assembly code “knows” that the ‘`size`’ is always a multiple of ‘`16`’, and also that the size is at least ‘`16`’, and they have embodied this knowledge in the assembly implementation.

By contrast, nowhere in the high-level C code above is there any indication to the compiler that the size is in anyway constrained to multiples of 16. Instead the compiler has to assume that that data has an arbitrary size and must generate code pessimistically to deal with this.

So in this case, the hand-written assembly program and the C high-level program are not for the same program, and there is no way that the compiler could (even in theory) come up with the same solution as the assembly writer with their “implicit” knowledge of the data set.

What can be done in the case? Well, first of all “tell” the compiler that the data-set is a multiple of 16 - you weren’t afraid to say it in assembly, so why not also say it in C?

In this case a very simple adjustment to the program is all that is necessary:

```
void copyFrame(size_t size, unsigned char* dst,
               unsigned char* src) {
  do {
    int j;
    for (j = 0; j < 16; ++j)
      *dst++ = *src++;
    size -= 16;
  } while (size);
}
```

Now the chunk size of ‘`16`’ is explicit in the C code as it is in the hand-written assembly and also the knowledge that the data set contains “at least” 16 bytes, so the programs are a lot closer to being equivalent, and the compiler has better information to use when selecting appropriate instructions and knows that the dataset is multiple of 16 in length, just as the hand-written assembly knows this.

11.3.3 Aliasing

But for some reason, despite the C code now telling the compiler that the dataset length is a multiple of 16, it still does not choose to move 16-bytes at a time! Why not? This is because of “aliasing”.

Aliasing refers to the property where two or more pointers may appear unrelated, but may in fact refer to the same place in memory. This is a fundamental issue for C programs which support “raw” pointers to the underlying machine data. So when you see something like:

```
    for (j = 0; j < 16; ++j)
      *dst++ = *src++;
```

The compiler “must” assume that the location pointed to by ‘`dst`’ is to the same or overlapping location that is pointed to by ‘`src+1`’ - it has no way of knowing that this is not the case, so it has to read the memory a byte (‘`sizeof char`’) at a time to ensure that the aliased semantics are respected.

Incidentally, aliasing is also a problem when you call a function - any function - the compiler “must” assume that the called function altered the data-set that the current function has a pointer to, for example:

```
char message[] = “Hello”;

char foo(const char* myString) {
  if (*myString == ‘H’)
    printf(“Probably Hello: %s\n”, myString);
  return *myString;
}

int main() {
  foo(message);
  return 0;
}
```

In this case you might expect that there is no reason to load '`*myString`' twice, yet the compiler will do so. This is because it has to assume that the function '`printf`' might have altered the memory pointed to by '`myString`'! How? Well a simple revision of this example will show how this might be possible:

```
char message[] = "Hello";

__attribute((noinline)) // To prevent the optimizer seeing this
void bar() {
  strcpy(message, "World");
}

char foo(const char* myString) {
  if (*myString == 'H')
    bar();  // Instead of 'printf'
  return *myString;
}

int main() {
  foo(message);
  return 0;
}
```

Clearly the reload is essential, and the reason that the compiler will perform this reload is due to the assumption that the data pointed to by '`myString`' has been aliased.

Referring to the previous example, what does the "call" site look-like? Well let's describe it as follows:

```
void nonAliasedData() {
  int i;
  unsigned char duplicateFrame[32];
  unsigned char originalFrame[32];

  // Initialise the original frame data
  for (i = 0; i < 32; ++i)
    originalFrame[i] = i;

  // Copy the frame
  copyFrame(32, duplicateFrame, originalFrame);

  // Verify that the copy was successful
  for (i = 0; i < 32; ++i)
    assert(duplicateFrame[i] == i);
}
```

This is reasonable too, and both the C solution above and the hand-written assembly will successfully compute the same values, though the hand-written assembly does it faster than the C code due to the failure of the compiler to vectorize into chunks of 16. But why does the compiler not choose to do it in chunks of 16?

Under ISO C rules, because of aliasing, when the compiler is processing '`copyFrame`' which does not know about the datasets, it is "required" to assume that '`src`' and '`dst`' may refer to the same or overlapping memory. So let's try a different data set with deliberately overlapping data (aliased data-sets):

```
void aliasedDataCopy() {
  int i;
  unsigned char frame[34];
```

```
        // Initialise the original frame data
        for (i = 0; i < 32; ++i)
          frame[i] = i;

        // Copy the frame
        copyFrame(32, frame + 2, frame);

        // Verify that the copy was successful
        for (i = 0; i < 32; ++i)
          assert(frame[i + 2] == (i % 2));
      }
```

What?

Well look at ‘`frame`’ before the copy, but after it has been initialized:

```
      frame ::
         0  1  2  3  4  5  6  7  8  9
        10 11 12 13 14 15 16 17 18 19
        20 21 22 23 24 25 26 27 28 29
        30 31 XX YY
```

Copying a single byte at a time to the “offset by 2” target yields (for each iteration):

```
      frame ::
         0  1  0  3  4  5  6  7  8  9
        10 11 12 13 14 15 16 17 18 19
        20 21 22 23 24 25 26 27 28 29
        30 31 XX YY
      frame ::
         0  1  0  1  4  5  6  7  8  9
        10 11 12 13 14 15 16 17 18 19
        20 21 22 23 24 25 26 27 28 29
        30 31 XX YY
      frame ::
         0  1  0  1  0  5  6  7  8  9
        10 11 12 13 14 15 16 17 18 19
        20 21 22 23 24 25 26 27 28 29
        30 31 XX YY
      frame ::
         0  1  0  1  0  1  6  7  8  9
        10 11 12 13 14 15 16 17 18 19
        20 21 22 23 24 25 26 27 28 29
        30 31 XX YY
      ...
      frame ::
         0  1  0  1  0  1  0  1  0  1
         0  1  0  1  0  1  0  1  0  1
         0  1  0  1  0  1  0  1  0  1
         0  1  0  1
```

In this case the “C” version does not fail the assertion, but the hand-written assembly does fail? Why? Well the hand-written assembly implements the copy as 2 x 16-byte copies, and the copy dataset looks like:

```
frame ::
   0  1  0  1  2  3  4  5  6  7
   8  9 10 11 12 13 14 15 18 19
  20 21 22 23 24 25 26 27 28 29
  30 31 XX YY
frame ::
   0  1  0  1  2  3  4  5  6  7
   8  9 10 11 12 13 14 15 14 15
  18 19 20 21 22 23 24 25 26 27
  28 29 30 31
```

A rather different outcome!

So what is going on here? Well again the programmer of the hand-written code has encoded some implicit assumptions in the implementation that are NOT also expressed in the corresponding C code - in this case they have implicitly assumed that the datasets do NOT overlap; so once again, the C program and the Assembly program are not the SAME program.

The C code is correct, it has honored ISO C rules regarding aliasing, and generated conservative code that implements this copy safely and correctly.

But in our case we “know” that overlapping datasets are never going to happen, and we want to use the more optimal solution that the hand-written assembly code “knows” is right.

In ISO C99 the Standard added the keyword ‘`restrict`’. The use of ‘`restrict`’ tells the compiler:

> “Trust me; the data referred to by this pointer is not aliased”

So a simple change to the declaration of ‘`copyFrame`’ lets the compiler choose a more optimal sequence of instructions than it is otherwise “permitted” to choose.

However, if you “do” pass an aliased data-set to a program where you have explicitly used ‘`restrict`’, then the program will have undefined behavior under ISO C rules, and the resulting data-set can differ wildly from what you might expect, and from platform to platform, and may even crash the program. Remember “Undefined” means that there is no valid defined result.

Using ‘`restrict`’ carefully for this example, you would simply rewrite the program as:

```
void copyFrame(size_t size, unsigned char* restrict dst,
               unsigned char* restrict src) {
  do {
    int j;
    for (j = 0; j < 16; ++j)
      *dst++ = *src++;
    size -= 16;
  } while (size);
}
```

Now the C program and the hand-written assembly program are much closer to expressing the “same” program:

- Both now know that the expected data-set is a multiple of 16-bytes.
- They also both know that there are “at least” 16-bytes in the data-set.
- And finally, they both assume that the data-sets do NOT overlap (or alias in C terminology).

So what does the compiler do with this information? The code generated is now:

```
copyFrame:
.LBB2_1:
  IAU.INCS i18 -16
    || LSU0.LD.64.l v15 i16
    || LSU1.LDO.64.h v15 i16 8
  CMU.CMZ.i32 i18
  PEU.PC1C NEQ
    || BRU.BRA .LBB2_1
  IAU.ADD i16 i16 16
  NOP 3
  LSU1.ST.64.l v15 i17
  IAU.ADD i17 i17 16
    || LSU1.STO.64.h v15 i17 8
  BRU.JMP i30
  NOP 6
```

And this is remarkably similar to the hand-tuned code.

But you can see that just these two simple "hints" to the compiler have allowed for a radically different choice in code generation - and the code is still not specifically tuned for `SHAVE` and is still ISO C99 Standards compliant.

The ISO C++ Standards do not define the keyword '`restrict`', but both the `GCC` Tool-Chain for the `Leon` and the `moviCompile` compiler for `SHAVE` implement the extension keyword '`__restrict`' which has exactly the same semantics as '`restrict`' from the ISO C99 Standard, but can be used with C++ or C90.

11.3.4 Using Vectorizing '#pragma's

`CLang` v3.5 introduced a set of '`#pragma`' directives to help provide additional hints to the compiler about the data so help with optimization. The full set of '`#pragma`' directives for vectorization can be found at the following link:

```
http://clang.llvm.org/docs/LanguageExtensions.html#extensions-for-loop-hint-
optimizations
```

So let's return to the original example, but this time make no alterations to the source code except to add a few of these '`#pragma`' directives:

```
void copyFrame(size_t size, unsigned char* dst,
               unsigned char* src) {
  size_t i;

#pragma clang loop vectorize(enable) vectorize_width(16)
#pragma clang loop interleave(enable)
#pragma clang loop unroll(enable)

  for (i = 0; i < size; ++i)
    *dst++ = *src++;
}
```

These '`#pragma`'s convey "some" but not all of the implicit assumptions about the data-set that are present in the hand-written assembly version. Specifically the compiler is being instructed that the programmer thinks that the use of vectorization and loop-unrolling will produce better code, and most importantly, the magic number ‘16’ is specifically provided, telling the compiler that it is permitted to group these 16-elements at a time.

The resulting code is better than the unaltered original, but not as good as the code generated by the previous example. However, it has the virtue of leaving the source code completely unaltered except for the addition of some ‘#pragma’ hints.

Consider another example; this one is from the MvCV kernel ‘absoluteDiff’:

```
#include <stdint.h>
#include <stdlib.h>
#include "absoluteDiff.h"

void mvcvAbsoluteDiff(uint8_t** in1,
                      uint8_t** in2,
                      uint8_t** out,
                      uint32_t width) {
  uint8_t* in_1 = *in1;
  uint8_t* in_2 = *in2;
  uint8_t* out_p = *out;

#pragma clang loop vectorize(enable) vectorize_width(16)
#pragma clang loop interleave(enable)
#pragma clang loop unroll(enable)

  for (uint32_t j = 0; j < width; j++)
    out_p[j] = abs(in_1[j] - in_2[j]);
}
```

The code generated with and without these ‘#pragma’s is no different, but the change to locally cache the first level indirection of ‘out’ and store it as ‘out_p’ has allowed the compiler to auto-vectorize anyway, and the resulting code is larger, but more performant than the original, and it is vectorized.

The following version of the code explicitly codifies that the data-set is expected to be a multiple of 16, so as for the ‘copyFrame’ example, knowledge that is present in the hand tuned assembly is also made known to the C compiler.

```
#include <stdint.h>
#include <stdlib.h>
#include "absoluteDiff.h"

void mvcvAbsoluteDiff(uint8_t** in1,
                      uint8_t** in2,
                      uint8_t** out,
                      uint32_t width) {
  uint8_t* in_1 = *in1;
  uint8_t* in_2 = *in2;
  uint8_t* out_p = *out;

#pragma clang loop vectorize(enable) vectorize_width(16)
#pragma clang loop interleave(enable)
#pragma clang loop unroll(enable)

  for (uint32_t j = 0; j < width;)
    for (int i = 0; i < 16; ++i, ++j)
      out_p[j] = abs(in_1[j] - in_2[j]);
```

```
        }
```

And this version uses loop-unrolling, but it is still not vectorizing as expected. And even when the vectorization does happen, it is not what the hand-written assembly version of this will do; and in particular it is vectorizing using vectors of ‘`uint4`’ (4 x 32-bit integers), and not using ‘`uchar16`’ (16 x 8-bit integers).

Why this happens is the subject of the next section on “Usual Arithmetic Conversions”.

11.3.5 Usual Arithmetic Conversions

The previous example vectorizes okay, but does so using units of 4 x 32-bit integers and not the 16 x 8-bit registers that the hand-tuned assembly uses. The problem here is how the “C Language” evaluates arithmetic expressions, and in particular what is happening in the sub-expression:

```
        in_1[j] - in_2[j]
```

To understand this, you need to understand how the ISO C Standard describes such expressions.

Each of the sub-expressions ‘`in_1[j]`’ and ‘`in_2[j]`’ have the type ‘`uint8_t`’ which on `SHAVE` is ‘`unsigned char`’, so the sub-expression we are interested in is a subtraction of two objects with the type ‘`unsigned char`’. In ISO C99, section “6.5.6 Additive operators”, there is a fairly large description of the semantics involved, but the extract of importance here is:

> If both operands have arithmetic type, the usual arithmetic conversions are performed on them.

These are described in section “6.3.1.8 Usual arithmetic conversions” of ISO C99. Under C99, the actual computation of the subtraction is performed as an ‘`int`’ or ‘`unsigned int`’ and not as an ‘`unsigned char`’, so in this case the C compiler frontend (`CLang`) will apply the “integer promotions” to the two operands prior to performing the subtraction. These are described in section “6.3.1.1 Boolean, characters, and integers”, and the relevant paragraph states:

> If an **int** can represent all values of the original type, the value is converted to an **int**; otherwise, it is converted to an **unsigned int**. These are called the *integer promotions*

So for the `SHAVE` target, both of the ‘`unsigned char`’ operands are first promoted to ‘`int`’ and the subtraction takes place as an ‘`int`’ expression. The implication of this is that the frontend whose job it is to enforce the semantics of ISO C will pass information onto the backend whose job it is to generate code, and this information tells the backend that the data-types involved in the subtraction are 32-bit integers and not the 8-bit integers that you might have expected. So the type of the expression ‘`in_1[j] - in_2[j]`’ is actually a ‘`signed int`’ and not an ‘`unsigned char`’.

The next stage comes when the auto-vectorizer tries to vectorize these operations, and since it sees them described as operations involving ‘`signed int`’, it has no choice but to respect that requirement as instructed by the frontend, so while the vectorization does take place, it is in units of “4 x 32-bit integers” and not the desired “16 x 8-bit integers”, hence the use of instructions like ‘`VAU.SUB.i32`’.

So what can be done about this[30]? Well, we can’t break the semantics of Standard C because a whole lot of bad things will happen, so this is where extensions to C can be used, and in particular the extensions for “Explicit Vectorization”.

[30] Actually, this issue is what inspired us to implement the UAC reduction optimization which analyses the code to see if it can use a smaller integer type without altering the outcome. This occurs before vectorization, so in fact the current compiler can deduce that it is possible to vectorize in units of 16 x 8-bit. You can disabled this using ‘`-fno-uac-reduction-pass`’ to see what used to happen.

11.3.6 Explicit Vectorization

The deficiencies of C for the purposes of vectorizing programs has been a problem for a very long time, and has been the topic of many attempts to help the programmer convey some of their knowledge about the data and the computational requirements to the compiler optimization process so as to better exploit the capabilities of the processor. Examples of these are the '`#pragma`'s that `GCC` and `CLang` have added, but also more structured approaches such as OpenMP and in particular OpenCL.

`moviCompile` does not support OpenCL, but it has borrowed some ideas from OpenCL to facilitate the writing of more optimal code. These are described in the header '`<moviVectorUtils.h>`' (section 6.3), and in particular for this example, the data-type '`uchar16`'.

In this case it is not possible to force the compiler to perform the '`unsigned char`' expressions using 8-bit arithmetic because of C's usual arithmetic conversion rules, but extensions are not subject to ISO C's rules. But by following the example of OpenCL, we are also not being entirely non-standard, as the OpenCL specification does describe the semantics needed. So rewriting this example using just a couple of OpenCL inspired extensions:

```
#include <stdint.h>
#include <stdlib.h>
#include <moviVectorUtils.h>
#include "absoluteDiff.h"

void mvcvAbsoluteDiff(uint8_t** in1,
                      uint8_t** in2,
                      uint8_t** out,
                      uint32_t width) {
  uchar16* in_1 = *(uchar16**)in1;
  uchar16* in_2 = *(uchar16**)in2;
  uchar16* out_p = *(uchar16**)out;

  width >>= 4;

  for (uint32_t j = 0; j < width; ++j)
    out_p[j] = mvuAdiff(in_1[j], in_2[j]);
}
```

In this case, two elements have been borrowed from OpenCL, specifically the vector data-type '`uchar16`' and the OpenCL function '`abs_diff`' which we have called '`mvuAdiff`' in the Movidius Vector Utility Library.

The resulting code-generation is very close to that of the hand-tuned assembly, and the C source still retains a very high degree of readability that is not too `SHAVE` specific ('`mvuDiff`' vs '`abs_diff`'). With the compiler, this example produces the following fairly reasonable assembly code at '`-O3`':

```
mvcvAbsoluteDiff:
  LSU0.LDIL i10 0x0010
  CMU.CMII.u32 i15 i10
  PEU.PC1C LT
    || BRU.JMP i30
  NOP 6

.LBB3_1:
  LSU1.LD.32 i9 i16
  LSU0.LD.32 i10 i17
    || LSU1.LD.32 i8 i18
  NOP 6

.LBB3_2:
  LSU1.LD.64.l v15 i10
    || LSU0.LD.64.l v14 i8
  LSU1.LDO.64.h v15 i10 8
    || LSU0.LDO.64.h v14 i8 8
  NOP 3
  BRU.BRA .LBB3_2
  NOP 2
  VAU.ADIFF.u8 v15 v14 v15
  NOP
  LSU1.ST.64.l v15 i9
  LSU1.STO.64.h v15 i9 8
```

On further inspection, it turns out that the hand-tuned assembly actually treats the data-set as having a multiple of 32-bytes, not 16 as I assumed. A simple revision to the above test to share this implicit knowledge with the C compiler is as follows:

```
  #include <stdint.h>
  #include <stdlib.h>
  #include <moviVectorUtils.h>
  #include "absoluteDiff.h"

  void mvcvAbsoluteDiff(uint8_t** in1,
                        uint8_t** in2,
                        uint8_t** out,
                        uint32_t width) {
    uchar16* in_1 = *(uchar16**)in1;
    uchar16* in_2 = *(uchar16**)in2;
    uchar16* out_p = *(uchar16**)out;

    width >>= 4;

    for (uint32_t j = 0; j < width; j += 2) {
      out_p[j] = mvuAdiff(in_1[j], in_2[j]);
      out_p[j+1] = mvuAdiff(in_1[j+1], in_2[j+1]);
    }
  }
```

And this is turn generates the following assembly code:

```
  mvcvAbsoluteDiff:
    IAU.SHR.u32 i10 i15 4
    CMU.CMZ.i32 i10
    PEU.PC1C EQ
      || BRU.JMP i30
    NOP 6

  .LBB3_1:
    LSU1.LD.32 i8 i17
      || LSU0.LD.32 i7 i16
    LSU1.LD.32 i9 i18
    LSU0.LDIL i6 0
```

```
        NOP 4
        SAU.ADD.i32 i7 i7 16
          || IAU.ADD i8 i8 16
        IAU.ADD i9 i9 16

      .LBB3_2:
        LSU1.LDO.64.l v15 i8 -16
          || LSU0.LDO.64.l v14 i9 -16
        LSU1.LDO.64.h v15 i8 -8
          || LSU0.LDO.64.h v14 i9 -8
        NOP 6
        VAU.ADIFF.u8 v15 v14 v15
        NOP
        LSU1.STO.64.l v15 i7 -16
        LSU1.STO.64.h v15 i7 -8
        LSU1.LD.64.l v15 i8
          || LSU0.LD.64.l v23 i9
        LSU1.LDO.64.h v15 i8 8
          || LSU0.LDO.64.h v23 i9 8
        IAU.ADD i9 i9 32
        IAU.ADD i6 i6 2
        CMU.CMII.u32 i6 i10
        PEU.PC1C LT
          || BRU.BRA .LBB3_2
        NOP 2
        VAU.ADIFF.u8 v15 v23 v15
        IAU.ADD i8 i8 32
        LSU1.ST.64.l v15 i7
        IAU.ADD i7 i7 32
          || LSU1.STO.64.h v15 i7 8
        BRU.JMP i30
        NOP 6
```

The final version combines the work of all of the above, and includes an example of when NOT to use vectorization.

This version of the example does not assume anything about the data-set size, except that it is probably reasonable to assume that it is large enough that vectors of 16 by 8-bit integers will represent the bulk of the work and that it is likely to be 32-bytes or more so some unrolling is worthwhile; and with the residual 15 or less bytes being tidied up by a scalar loop.

In this tidying up scalar loop, there is very little point in vectorizing and loop unrolling, as the cost will not be saved over the tiny data-set (in fact it is likely to be more costly), so in this example the '`#pragma`' directive is used to **disable** the auto-vectorizer.

It has also been adjusted for C++ conformance:

```
#include <cstdint>
#include <cstdlib.h>
#include <moviVectorUtils.h>

using namespace std;
```

```
// Assume the data set has an arbitrary size
void mvcvAbsoluteDiff(uint8_t** in1,
                      uint8_t** in2,
                      uint8_t** out,
                      uint32_t width) {
  uchar16* in_x = *reinterpret_cast<uchar16**>(in1);
  uchar16* in_y = *reinterpret_cast<uchar16**>(in2);
  uchar16* out_z = *reinterpret_cast<uchar16**>(out);

  // First process in vector chunks
  const uint32_t width16 = (width >> 4);

#pragma clang loop unroll_count(2)
  for (uint32_t j = 0; j < width16; ++j)
    out_z[j] = mvuAdiff(in_x[j], in_y[j]);

  // Finish off the residual bytes with a scalar loop
  uint8_t* in_1 = *in1 + (width & ~15u);
  uint8_t* in_2 = *in2 + (width & ~15u);
  uint8_t* out_p = *out + (width & ~15u);

#pragma clang loop vectorize(disable)
#pragma clang loop interleave(enable)
#pragma clang loop unroll(enable)
  for (int j = 0; j < (width & 15); ++j)
    out_p[j] = abs(in_1[j] - in_2[j]);
}
```

The compiler can now emit an optimized loop using vectors of 16 by 8-bit values for the main block of data. The unrolling also gives it the hint that it might be worthwhile unrolling the loop to the depth of 2 which implies that there are likely to be 32-bytes or more of data; and finally it uses non-vectorized scalar code for the residual bytes at the end of the data-set. This code is physically larger, but can handle an arbitrary number of bytes in the data-sets:

```
mvcvAbsoluteDiff:
  IAU.SHR.u32 i10 i15 4
    || LSU1.LD.32 i7 i16
  CMU.CMZ.i32 i10
    || LSU1.LD.32 i9 i18
    || LSU0.LD.32 i8 i17
  PEU.PC1C EQ
    || BRU.BRA .LBB5_5
  NOP 5
  PEU.PC1C NEQ
    || SAU.ADD.i32 i7 i7 16
    || IAU.ADD i8 i8 16
  IAU.ADD i9 i9 16
    || LSU0.LDIL i6 0x0001

.LBB5_2:
  LSU1.LDO.64.l v15 i8 -16
    || LSU0.LDO.64.l v14 i9 -16
  LSU1.LDO.64.h v15 i8 -8
    || LSU0.LDO.64.h v14 i9 -8
  NOP 2
  CMU.CMII.u32 i6 i10
  PEU.PC1C GTE
    || BRU.BRA .LBB5_4
  NOP 2
  VAU.ADIFF.u8 v15 v14 v15
  NOP
  LSU1.STO.64.l v15 i7 -16
  LSU1.STO.64.h v15 i7 -8
  LSU1.LD.64.l v15 i8
    || LSU0.LD.64.l v14 i9
  IAU.ADD i9 i9 32
    || LSU1.LDO.64.h v15 i8 8
    || LSU0.LDO.64.h v14 i9 8
  NOP
  IAU.ADD i5 i6 1
  CMU.CMII.u32 i5 i10
  PEU.PC1C LT
    || BRU.BRA .LBB5_2
  SAU.ADD.i32 i6 i6 2
  NOP
  VAU.ADIFF.u8 v15 v14 v15
  IAU.ADD i8 i8 32
  LSU1.ST.64.l v15 i7
  IAU.ADD i7 i7 32
    || LSU1.STO.64.h v15 i7 8

.LBB5_4:
  LSU1.LD.32 i7 i16
  LSU0.LD.32 i8 i17
    || LSU1.LD.32 i9 i18
  NOP 6

.LBB5_5:
  LSU0.LDIL i10 0x000f
  IAU.AND i10 i15 i10
  CMU.CMZ.i32 i10
  PEU.PC1C EQ
    || BRU.JMP i30
  NOP 6

.LBB5_6:
  LSU0.LDIL i5 0
  LSU1.LDIL i6 0xfff0
    || LSU0.LDIH i6 0xffff
  SAU.SUB.i32 i10 i5 i10
    || IAU.AND i6 i15 i6
  SAU.ADD.i32 i9 i9 i6
    || IAU.ADD i8 i8 i6
  IAU.ADD i7 i7 i6
```

```
.LBB5_7:
  LSU0.LDI.32.u8.u32 i6 i9
    || LSU1.LDI.32.u8.u32 i5 i8
  NOP 3
  IAU.ADD i10 i10 1
  CMU.CMZ.i32 i10
  PEU.PC1C NEQ
    || BRU.BRA .LBB5_7
  IAU.SUB i6 i6 i5
  IAU.SHR.i32 i5 i6 31
  IAU.ADD i6 i6 i5
  IAU.XOR i6 i6 i5
  LSU1.STI.8 i6 i7
  PEU.PC1C EQ
    || BRU.JMP i30
  NOP 6
```

11.3.7 Implicit Function Declarations

Okay, this sub-section is not actually about optimizations, but it describes a problem that comes up a lot, and whose diagnosis can be very difficult.

In a nutshell, for C++ programs it is always illegal to refer to a name before it has been declared, so calling a function whose declaration has not yet been seen will result in a compile-time error in C++. The same is NOT true for C although the compiler will probably issue a warning.

You need to be very careful with C, and ensuring that a declaration occurs before use is critical in a modern C compiler.

11.3.7.1 Why is this?

Well, modern C compilers can place arguments to functions on the stack or in registers, but why, what and when it makes this choice is dependent on the function prototype. For example, let's take a very simple "complete" example, and one that will break for `SHAVE` and probably for other targets too:

```
int main(void) {
  printf("Hello %s\n", "World");
  return 0;
}
```

This breaks?

Yes, but it is the "why" it breaks that is important. Note that it does NOT declare '`printf`'. Examining the correct prototype for '`printf`' reveals that its correct prototype as specified by ISO C and which is <u>ONLY</u> validly introduced using '`#include <stdio.h>`' is as follows:

```
extern int printf(const char*, ...);
```

This is a variadic function - that is, a function which takes a variable number of arguments. Argument passing to variadic functions is typically achieved in one of two ways:

1. All of the arguments are placed on the stack
2. Arguments which have a specified type in the prototype are placed in registers, and the remainder are placed on the stack

The method used by `moviCompile` for variadic function argument places all arguments on the stack (this is the `GCC` convention).

Compare this against a non-variadic function, for example:

```
extern int bar(const char*, const char*);

int main(void) {
  bar("Hello %s\n", "World");
  return 0;
}
```

In this case the compiler has been provided with a specific type for both arguments and `moviCompile` will place both in registers when calling the function '`bar`'.

11.3.7.2 So is this a `moviCompile` bug?

Well curiously enough, no.

Effectively all compilers have the following choices for passing arguments:

1. Always place all arguments on the stack - this is the earliest form of argument passing for C (PDP-11 era) and is seldom used on modern general purpose processors
2. If the function has a non-variadic prototype, pass as many as possible in registers, and the remainder on the stack (this is the method `moviCompile` uses)
3. If the function has a variadic prototype:
    a. Pass as many of the arguments as possible in registers where the corresponding argument has been specified in the prototype, and all others on the stack
    b. Pass all arguments on the stack (this is the method `moviCompile` uses)

So let's have a look at another ANSI C89 (ISO C90) compliant example:

```
extern int foo(const char*, ...);
extern int bar(const char*, const char*);

int main(void) {
  foo("Hello %s\n", "World");
  bar("Hello %s\n", "World");
  huh("Hello %s\n", "World");
  return 0;
}
```

In this example there are three functions called, each with identical arguments. The first two have prototypes, but '`foo`' is a variadic function while '`bar`' is not. The third function '`huh`' has no prototype.

With `moviCompile` the function '`bar`' is called with the address of the string literal '`"Hello %s\n"`' placed in register `I18`, and the address of the string literal '`"World"`' placed in register `I17`.

By contrast, the function '`foo`' is called by placing no arguments in registers, and both argument are instead placed on the stack.

So how should '`huh`' be called? No prototype is present, so should the compiler call the function by placing arguments in registers like '`bar`', or by placing them on the stack like '`foo`'? Well `moviCompile` works out the types of the arguments, and passes them in registers like '`bar`'. This means that if the "actual" function '`huh`' has been defined like '`bar`', then all is well; but if it is defined like '`foo`' then it is not going to work.

But if `moviCompile` implemented this the other way around and '`huh`' was actually defined like '`bar`' then that would break too - you can't win!!!

What does ISO C99 have say about this? Section "6.5.2.2 Function calls" says:

> If the expression that denotes the called function has a type that does not include a prototype, the integer promotions are performed on each argument, and arguments that have type **`float`** are promoted to **`double`**. These are called the *default argument promotions*. If the number of arguments does not equal the number of parameters, the behavior is undefined. If the function is defined with a type that includes a prototype, and either the prototype ends with an ellipsis (`, ...`) or the types of the arguments after promotion are not compatible with the types of the parameters, the behavior is undefined.

So this means that the method that `moviCompile` uses is the correct way. In fact, implicit function declarations are illegal in C99 so this type of error will not happen undetected.

Returning to the original example, the call is to '`printf`', but '`printf`' has not been declared. Consequently the compiler treats it exactly like the function '`huh`' above and passes both arguments in registers. But '`printf`' has been "defined" in C using ellipsis and so the program falls foul of ISO C99 rules and the program has "undefined behavior". And indeed it does break if compiled with `moviCompile`.

11.3.7.3 In Summary

Always ensure that a function or object is declared before it is used, even though C90 does not mandate it.

If the function or object is defined by ISO C, then the only valid way of declaring it is to include the appropriate header - do not be tempted to "declare" your own ISO C names, or a whole slew of other bad things can happen.

A useful resource for finding ISO C and C++ names and their associated headers is:

```
http://www.cplusplus.com/
```

NOTE: C++ and C99 require that a function is declared before it is called, but because C90 still allows tentative declarations the same problem can occur. This is never a problem in C++ which has always required complete prototypes in the declarations.

TIP: It is not a bad idea to add '`–Werror=implicit-function-declaration`' to the set of `moviCompile` options to draw attention to this potential problem.

12 Appendix A – Reporting `moviCompile` Bugs

Despite considerable efforts to minimize bugs in `moviCompile`, there will always be occasions where a bug is detected that the compiler development team had not been aware of. To help the Movidius compiler team correct such bugs and to enhance the quality of the compiler, it is important that any bugs discovered are reported to Movidius at the `movidius.org` customer support web-site.

Bugs can occur in many shapes and forms, and providing suitable information is essential to allowing the problem to be analyzed and fixed as quickly as possible.

12.1.1 Fixing Bugs

On receipt of a bug report, we will attempt to find a viable work-around for the defect that will allow the programmer to continue the development of their programs. In general, bugs that have a viable work-around will have a formal fix in the following scheduled revision of `moviCompile`.

However, in some cases the bug may not have a viable work-around. In this case, and depending on the severity of the bug, when we have a fix for the defect ready we will provide a solution in one of the following ways:

Scheduled Release: Include the fix in the next scheduled release of `moviCompile` – this is usually the choice if a Scheduled release is imminent. A Scheduled release goes through extensive testing to reduce the probability that a bug is present in the compiler, but despite continuous enhancement to the testing process, bugs may unfortunately remain undetected until after the product is released.

Patch Release: Include the fix in a "Patch Release" of `moviCompile` – patch releases are occasionally issued when there are a number of bugs that are severe enough to warrant the production of an interim release of the compiler. These Patch releases are always made to the current released version of the compiler and include no new features. They will also include cumulative fixes for other bugs that have been identified and fixed since the previous Scheduled or Patch release.

A Patch release also goes through the same rigorous test process as a normal Scheduled release.

HotFix: Produce a "HotFix" to `moviCompile` – HotFixes are emergency repair versions that are intended to fix a very discrete issue that is urgently required by a customer. A HotFix will be issued in the limited context where the fix does not imply rebuilding the libraries or changing the headers that ship with `moviCompile`. They are also conditional on the limited scope of the fix, and the low probability that the fix might have other unintended consequences.

A HotFix version of the compiler will not have gone through the same rigorous testing as a Scheduled or Patch release. Finally, the nature of the change for a HotFix may not be propagated in the same way to the next Scheduled or Patch release (e.g. the addition of a new option in a HotFix).

12.2 Compiler Crashes

In the rare event that `moviCompile` crashes, it generally tries to create two files for the customer to submit to Movidius for analysis. These two files are:

1. The pre-processed C or C++ file that was being compiled at the time the crash occurred. This has all of the included header files expanded, and all macro substitutions completed
2. A small "shell" script that captures the exact set of options that were passed to `moviCompile` at the time the crash occurred

These two files are named:

```
/tmp/yourFile-hexCode.c (or yourFile-hexCode.cpp)
/tmp/yourFile-hexCode.sh
```

The value of '*hexCode*' for both files will be the same for any particular crash, and this is an automatically generated unique 8-digit hexadecimal value. The shell script will be have the extension '`.sh`' even on Windows.

In the event that a compiler crash does occur, please attach both of these files to the bug ticket. Usually when a crash occurs, `moviCompile` will automatically inform you that it has generated these two files and that they should be sent to the Movidius compiler support team.

12.3 Code-Generation Errors

Sometimes a bug may have nothing to do with a crash, but it may appear that the compiler has generated incorrect code. If you suspect that the compiler has generated incorrect code, the compiler team still needs the same kind of information that a crash would have automatically produced, but in this case the two files have to be created manually.

To do this it is recommended that you use the following procedure.

12.3.1 Preparing the Source File

1. Find the exact command-line options used when invoking `moviCompile`, call this '`<original-options>`'.
2. If '`-S`' or '`-c`' is present in '`<original-options>`' remove it; add '`-E -frewrite-includes`'; if '`-o <filename>.s`', '`-o <filename>.o`' or '`-o <filename>.asmgen`' is present remove it; add '`-o reduced.c`' or '`-o reduced.cpp`' depending on whether it is a C or C++ source. I'll refer to this edited version of '`<original-options>`' as '`<preproc-options>`'.
3. Now invoke `moviCompile` with these options, thus:

   ```
   <your-path-to>/moviCompile <preproc-options>
   ```

   This produces a file named '`reduced.c`' or '`reduced.cpp`' which is the pre-processed version of the original test-case.

You should submit this file, and the options '`<original-options>`' to the Movidius support site, along with a description of the problem you observe in the generated assembly code, the exact version of the compiler as reported by[31]:

```
<your-path-to>/moviCompile --version
```

and clearly identify where in the generated assembly file you have observed the suspicious code with as much detail about what you think is wrong as possible; perhaps by providing the generated assembly code with the ticket and indicating with comments or annotations what it is that concerns you about the generated code.

NOTE: If you are using inline-assembly, please ensure first that there are enough NOPs after the final instruction of your inline-assembly code to account for all outstanding latencies in the instructions you have used in the inline-assembly – see section 11.2.1. Failure to do this can result in the instructions you have used in the inline-assembly code being in contention with the instructions generated by the compiler following your inline-assembly code.

TIP!: If you are using the Movidius MDK, this provides support in the Makefile that will allow you to more simply produce the pre-processed file. You should refer to the MDK documentation for information on how to do this.

[31] Because of the different development schedules, the version reported by `moviCompile` and the version for the collection of development tools in which it is contained are commonly different, and it is important to know the exact version of `moviCompile` and not the tool set in which it is contained

TIP!: By adding the options '`-g -save-temps -fverbose-asm`' to the '`<original-options>`', `moviCompile` will generate richly annotated assembly code that can help when trying to correlate the generated assembly code to the corresponding C or C++ code.

12.3.2 Reducing the Test Source

Whether the sample test source file was automatically generated by the compiler during a crash, or manually as above, you may have sensitive information or Intellectual Property that you do not wish to disclose to Movidius. In this case it will be necessary to "reduce" the test to the minimal information necessary to reproduce the observed problem, while at the same time removing as much sensitive information as possible. This process of reduction is normally performed by the compiler developer team in order to narrow down the problem, but in the case of sensitive information the customer will have to perform this task.

1. Firstly, perform step #2 of the above test preparation task, but omit the option '`-frewrite-includes`'. This will still generate a pre-processed file, but with a lot of additional information omitted such as comments.
2. Edit '`reduced.c[pp]`' and delete all lines that start with '`# [1-9]`'. You can also remove all blank-lines if you like, and this edited version will be used in all subsequent steps
3. Create a simplified set of options by removing all flags from '`<original-options>`' that start with '`-I`', '`-D`' or '`-U`' - these are not used by a file that has already been pre-processed, and will greatly simplify the set of options to use from here on. Remove the reference to the original source code '`<your-path-to>/originalCode.[c|cpp]`', and also remove '`-o <filename>.[o|s|asmgen]`'; append '`reduced.c`' or '`reduced.cpp`' depending on whether the code is C or C++, and I will refer to this as '`<test-options>`'.
4. Invoke `moviCompile` again as follows:

   ```
   <your-path-to>/moviCompile <test-options>
   ```

   This produces a file named '`reduced.s`' which is the generated assembly version for the test-case.
5. Verify that the problem you observed is still present in '`reduced.s`'.
6. Since compilers generally generate code in "function" units, you should identify the C/C++ source for the function for which you have observed the suspect or invalid generated assembly code. If the function is in C, the name is identical in assembly as it is in C ('`static`' functions are preceded with '`.I`'). However, if it is in C++, then the mangled name is in the assembly code, and this starts with the characters '`_Z`'. Copy this name and run the following command (see section 9.10.1):

   ```
   sparc-myriad-rtems-c++filt _<function-name>
   ```

   You might have to prepend an underscore, or remove the leading underscore depending on the version of '`c++filt`'. The Movidius tools contain a pre-built a version of this tool that is located in the '`sparc-myriad-rtems-6.3.0`' tool-chain, and is called `sparc-myriad-rtems-c++filt`, but if you are using Linux or Cygwin, you can generally use the local '`c++filt`'.

At this point you can submit the file '`reduced.c[pp]`' along with '`reduced.s`', the '`<test-options>`' and the exact version of `moviCompile` with the ticket.

12.3.2.1 Further Reduction - Automated

However, after examining the file '`reduced.c[pp]`' you may decide that it still reveals too much information, and that further reduction is necessary. There are some automated tools that can perform this reduction which you can try, for example:

• Delta	http://delta.tigris.org/

- C-Reduce https://embed.cs.utah.edu/creduce/

And either of these can be used to assist you in the reduction process. However, this still may not be sufficient, and at this stage there is no alternative but to use an iterative process to reduce what the file contains.

12.3.2.2 Further Reduction - Manual

Since the compiler generates code a function at a time, you can usually delete the source for all functions in '`reduced.c[pp]`' that follow the function in which you have observed a problem. The next stage is to eliminate as many functions preceding this one as possible. It is not always obvious what functions to remove, so one way of doing this is to "bracket" each preceding function with pre-processor directives, thus:

```
#if 0 // Exclude this function
The function you are excluding
#endif
```

Recompile to ensure that the file still compiles successfully, and if it does verify that the problem in the generated assembly code is still present.

In the event that the compilation fails following the exclusion of a function definition, you can often replace the particular function definition with an '`extern`' declaration for the same function, and then delete the body of the function. Depending on your options, you may need to retain the definitions of inline-functions or small functions that are automatically inlined to preserve the observed problem in the generated assembly code.

When you are satisfied that all unnecessary functions are removed, and the issue is still observable in the generated assembly code, you can go in and remove all blocks that are still guarded by the '`#if 0/#endif`' pair.

Sometimes, to assist in determining what to keep versus what not to keep, it is convenient to change the '`#if 0`' to '`#if 1`' during this iterative process to indicate that the particular code needs to be retained to reproduce the problem.

Following the elimination of the functions that are not required, the same approach can be used to eliminate the declarations, types and data objects that are not required. And following this, you can reduce the body of the function that generates the suspect code until it too is minimized.

At each stage in the iteration, the size and complexity of both the '`reduced.c[pp]`' and '`reduced.s`' files is decreasing. You can continue to perform this reduction process until you are satisfied that the example no longer contains any information that you do not wish to reveal to Movidius support.

12.3.2.3 Further Reduction – Obfuscation

Sometimes, even after exhaustive reduction, the names of types, data and functions may still reveal more than you would like. Since code generation does not depend on the names of functions, types and objects (though the mangled names might), you can generally replace all function names with a simple name. For example, replace each function name in turn with '`f1`', '`f2`', '`f3`' ... '`f`*N*'; and similarly types with '`T1`', '`T2`', '`T3`' ... '`T`*N*' and data in the same fashion. Since C++ will encode the name of a type into the function name (the mangled-name) you may need to find the new '`_Z`' prefixed mangled name. Alternatively, if overloading is not an issue, simply prefix the function declaration with '`extern "C"`' and the name will not be mangled at all.

12.4 Preparing a Test Case

Ideally a test case should be reduced to the simplest self-checking form possible. Since compilers work on a function at a time, it should be possible to identify the function containing the defect. Once the function exhibiting the defect has been identified, and the test reduced, the function should be placed in a source file of its own. If the function has other dependencies required to permit it to be compiled, linked and executed, then these should also be reduced to the minimum necessary and included with the test.

A "Unit Test" can then be constructed as a simple 'main' function that calls the function exhibiting the defect with a contrived sample set of input data. After execution of the function, the set of output data should be compared with a reference set of expected data. The unit test should exist in a separate file so that the compiler does not "see" both the implementation and the use, and does not perform some optimization that masks the problem being observed (e.g. inlining).

For example, if the function exhibiting the observed problem is called 'invert8' and it is supposed to read 8 bytes from the source and write the binary complement of these to the destination, then the unit test might look like:

```
extern void invert8(unsigned char* pDst, const unsigned char* pSrc);

const unsigned char input[8] =
    { 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
const unsigned char expected[8] =
    { 0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F };
unsigned char output[8] = { 0, 0, 0, 0, 0, 0, 0, 0 };

int main() {
  // Call the function with the input data set
  invert8(output, input);

  // Check the actual output against the expected output
  for (int i = 0; i < 8; i++)
    if (output[i] != expected[i])
      return 1;  // Failure

  return 0;  // Success
}
```

Assume the file containing the function exhibiting the defect is called 'invert8.c' and its unit test is called 'invert8_unittest.c', then place these together in a directory with no other source files (or only other files necessary to complete the test).

12.4.1 The Simple Test "Stub"

The moviCompile distribution contains a very simple project skeleton that will build and run a simple program using the conventional 'main' execution model (see sections 10.1.1 and 10.2.1), and will execute this program on SHAVE #0. This project is located in:

```
<MOVITOOLS>/common/moviCompile/testStub
```

Building and running a simple program is done using the provided 'Makefile'. By default it will build and run a sample "Hello World" application located in:

```
<MOVITOOLS>/common/moviCompile/testStub/helloWorld
```

To run this on the simulator just do the following:

```
cd <MOVITOOLS>/common/moviCompile/testStub

make sim
```

This will run build and run the 'helloWorld' project by default, but to run an arbitrary simple program that uses the conventional 'main' execution model, you can specify the path to your program using the variable 'TEST_PATH'.

For example, assume the unit test for the observed defect in ‘`invert8`’ is located in the following directory:

```
<SOMEPATH>/invert8/
```

Then, this can be built and run using the simulator as follows:

```
cd <MOVITOOLS>/common/moviCompile/testStub

make TEST_PATH=<SOMEPATH>/invert8 sim
```

The ‘`Makefile`’ has a small number of useful targets, and several variables that may be changed to adapt for the requirements of your particular program. There is a ‘`help`’ target that describes these targets and the variables:

```
cd <MOVITOOLS>/common/moviCompile/testStub

make                    # Will display all help information
make help               # Will display all help information
make help_targets       # Will just list the targets provided
make help_variables     # Will list the variables which may be changed
make info               # Additional general information
```

When ‘`main`’ returns ‘`0`’ the simple Leon test stub considers the test to have succeeded and will write the following message to the UART:

```
'moviCompile' test passed
```

If a value other than ‘`0`’ is returned from ‘`main`’, then the test is considered to have failed, and the Leon test stub will write the following message to the UART:

```
'moviCompile' test failed with exit code '1'
```

In both cases additional information about the program being tested is output which may be useful.

NOTE: For your convenience, this example is provided with `moviCompile` in the directory:

```
<MOVITOOLS>/common/moviCompile/testStub/invert8
```

12.4.1.1 Runtime Environment and Limitations

This stub is not organized as a real-world Myriad program would be. Instead it is configured to run all code from CMX Slice #0, so the code cannot exceed 128KB.

All data is also configured to reside in CMX:

- 256KB is provided for data, which is placed in CMX Slices #2 and #3.
- The stack is located in CMX Slice #5, and is limited to 128KB.
- The default heap is 768KB and is located in CMX Slices #6 thru #11.

No provision is currently provided for running applications from DDR.

The target ‘`make board`’ is not yet implemented, but in a future release of `moviCompile` this will build and run the sample program on the Myriad development board.

12.4.1.2 Supplementary Object Files

Sometimes when trying to provide an executable unit test to illustrate a particular problem, it is necessary for the function containing the observed defect to use other functions or data which do not have the defect. However, the source for these additional functions or data may contain sensitive or proprietary information that you do not want to share with Movidius. This might include the reference input and expected output data for the unit test, and the routine used to validate the actual output data against the reference expected output data.

In this case the test can provide just the “object” files containing these elements. The ‘`Makefile`’ will include any files with the extension ‘`.o`’ that are present in a sub-directory of the directory containing the test source itself. The sub-directory must have a particular name, as it is dependent on the CPU being selected. These sub-directory names are as follows:

- `ma2100/` - When the CPU is MA2100 {deprecated}
- `ma215x/` - When the CPU is MA2150 {default} or MA2155
- `ma245x/` - When the CPU is MA2450 or MA2455
- `ma248x/` - When the CPU is MA2480 or MA2485

This way it is not necessary to provide all of the elements for the unit test in source form.

13 Appendix B - Math Libraries Performance and Accuracy

The following table describes the expected performance and accuracy of the optimized `SHAVE` math library '`mlibm.a`' for the FP32 32-bit floating-point operations (for `SHAVE` these are for the '`float`' and '`double`' data types); for their FP16 16-bit floating-point equivalents (for `SHAVE` this is the '`__fp16`' or '`half`' data type[32]); and for the FP64 64-bit floating-point equivalents (for `SHAVE` this is the '`long double`' data-type). The data presented below shows the average number of cycles across the entire range of values, and the worst case ULP error for each of the functions named in the first column:

Function	FP32		FP16		FP64	
	Cycles	ULP Error	Cycles	ULP Error	Cycles	ULP Error
acos	203	2	101	0	5,169	1
acosh	232	1	60	0	5,222	1
asin	221	2	96	0	4,168	1
asinh	257	1	185	0	5,192	1
atan	93	1	57	0	3,770	1
atan2	137	1	133	0	3,646	1
atanh	90	1	74	0	5,644	2
cbrt	102	3	64	0	1,279	2
ceil	25	0	26	0	77	0
__clamp	2	0	2	0		
copysign	24	0	28	0		
cos	101	2	96	0	1,790	1
cosh	128	1	72	0	3,152	1
erf	95	1	62	0	1,853	3
erfc	319	12	105	0	4,453	4
exp	98	1	77	0	2,575	1
exp2	66	1	48	0	2,453	2
expm1	110	3	93	0	2,712	2
fabs			25	0		
fdim	42	0	45	0	295	0
floor	31	0	29	0	72	0
fmax	42	0	44	0	51	0
fmin	42	0	44	0	51	0
fmod	87	0	119	0	560	0
frexp			25	0		
hypot	164	1	88	0	1,094	2
ldexp			17	0	103	1
lgamma	435	9601813	305	1	17,506	2
log	86	1	58	0	4,157	1
log10	108	2	60	0	4,099	1
log1p	127	1	96	0	4,863	1
log2	102	1	25	0	4,054	2
logb			28	0		
nearbyintTONEAREST			86	0		
nearbyintTOWARDZERO			80	0		

[32] The data type '`half`' is not a keyword, but rather a '`typedef`' that is available when the header '`<moviVectorUtils.h>`' is included

Function	FP32		FP16		FP64	
	Cycles	ULP Error	Cycles	ULP Error	Cycles	ULP Error
nearbyintUPWARD			90	0		
nextafter	22	1	30	0	164	0
pow	335	14	171	0		
__powr			130	0		
remainder	109	0	123	0	514	0
remquo	123	0			496	0
round	30	0	30	0	89	0
sin	105	2	94	0	1,797	1
sinh	187	1	77	0	3,124	1
sqrt	115	0	22	0	812	1
tan	115	4	115	0	3,775	1
tanh	130	2	78	0	3,335	1
tgamma	359	32	198	0	150	4
trunc	29	0	24	0	62	0

Table 11. Math Library Performance Data

Some entries in the table above are shaded and contain no information; this is because at the time of writing, no data was yet available for these entries.

NOTE: All of the math performance data was measured on the MA2150 processor and optimized at '`–O3`'.

13.1 What is ULP?

ULP means "Unit in the Last Place" and is a commonly accepted measure of the accuracy of functions involving Floating-Point numbers.

Unlike Real numbers in mathematics which cover an infinite continuum of values, the approximation to Real numbers on computers generally uses Floating-Point representations, and because these use a finite number of bits, they do not represent an infinite continuum of values, but rather a collection of discrete values along that continuum.

In simplistic terms, the ULP essentially measures how many values in this collection of discrete values are closer to the best possible value than the value actually computed by the function being measured.

So for instance, a ULP of 0 for a function means that the function always picks the best possible value.

A good article describing the definition and application of ULP can be found on Wikipedia at:

https://en.wikipedia.org/wiki/Unit_in_the_last_place

14 Appendix C – Additional Utilities

`moviCompile` is provided with a small number of additional utilities that may be useful to the programmer. These are standard utilities that are often provided with other distributions of `CLang` and LLVM, but the set of additional utilities provided with `moviCompile` is more limited.

14.1 Scan-Build

Common distributions of `CLang` (or '`clang`') are distributed with the utility '`scan-build`'. This utility provides support for the static analysis of a program by inserting itself into the normal '`make`' build process.

This utility has been adapted for use with `moviCompile`, and also for `sparc-myriad-rtems-gcc` and `sparc-myriad-rtems-g++` which are the C and C++ compilers for the Myriad2 `Leon` processor. The standard LLVM documentation for this is located at the following URL:

http://clang-analyzer.llvm.org/scan-build.html

The '`scan-build`' utility provided with `moviCompile` has been adapted to the requirements of Myriad Application Development. These adaptations comprise of additional command-line options, and additional environment variables that are passed to '`make`' as part of the normal static analysis process.

14.1.1 Myriad Specific Options

A small number of options have been added to '`scan-build`' to support the static analysis of Myriad programs:

`--use-cc [=]`*`<path-to-compiler>`* – this option specifies the path to the C compiler to be used for building the `Leon` parts of the program. This is a standard option to '`scan-build`' but has been adapted for `moviCompile` to default to `sparc-myriad-rtems-gcc`.

`--use-c++ [=]`*`<path-to-compiler>`* – this option specifies the path to the C++ compiler to be used for building the `Leon` parts of the program. This is a standard option to '`scan-build`' but has been adapted for `moviCompile` to default to `sparc-myriad-rtems-g++`.

`--use-mvcc [=]`*`<path-to-compiler>`* – this option specifies the path to the C compiler to be used for building the `SHAVE` parts of the program. This option is specific to the `moviCompile` implementation of '`scan-build`' and defaults to `moviCompile`.

`--use-mvc++ [=]`*`<path-to-compiler>`* – this option specifies the path to the C++ compiler to be used for building the `SHAVE` parts of the program. This option is specific to the `moviCompile` implementation of '`scan-build`' and defaults to `moviCompile`.

`--no-run-number` – the standard implementation of '`scan-build`' places all of the scan analysis results in a sub-directory off the location selected, and which is named using the current date and time, and a sequential index. This is to facilitate concurrent uses of '`scan-build`' whose output is sent to a common root location. However, for many users, this is not necessary as they know that their results are unique and do not need to be differentiated in this way.

The option '`--no-run-number`' is specific to the `moviCompile` implementation of '`scan-build`' and suppresses the use of this intermediate sub-directory.

14.1.2 Myriad Specific Environment Variables

'`scan-build`' works by inserting the static analyser into the normal build process. It does this by replacing the variables '`CC`' and '`CXX`' that are used by '`make`' with a proxy which performs the static analysis and then the normal compilation. In this way it is possible to ensure that the analyser uses the same command-line options when analysing the source file as are used when compiling it normally.

However, Myriad Application Development supports two different CPU architectures (`Leon` and `SHAVE`) and two corresponding different compilers. For this reason the Myriad Software Development environment supplements these standard ‘`make`’ variables with two additional variables: ‘`MVCC`’ and ‘`MVCXX`’; and these are used to identify the compiler to be used when compiling the `SHAVE` C and C++ source files respectively. The standard variables ‘`CC`’ and ‘`CXX`’ are used to identify the compiler to be used when compiling the `Leon` C and C++ source files respectively. The additional variables ‘`MVCC`’ and ‘`MVCXX`’ are used to identify the compiler to be used when compiling the `SHAVE` C and C++ sources files respectively.

When ‘`scan-build`’ is used, these four variables are automatically redefined to point to the appropriate analyser proxy which then performs the static analysis on the source file as appropriate to the normal build process.

14.1.3 Known Issues When Analyzing `Leon` Sources

The static analyser used by ‘`scan-build`’ is actually `moviCompile`, but invoked with a series of internal undocumented options that are not supported outside the context of their implicit use by ‘`scan-build`’. However, `moviCompile` has been designed to compile code for `SHAVE` and not for `Leon`, and this can lead to some false analysis results.

In the majority of cases this should not present many problems, but it is useful to be aware of them when interpreting the results of the static analysis for the `Leon` source code:

- `moviCompile` locates its system header files using the following paths:

  ```
  ../../common/moviCompile/include
  ../../../shared/include
  ```

  relative to the location of the `moviCompile` program itself. However, the system header files that would normally be used by `sparc-myriad-rtems-gcc` are found in the following location:

  ```
  ../sparc-myriad-rtems/include
  ```

  Since `moviCompile` has been designed for `SHAVE` and not for `Leon`, it is unaware of this difference and will use the `SHAVE` system headers during analysis rather than those normally used for `Leon`.

  In most cases this will not matter, as many of the headers are identical, or if not identical, compatible. But there are occasions where the difference may result in some false analysis results.

- The size of ‘`double`’ for `SHAVE` is 32-bits, and identical to the size of ‘`float`’. However, for `Leon` the size of a ‘`double`’ is 64-bits, although the size of ‘`float`’ is still 32-bits. This may result in some false analysis diagnostics where the programmer is using expressions involving ‘`float`’ and ‘`double`’ in a manner which is dependent on this information. Such false diagnostic are rare, but it is worth keeping in mind when interpreting the results of the static analysis of the `Leon` source code.

14.2 `CLang`-Tidy

`moviCompile` provides an implementation of the “`CLang Extra`” utility ‘`clang-tidy`’. This utility supplements ‘`scan-build`’ and the `moviCompile` static analysers to provide additional program health analysis, and in many cases also provides “Fix Support” that will alter the programmers’ code to correct the issues identified by ‘`clang-tidy`’.

There are no `moviCompile` specific changes made to this utility, and the full documentation for this utility can be found at the following URL:

http://clang.llvm.org/extra/clang-tidy

14.3 LLVM-ProfData

moviCompile provides an implementation of the utility ‘llvm-profdata’. This utility processes the data produced by instrumentation enabled by the ‘-fprofile-instrument’ option. This data can be used for conventional profiling analysis and for “Profile Guided Optimisations”, or PGO.

This option is used in conjunction with the ‘mlibprofile.a’ library.

There are no moviCompile specific changes made to this utility, and the full documentation for this can be found at the following URL:

http://llvm.org/docs/CommandGuide/llvm-profdata.html

15 Appendix D – Errata

With the v00.90.0 Build 3000 scheduled release of `moviCompile`, the only known issue at the time of completing this documentation are as follows:

- In section 12.4.1.1 the documentation states that the limit on the stack is 128KB. While this is true, the `Leon` code that invokes '`_EP_start`' incorrectly informs the `SHAVE` execution context that only 4KB is available. This means that if the program enables the stack overflow instrumentation (section 10.6.1), when the program exceeds 4KB of stack use, stack-overflow will be falsely detected and the program will terminate with the '`SHAVEExitStackOverflow`' exit code.

  This will be corrected in the next release of `moviCompile`.

- In section 6.1.5.3 the documentation states that the size of the tag field to '`BRU.SWIH`' has been widened from 5 to 13-bits for MyriadX. While this is true, due to a bug in the Leon hardware only the low 5-bits are visible to the Leon. Programmers are advised not to use values greater than 31 (0x1F).