Netronome Network Flow Processor 6xxx

NFP SDK version 6.1 Preview

Network Flow C Compiler User's Guide

- Proprietary and Confidential -

b3286.dr7273

Netronome Network Flow Processor 6xxx: Network Flow C Compiler User's Guide

Copyright © 2008-2018 Netronome

COPYRIGHT

No part of this publication or documentation accompanying this Product may be reproduced in any form or by any means or used to make any derivative work by any means including but not limited to by translation, transformation or adaptation without permission from Netronome Systems, Inc., as stipulated by the United States Copyright Act of 1976. Contents are subject to change without prior notice.

WARRANTY

Netronome warrants that any media on which this documentation is provided will be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment. If a defect in any such media should occur during this 90-day period, the media may be returned to Netronome for a replacement.

NETRONOME DOES NOT WARRANT THAT THE DOCUMENTATION SHALL BE ERROR-FREE. THIS LIMITED WARRANTY SHALL NOT APPLY IF THE DOCUMENTATION OR MEDIA HAS BEEN (I) ALTERED OR MODIFIED; (II) SUBJECTED TO NEGLIGENCE, COMPUTER OR ELECTRICAL MALFUNCTION; OR (III) USED, ADJUSTED, OR INSTALLED OTHER THAN IN ACCORDANCE WITH INSTRUCTIONS FURNISHED BY NETRONOME OR IN AN ENVIRONMENT OTHER THAN THAT INTENDED OR RECOMMENDED BY NETRONOME.

EXCEPT FOR WARRANTIES SPECIFICALLY STATED IN THIS SECTION, NETRONOME HEREBY DISCLAIMS ALL EXPRESS OR IMPLIED WARRANTIES OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to some users of this documentation. This limited warranty gives users of this documentation specific legal rights, and users of this documentation may also have other rights which vary from jurisdiction to jurisdiction.

LIABILITY

Regardless of the form of any claim or action, Netronome's total liability to any user of this documentation for all occurrences combined, for claims, costs, damages or liability based on any cause whatsoever and arising from or in connection with this documentation shall not exceed the purchase price (without interest) paid by such user.

IN NO EVENT SHALL NETRONOME OR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THE DOCUMENTATION BE LIABLE FOR ANY LOSS OF DATA, LOSS OF PROFITS OR LOSS OF USE OF THE DOCUMENTATION OR FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, EXEMPLARY, PUNITIVE, MULTIPLE OR OTHER DAMAGES, ARISING FROM OR IN CONNECTION WITH THE DOCUMENTATION EVEN IF NETRONOME HAS BEEN MADE AWARE OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL NETRONOME OR ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION, PRODUCTION, OR DELIVERY OF THE DOCUMENTATION BE LIABLE TO ANYONE FOR ANY CLAIMS, COSTS, DAMAGES OR LIABILITIES CAUSED BY IMPROPER USE OF THE DOCUMENTATION OR USE WHERE ANY PARTY HAS SUBSTITUTED PROCEDURES NOT SPECIFIED BY NETRONOME.

Revision History

Date	Revision	Description
September 2018	004	Updated for NFP SDK 6.1 Preview
April 2016	003	Updated for NFP SDK 6.0 Beta 1
January 2014	002	Updated for NFP SDK 5.0
August 2013	001	Initial release

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 About This Guide

This document presents information and language structures specific to the Netronome Network Flow C Compiler for Netronome Network Processors. It also specifies the extensions to the language that support the unique features of the Netronome NFP-6XXX product line.

The guide is organized as follows:

- Chapter 1 : Description of this guide, related documentation, and table of conventions used.

- Chapter 2 : Overview of network processor features and compilation model.

- Chapter 3 : Compiler commands and options.

- Chapter 4 : C language support.

- Chapter 5 : Inline assembly language support.

- Chapter 6 : Optimizations and information that can affect debugging.

- Chapter 7 : Tips for Optimization, Troubleshooting, and Debugging.

- Chapter 8 : Compiler Intrinsics.

- Chapter 9 : Mutual exclusion library.

- Chapter 10 : Semaphore library.

- Chapter 11 : Technical support.

- Appendix A : Qperfinfo Output Information.

> **Note**
>
> For simplicity throughout this document, the compiler will be referred to as the *C compiler*, or in some cases simply the compiler. Also, the Netronome NFP-6XXX network processor may be referred to as *the network processor*.

1.2 Related Documentation

Descriptive Name	Description
Netronome Network Flow Processor 6xxx: Micro-C Standard Library Reference Manual	Specifies the subset and the extensions to the language as well as the intrinsic functions that support the unique features of the Netronome Network Flow Processor NFP-6xxx product line.

Descriptive Name	Description
Netronome Network Flow Processor 6xxx: Microcode Standard Library Reference Manual	Provides a reference to the Microcode Standard Library that supports the Netronome Network Flow Processor NFP-6xxx product line.
Netronome Network Flow Processor 6xxx: Development Tools User's Guide	Describes Programmer Studio and the development tools you can access through Programmer Studio.
Netronome Network Flow Processor 6xxx: Microengine Programmer's Reference Manual	Provides a reference for microcode programming of the Netronome Network Flow Processor NFP-6xxx.
Netronome Network Flow Processor 6xxx: Databook	Contains detailed reference information on the Netronome Network Flow Processor NFP-6xxx.
Netronome Network Flow Processor 6xxx: Network Flow Assembler System User's Guide	Provides information on running the Assembler.
Netronome Network Flow Processor 6xxx: Datasheet	Provides a functional overview of the Netronome Network Flow Processor NFP-6xxx's internal hardware, signals and electrical and mechanical specifications.

1.3 Acronyms

Acronym	Description
ARM	ARM Holding plc
BEQ	Branch if Equal
CLI	Command-Line Interface
CSR	Control and Status Register
CTM	Cluster target memory and part of MEM
DRAM	Dynamic Random Access Memory. This is part of MEM
EMEM	External memory and part of MEM (formerly known as DRAM)
FIFO	First In First Out
GPR	General Purpose Register
IMEM	Internal memory and part of MEM
IPO	Inter-Procedural Optimization
LM	Local Memory
ME	Microengine
MEM	This includes IMEM, EMEM (previously DRAM) and CTM
MSF	Media and Switch Fabric interface - deprecated
NFAS	Network Flow Assembler System
NFCC	Network Flow C Compiler
NFLD	Network Flow Linker

Acronym	Description
NFP	Network Flow Processor
NN	Next Neighbor
PFM	Precision Flow Modeler
SDK	Software Development Kit
SRAM	Static Random Access Memory
XFR	Transfer Register
NFFW	Netronome Flow Firmware

1.4 Conventions Used in this Manual

The following conventions are used in this manual.

Table 1.1. Conventions

keyword=<required value>	Angle brackets show mandatory value that must be supplied.
...	Ellipsis indicates that an item may be repeated.
[Option]	Items in square brackets are optional.
[Option1...]	Optional items can have multiples. The equivalent of [Option1 [Option2]...]
Option=1..5	Range of allowable values. Equivalent to Option=1, 2, 3, 4, or 5
Command1|Command2|Command3	(For Windows*) Selecting cascading options. Indicates that you should follow these steps: • Click on Command1, which offers options including Command2 • Click on Command2, which offers options including Command3 • Click on Command3. For example: **Start|Programs|Accessories|Command Prompt**
SDK x.y.z	The x.y represents the current version, and z the latest point release of SDK that is installed on your system. This could be 4.5.2, for example.
NFP6000 file	Keyboard input, keywords and code items are shown in *monospaced font.*
Netronome NFP-32XX product line	The family of Netronome NFP-32XX network processors, where 32XX=the four-digit designator of the target chip.
Netronome NFP-6XXX product line	The family of Netronome NFP-6XXX network processors, where 6XXX=the four-digit designator of the target chip. Currently it is 6000
Adobe* Acrobat*	An asterisk at the end of a word or name indicates it is a third-party product trademark.

1.4.1 Version-Specific References

Wherever version-specific files or commands are shown in this book, versions are shown as x and point release numbers are shown as y, as in the following example:

Example 1.1. A typical compile command could take this form

```
nfcc -chip nfp-6xxx -Qnctx=1 -I\NFP_SDK_x.y.z\components\standardlibrary\microc\include
```

Where x.y.z = the current software release that is installed on your system.

1.5 Data Terminology

The following data terminology is used in this guide:

Table 1.2. Data Terminology

Term	Words	Bytes	Bits
Byte	½	1	8
Word	1	2	16
Longword	2	4	32
Quadword	4	8	64

> **Note**
>
> The *Netronome Network Flow Compiler LibC Reference Manual* details the data types and their naming conventions supported by the compiler.

2. Overview

The Netronome NFP-6xxx network processor contains an ARM11 processor and up to 120 microengines.

Table 2.1. Number of Microengines in the Netronome NFP-6xxx processor

Processor Type	Number of Microengines
NFP-6000	120

This manual is concerned only with compilation of application programs for microengines. Each microengine has hardware support for up to 8 contexts with zero latency task switches. The microengine can be set up to run 4 contexts instead of 8 by setting a bit in the CTX_ENABLE CSR. In this mode, the 4 contexts that are enabled are contexts 0, 2, 4, and 6.

The compiler focuses on one given microengine. The C language that is implemented is an explicitly parallel C language with threading and synchronization exposed at the language level.

When a program is executed on a microengine, all threads of that microengine execute the same program. Therefore, each thread has a private copy of all variables and data structures in memory. Consider the following program fragment:

Example 2.1. Threads with private copies of variables

```
__declspec(imem) int x;
    int a;
    x = 1;
    while (1) {
        a = 5;
        x += a;
        }
```

All the threads that execute this program fragment get their own private copies of the variables a and x (variable x is allocated to a different IMEM location for each thread). Each thread also executes its own private copy of the while loop.

2.1 Features

- The C compiler provides a high-level language programming environment for the network processor, which speeds up application development time by reducing the amount of architectural detail which the programmer must manage.

- The compiler supports programming for the network processor microengines by supporting a combination of the standard C language, language extensions, and intrinsic functions. It supports unique features of the processor through language extensions, intrinsic functions, and inline assembly.

- All existing reference designs currently written in assembly language can be converted to C for the compiler to handle.

- The compiler works with existing tools, including the microcode linker loader and the Programmer Studio.

2.2 General Characteristics and Limitations

- The C compiler is not a complete ANSI C implementation.
- There are many features of ANSI C, for example floating point operations, that are outside the realm of applications on the network processor architecture. The compiler omits these features.
- The compiler may not compile existing general purpose C code.
- The compiler does not support the full standard C runtime library. It implements useful or necessary functions according to the C runtime library specification, but it does not fully implement the library.
- The compiler does not implement automatic parallelization of code. It expects explicitly multithreaded code as input.
- The compiler does not support separate compilation and linking of C code with assembler code.
- The compiler does not support C++.
- The compiler does not support floating point data types (float or double).

2.3 Compilation Model

Since microengine programs are necessarily very small, the C compiler always compiles the entire program for a microengine. This is done through the Inter-Procedural Optimization (IPO) feature of the compiler. In this model, you can separately compile C source code, but instead of generating code, the compiler writes the intermediate language to an object file. Then when you link the program, the driver calls the compiler for all of the precompiled objects. The compiler performs a global analysis and calls the code generator for each function in the complete program. At this point, the entire call graph and global usage of all variables are known, allowing for much better code generation than would be possible compiling one function or even one source module at a time.

This compilation model allows the compiler to optimize the linkage between functions based on its knowledge of both the caller and called function. The compiler can allocate static variables to registers, fine tune calling sequences, and avoid the stacking of return addressing and the saving and restoring of registers across calls except when absolutely necessary.

2.3.1 Number of Contexts

When 4 or fewer contexts are needed in the microengine program, it is beneficial to set the microengine to run in 4-context mode. This is done using the compiler option -Qnctx_mode=4. In this mode each context has twice as many context relative registers available, potentially leading to fewer spills and better performance. The exact number of contexts also needs to be specified through a command-line option. It is important to remember that in 4 context mode, the contexts are numbered 0, 2, 4, 6. This impacts any code that checks the context number. For example, in 4- context mode the following line of code never evaluates to true:

```
if (__ctx() == 1)
```

2.3.2 Inlining

The compiler performs inlining of functions. This feature of the compiler is controllable through compiler options as well as through the use of directives in the C source code. The keywords __inline and __forceinline appearing in the function definition (as shown in the example below), indicate to the compiler that the function is to be inlined in all the places it is called from. The __forceinline keyword forces the compiler to inline the function regardless of the size of the function as long as inlining has not been turned off via the -Ob compiler option or in debug code via the -Od option. The __inline keyword allows the compiler to decide whether or not to inline the function based on cost/benefit analysis performed by the compiler.

Example:

__forceinline int foo() {...}

__inline int bar() {...}

The function to be force inlined with a __forceinline keyword should normally be in the same file as the caller to the function. If however the caller and callee are in different files, either a prototype for the function with the "extern" keyword should be present (or included) in the file containing the callee, or the function definition should specify the "extern" keyword in addition to the __forceinline keyword.

Example:

__forceinline extern int foo() {...}

Without this, the force inlined function is treated as if it were a static function and hence is not externally visible or inlined in other files.

The keyword __noinline can be used to prevent the compiler from inlining a function. This can be used to control code size.

> **Note**
>
> Use the -Obn compiler option to control the amount of automatic inlining the compiler will perform. See Table 3.1 for more information on the -Obn compiler option.

---

3. Compiling, Linking, Running

3.1 Running the C Compiler

You can compile C source code in one of two ways, by using one of the following:

- Compiler command-line interface (CLI)
- Programmer Studio

3.1.1 The Command Line

You can use the compiler command-line interface from a command prompt window on your system. To compile source code with the command-line interface, do the following:

1. Open a command prompt window.

2. Go to the folder containing the C source files, typically:

   ```
   C:\NFP_SDK_5.y.z\bin
   ```

3. Invoke the C compiler using this command:

   ```
   nfcc -chip <processor> [options] filename...
   ```

   > **Note**
   >
   > You must specify the `-chip <processor>` option when you use the `nfcc` command. If no `-chip <processor>` option is passed to nfcc, `-chip nfp-6xxx` will be passed to the compiler.

The name that you assign to a source file is usually determined by your organization's naming conventions and policies. Generally, it is useful for a filename to be easily recognizable and intuitively connectable with the function of the file. For example, whereas a file name like `AA345.c` is relatively meaningless, file names like `SwitchBostonNE4.c` and `string.h` are more intuitive.

> **Note**
>
> `C:\NFP_SDK_5.y.z\bin` should be on your PATH. If you want to run the V5.y.z C compiler from the command line, the system path needs to be set up accordingly.

3.1.2 Supported Compilations

Two kinds of compilations are supported:

- Compile one or more source files (`*.c`, `*.i`) into separate object (`*.obj`) files.

- Compile any combination of source file (`*.c`, `*.i`) and/or object file (`*.obj`) into one list file (`*.list`).

In the first case, you must use the `-c` option in the command line in order to compile `.c` files into separate `.obj` files. You might want to use this method to compile `.c` files that do not change very often, for example, `rtl.c`, so that you do not have to recompile them every time you make a `.list` file.

Example:

```
nfcc -c file1.c file2.i
```

In the second case, do not use the -c option. In the following example, two source files (`.c` and `.i`) and an object file (`*.obj`) are compiled to produce a `.list` file.

Example:

```
nfcc file1.c file2.i rtl.obj
```

3.1.3 Supported Compiler Options

Table 3.1 lists and defines all the supported C compiler command-line options. The Command-Line Interface (CLI) warns and ignores unknown options. The CLI honors the last option if it conflicts with a previous one, for example,

```
nfcc -c -O1 -O2 file.c
```

generates the following warnings and proceeds:

```
nfcc: Command line warning: overriding '-O1' with '-O2'
```

If you enter other conflicting options such as -E and -EP, the last option entered always prevails.

Options that do not take a value argument, such as -E, -c, etc., are off by default and are enabled only if specified on the command line.

Table 3.1. Supported CLI Options

Option	Definition
`-? -help --help`	List the available options.
`-newgen`	Use the new generation compiler (by default, the old compiler is used for compatibility). This option may be specified as `-ng` also.
`-c`	Compile only: for each `.c` or `.i` file, produce a `.obj` file but do not produce the final `.list` file.
`-compat32`	Use NFP-32xx compatible modes and settings.
`-Dname[=value]`	Specify a `#define` macro. The value, if omitted, is 1.
`-E`	Preprocess to stdout.
`-EP`	Preprocess to stdout, omitting #line directives.

Option	Definition
`-P`	Preprocess to a `.i` file.
`-Fo<file> -Fo<dir>`	Specify name of object file, or object directory (for directory, include a trailing slash).
`-Fe<file>`	Set base name of executable. Defaults to the base name of the first source or object file specified on the command line followed by the extension (`.list`).
`-Fi<file>`	Override the base name of the `.ind` file.
`-FI<file>`	Force inclusion of a file.
`-chip <chip_id>` (where `<chip_id>` is a generic or specific chip SKU.)	Specify the target processor. Chip nfp-6xxx is the default. The compiler adds `-D__NFP_IS_6000` or `-D__NFP_IS_3200` as appropriate. To list the supported processors, use the `-chip_list` option.
`-chip_list`	Print a list of possible chip targets.
`-I path[;path2...]`	Path(s) to include files, prepended before path(s) specified in environment variable `NFCC_INCLUDE`.
`-On`	Optimize for: • n=1: size (default) • n=2: speed • n=d: debug (turns off inlining and many optimizations, overriding `-Obn` below).
`-Obn`	Inlining control: • n=0: none • n=1: explicit (inline functions declared with `__inline` or `__forceinline` (default)) • n=2: any (inline functions based on compiler heuristics, and those declared with `__inline` or `__forceinline`).
`-Qapp_metadata=<string>`	Inserts the specified string into the `.list` file.
`-Qdefault_sr_channel=<0...3>`	Specify the SRAM channel that should be used when allocating compiler-generated SRAM variables and variables that are specified as `__declspec(sram)`. The default is channel 0.
`-Qerrata`	Report when the compiler-generated code triggers a known processor erratum.
`-Qip_no_inlining`	Turns off all inter-procedural inlining. Inter-procedural inlining is on by default.
`-Qliveinfo`	Equivalent to `-Qliveinfo=all`
`-Qliveinfo=gr,sr,...`	Print detailed liveness information for a given set of register classes: • gr: general purpose registers • xr: SRAM/xfer read registers • xw: SRAM/xfer write registers

Option	Definition
	• xrw: SRAM/xfer read/write registers
	• dr: DRAM read registers
	• dw: DRAM write registers
	• drw: DRAM read/write registers
	• nn: neighbor registers (only when -Qnn_mode=1)
	• sig: signals
	• all: all of the above
`-Qlm_start=<n>`	Provides a means for user to reserve some low-numbered (from 0) local memory addresses (in longwords) for direct use in inline assembly. The compiler does not allocate any variables to this address range. Note: Setting `-Qlm_start=1024` reserves all the local memory, and thus disables local memory usage by the compiler.
`-Qlm_size=<n>`	Provides a means for user to reserve some high-numbered (up to 1023) local memory addresses (in longwords) for direct use in inline assembly. The compiler does not allocate any variables to this address range. Note: Omitting, or setting `-Qlm_size=0` allows all local memory from `lm_start` to be used by the compiler.
`-Qlm_unsafe_addr`	Disables the compiler's use of local memory auto increment addressing. Used when user code writes local memory pointers with invalid values. See Section 6.2.6 for more information.
`-Qlmpt_reserve`	Reserves local memory base pointer `l$index1` for user inline assembly code.
`-Qlmpt_reserve_option=n`	Reserves zero or more local memory base pointers for user inline assembly code. • n=0: Reserve none • n=1: Reserve `l$index0` • n=2: Reserve `l$index1` • n=4: Reserve `l$index2` • n=8: Reserve `l$index3`
`-Qmapvr`	Prints out pseudo-assembly code with annotations that map physical registers to user variables and compiler-generated temporary variables.
`-Qnctx=<1, 2, 3, 4, 5, 6, 7, 8>`	Specifies the number of contexts that will be made active in your program. Unused contexts will be made to execute the `ctx_arb[kill]` instruction and terminate. Compiler-allocated resources such as memory will not be allocated to unused threads. The underlying number of contexts supported by the hardware will not be changed, so hardware-managed resources such as registers will still be allocated to all threads. Defaults to the value of `-Qnctx_mode` (which defaults to 4). If `-Qnctx` is set greater than the value of `-Qnctx_mode`, `-Qnctx_mode` will be changed to the higher value.

Option	Definition
`-Qnctx_mode=<4, 8>`	Specify the number of contexts that the hardware should support. Changing this value from 4 to 8 halves the number of available context specific registers. Defaults to 4.
`-Qnn_mode=<0, 1>`	Sets `NN_MODE` in `CTX_ENABLE` for setting up next neighbor access mode. • 0=neighbor (default) • 1=self.
`-Qnolur=<func_name>`	Turns off loop unrolling on specified functions. You can supply one or more function names to the option. For example: • -Qnolur="_main"; turn off loop unrolling for main(). • -Qnolur="_main,_foo"; turn off loop unrolling for `main()` and `foo()`. The supplied function name must have the preceding underscore ('_').
`-Qpe_dma_dual_signal`	Sets the compiler to provide for the second, optional signal for MU PE DMA commands.
`-Qperfinfo=n`	Prints performance information. It is possible to specify multiple options by repeating the -Qperfinfo option several times. • n=0: No information (similar to not specifying) • n=1: Register candidates spilled (not allocated to registers) and the spill type • n=2: Instruction-level symbol liveness and register allocation • n=4: <deprecated> • n=8: Function sizes • n=16: Local memory allocation • n=32: Live range conflicts causing IMEM spills • n=64: Instruction scheduling statistics • n=128: Warn if the compiler cannot determine the size of a memory I/O transfer • n=256: Display information for "restrict" pointer violations • n=512: Print offsets of potential jump[] targets • n=1024: Information about the Boolean propagation optimization • n=2048: Register requirements report • n=4096: Information on switch statement optimizations • n=8192: Print information on I/O parallelization.
`-Qerrinfo=n`	Prints supplementary error information. • n=0: No information (similar to not specifying) • n=1: LM allocation failure

Option	Definition
`-Qrevision_min=n` `-Qrevision_max=m`	The version arguments allow the compiler to generate code that works on a range of processor versions (steppings). • 0x00=A0 (default for -Qrevision_min) • 0x01=A1 • 0x10=B0 • 0x11=B1 The default revision range is 0x00 to 0xff (all possible processor versions). The default for `-Qrevision_max` is 0xff. The compiler adds `-D__REVISION_MIN=n` and `-D__REVISION_MAX=m`. Note: The Netronome network processor program loader reports an error if a program compiled for a specific set of processors is loaded onto the wrong processor.
`-Qspill=<n>`	Selects the alternative storage areas ("spill regions") chosen when variables cannot be allocated to general-purpose or transfer registers: (LM=local memory, NN=next neighbor registers, CLS=Cluster Local Scratch) • n=0: LM (most preferred) → NN → IMEM (least preferred) • n=1: NN→LM→IMEM • n=2: NN only; halt if not enough NN • n=3: LM only; halt if not enough LM • n=4: NN→LM; halt if not enough LM or NN • n=5: LM→NN; halt if not enough LM or NN • n=6: IMEM only • n=7: No spill; halt if any spilling required • n=8: LM→IMEM • n=9: NN→LM→CLS→IMEM • n=10: LM→NN→CLS→IMEM • n=11: CLS only • n=12: LM→CLS Default is n=0. You must set `-Qnn_mode=1` to use the NN registers as a spill region. If the NN registers are used by program code, NN spilling will be automatically disabled.
`-Qspill_cls_limit=<n>`	In the event that spilling is enabled with cluster local scratch as possible spill region with IMEM following, the spill limit n will dictate how many bytes are allocated in cluster local scratch for spilling. Default is n=65536 and can be a value between 0 and 65536.
`-Wn where n=<0, 1, 2, 3, 4>`	Warning level: (default value is 1) • 0: print only errors • 1, 2, 3: print only errors and warnings

Option	Definition
	• 4: print errors, warnings, and remarks.
-WX	Turns warnings into errors.
-Wextra	Generates extra diagnostics: mostly intended for debugging.
-Zi	Produces debug information. The compiler generates a file with a .dbg extension for each source.

3.1.3.1 Environment Variables

The following environment variable is recognized by the compiler:

NFCC_INCLUDE: A list of directories to be added to the include path. The list is separated by semicolons: dir1;dir2;dir3..., and is appended after the directories supplied on the command line using -I.

3.1.3.2 Predefined Macros

3.1.3.2.1 NFP SDK C or Assembler Predefined Processor Type and Revision Macros

As the IS_NFPTYPE mechanism (described in the next section) can only be used with assembler (i.e. microcode), an alternate mechanism to determine the processor type can be used. The mechanism can be used in Micro-C source code, assembler source code, and .IND script files. This mechanism relies on the fact that the NFP SDK will define the macro __NFP_IS_XXXX when compiling C code, assembling microcode, or running .IND scripts associated with NFP-xxxx processors. In addition to this, the processor variant can be determined, as either __NFP_IS_3200 or __NFP_IS_6000 will be defined. See following code:

Example 3.1. Different Code Sequences for Netronome NFP Devices and Other (e.g. Intel IXP) Devices

```
#ifdef __NFP_IS_3200
   // Micro-C code, assembler code, or .IND script code for NFP-32xx processors
#endif
#ifdef __NFP_IS_6000
   // Micro-C code, assembler code, or .IND script code for NFP-6xxx processors
#endif
```

3.1.3.2.2 Additional Predefined Macros

Table 3.2. Additional Predefined Macros

Macro	Description
__NFCC_VERSION	This is defined to a string indicating the version of the C compiler. It is in the form M.m.b where M, m, and b are

Macro	Description
	integers and M is the major version, m is the minor version, and b is the build number.
__NFCC_MAJOR_VERSION	This is defined to an integer indicating the major version of the C compiler.
__NFCC_MINOR_VERSION	This is defined to an integer indicating the minor version of the C compiler.
__NFCC_BUILD_NUM	This is defined to an integer indicating the build number of the C compiler.
__NFP_TOOL_NFCC	This is defined when code is being processed by NFCC, and not another tool. It can be used to for example permit code to be either preprocessed by a standalone preprocessor or compiled with NFCC, with `#ifdef` directives being used to ensure that the appropriate code is emitted in each case. (NFP SDK tools other than NFCC will define similar macros, e.g. `__NFP_TOOL_IND` and `__NFP_TOOL_NFAS` are defined by the IND script interpreter and NFAS respectively.)
__NFP_LANG_MICROC	This is defined when the file being processed should be written in the Micro-C language, e.g. is a `.c` file, or when an include file, e.g. a `.h` file, should conform to C language syntax and conventions. It can be used to, for example, ensure that appropriate constructs are used in files that are included by Micro-C, assembler, and even .IND script files. (NFP SDK tools other than NFCC will define similar macros, e.g. `__NFP_LANG_IND` and `__NFP_LANG_ASM` are defined by the IND script interpreter and NFAS respectively.)
__NFP_THIRD_PARTY_ADDRESSING_32_BIT	This is defined when 32 bit addresses for third party transfers are being used.
__DATE__	ANSI standard macro defined to a string indicating the date that the C compiler was invoked. It is in the form: Mmm dd yyyy, where Mmm is the first three letters of the month name, e.g. "Jan", dd is the date with a leading zero, and yyyy is the year.
__TIME__	ANSI standard macro defined to a string indicating the time that the C compiler was invoked. It is in the form hh:mm:ss, where hh is the hour, mm is the minutes, and ss is the seconds.

3.1.4 Deprecated Compile Options

The following compile options are supported but officially deprecated:

`-Qno_decl_volatile` Shared variables are normally treated as volatile by the compiler. This option causes such declarations not to be flagged as volatile, with the user taking responsibility for ensuring that variables are correctly evaluated after a context swap.

3.1.5 Input and Output File Types

Table 3.3. Input File Types

Extension	File type
`.c`	Source file
`.h`	Header file
`.i`	Source file after preprocessing
`.obj`	Object generated by the compiler invoked by the -c option

Table 3.4. Output File Types

Extension	File type	Command option
`.list`	Output file from compiler used by linker	-Fe<file>
`.obj`	Object generated by the compiler	-c and -o
`.nffw`	Linker output file	-o, -elf32 or -elf64
`.ind`	A Precision Flow Modeler (PFM) script to assemble and run the program	-Fe<file> or -Fi<file>

3.1.6 Linking a Microengine

Before running a program you must link using the nfld linker. The linker can combine multiple files into a single executable NFFW file. The executable file can contain microcode and data for one or more microengines.

Example: To link a single microengine program to run on microengine 0 in island 32, use

```
nfld –u i32.me0 file.list –o file.nffw
```

Table 3.5 lists and defines all the supported C linker command-line options. The CLI ignores and issues a warning for unknown options.

Table 3.5. Linker Command Line Options

Option	Definition
-h	Print a description of the nfld commands.
-res.<mem>.base byte_address	Define the base address from which the linker may allocate symbols for the resource specified by <mem> which can be iX.{emem,imem,ctm,cls}, where iX can also be an island alias..

Option	Definition
-res.\<mem\>.size byte_size	Define the size of the resource from which the linker may allocate symbols for the resource specified by \<mem\> which can be iX.{emem,imem,ctm,cls}, where iX can also be an island alias..
-o outfile	Creates the output ELF64-format NFFW file. The default output file is the name of the first `.list` input file with a file type of `.nffw`.
-g	Include debugging information, to be used by Programmer Studio, in the output object file.
-seg file	Creates a 'C' header file defining the variables memory segments.
-u meid [meid...] [list_file]	Associates a list of microengines to subsequent list_file. The meids are of the form iX.meY, so i32.me0 refers to the first microengine in island 32. All microengines are assigned by default.
-codeshare meid1 meid2	Share codestore of microengines meid1 and meid2. meid1 must be an even numbered microengine and meid2 must be (meid1 + 1).
-l	Used together with a preceding -u. The –l switch is used to disambiguate between microengine number and `*.list` file path. In earlier versions, if a `*.list` file name started with a number, that number would be parsed as a microengine number rather than as part of the `*.list` file name. Using the –l switch permits `*.list` file names that begin with a number.
-v	Print a message that provides information about the version of the Linker being used.
-map [file]	Generate a linker `.map` file. The generated filename is the same as the `NFFW` file but with the extension `.map`. The `.map` file contains the symbols and their addresses.
-noecc	Disable codestore ECC (correction/detection) in output image.
-nn_chain.\<island\> {A,B}	Set an island's next neighbor chain mode: • A - normal NN chaining • B - alternate NN chaining Default is A. See linker help for more information.
-mip	Insert a default MIP in the NFFW output image if none exists.
-rtsyms	Insert a runtime symbol table in the NFFW output image.

3.1.7 Example – Using the C Compiler

A simple C program, `hello.c`

3.1.7.1 The C File

`Hello.c` looks like this:

```
#include <nfp.h>
```

```
main()
{
    if(__ctx() == 0) {
        volatile unsigned __declspec(mem) * p;
        p = (void *) 0;
        *p = 0xcafef00d;
    }
}
```

3.1.7.2 Compiling the File

To compile, the command line looks like this:

```
nfcc -chip nfp-6xxx -Qnctx=1 -I\NFP_SDK_5.y.z\components\standardlibrary\microc\include \
        hello.c \
        \NFP_SDK_5.y.z\components\standardlibrary\microc\src\rtl.c \
        \NFP_SDK_5.y.z\components\standardlibrary\microc\src\intrinsic.c
```

or if the rtl and intrinsic files are pre-compiled:

```
nfcc -chip nfp-6xxx -Qnctx=1 -I\NFP_SDK_5.y.z\components\standardlibrary\microc\include \
        hello.c \
        \NFP_SDK_5.y.z\components\standardlibrary\microc\lib\rtl.obj \
        \NFP_SDK_5.y.z\components\standardlibrary\microc\lib\intrinsic.obj
```

> **Note**
>
> Use the slash character (/) in place of the backslash character (\) as the directory delimiter under the Linux operating system.

3.1.7.3 Linking the File

To link, enter the following command on the command line:

```
nfld -u i32.me0 hello.list
```

3.1.8 C Compiler Graphical User Interface from Programmer Studio

The Programmer Studio supports an integrated compiler for creating, compiling, testing, and debugging network processor applications. For more detail on how to configure and use NFCC within Programmers Studio, refer to the *Netronome NFP Development Tools User's Guide*.

3.1.8.1 Build Features

The Programmer Studio supports the following build features:

- Support for creating, editing, and managing C source files.
    - Inserting C sources into project.
    - Dependency checking for `.c` and `.h` files.
    - Syntax coloring for `.c` and `.h` files.
    - FileView for `.c` files.
- Support for setting up projects to include producing microstore images from C source files and Assembler `.uc` files.
    - Build setting/compiler dialog.
    - GUI control for specifying compile options.
    - Support for include paths for C sources separate from Assembler include paths.
    - Support for setting target paths for images.
    - GUI for preprocessor macro definition to support #ifdef.
    - GUI controls for specifying link options.
- Support for building microstore images from C source files and Assembler `.uc` files.
    - Running compiler and providing parameters.
    - Displaying and interpreting output messages.
    - Relating errors and warnings to source lines.
- Support for persisting project and option files.
    - Persist new project data in dwp file.
    - Persist new option data in dwo file.

3.1.8.2 Debug Feature

The compiler supports source level debugging. However, when compiling for debug, optimizations are generally turned off.

- Support for source-level debugging.
    - Display of register and variable values based on scope and live-range.
    - Display of memory variables.
    - Display of values as C structures.
    - Optional expanded display of assembly instructions generated by each C source statement.
    - Single-stepping based on C source statements or expanded assembly instructions.
    - Setting breakpoints.
    - Run to cursor.
    - Current instruction markers.
    - Execution coverage.
    - Thread history.

- Data watches.

- Go to source.

3.1.8.3 Running and Debugging Under the Programmer Studio

To create and run a project under the Programmer Studio, perform the following steps:

1. Start the Programmer Studio.

    Depending on how you installed it, you can usually click **Start** on the Task bar and then select **Programs**→ **NFP SDK 5.y.z**→**Programmer Studio.**

2. On the **File** menu, click **New Project**.

3. Enter the location and project name, select the appropriate chip type. Then click **OK**.

4. Click the **Build**→ **Settings** menu, and select **General** tab. Add include directories specific to your application. Include files in C:\NFP_SDK_x.y.z\components\standardlibrary\microc\include will also be found if **Include standard library paths** is checked (which is the default).

5. Select **Compiler** tab and click **Compiler**→ **Insert compiler source file**. By default, the files rtl.c, intrinsics.c and libc.c that are shipped with the NFP SDK are compiled with the application. These files supply intrinsic functions as well as string functions. Only add these files manually if you need to use custom versions of the files.

6. Select the .c file(s) needed for your project and click **Insert**.

7. Click **Choose source files** and select the needed source files for this program and click **OK**.

8. Click **New .list file** to specify the path and name of the output (.list) file.

9. Click the **Linker** tab.

10. Click the browse button to the right of the **NFFW file** box.

11. Enter the name of the NFFW file you wish to create.

12. Click the **List File Assignments** . From the list on the right select the appropriate **ME** and select the file you specified in Step 7 and then click +. Then click **OK**.

13. On the **Build** menu, click **Build**.

14. If you get no errors, you can start debugging by selecting **Start Debugging** on the **Debug** menu.

3.1.9 Compiling and Linking on Linux

A Linux version of the compiler and linker is available for use from the command-line. The instructions and options in Section 3.1.1 to Section 3.1.7 still holds true for the Linux version, except that paths need to be specified in the Linux forward slash (/) notation.

3.1.10 Mixed C and Assembly Language Compilation

The compiler driver provides some support for mixed C and assembly language compilation. At present, this is restricted to allowing "codeless" .uc files -- that is, files acceptable to the microcode assembler when using the -codeless option -- to be specified along with regular Micro-C files. The linker directives produced by the assembler are then transparently translated into an equivalent C form through the use of #pragmas.

This feature is useful for specifying linker-oriented pseudo-ops that the compiler does not otherwise support, or that that may rely on special features of the microcode assembler. It cannot, however, be used to include separately-assembled microcode in Micro-C programs.

4. C Language Support

4.1 Orientation

One thing that distinguishes Micro-C from standard C is the extensive use of `__declspec` directives. Micro-C declspecs fill the same kind of general role as standard C's `const` and `volatile`: they act as declaration specifiers and type qualifers.

Syntactically, a declspec directive consists of the keyword `__declspec` followed by, in parentheses, one or more declspec modifiers, optionally comma-separated. Multiple declspec directives can strung together. Examples follow:

```
__declspec(cls addr32) int foo;
__declspec(cls, addr32) int bar;
__declspec(cls) __declspec(addr32) int baz;
```

Table 4.1 lists the primary declspec modifiers. These are documented in more detail in following sections. The many NFP MU and CLS allocation types (for example `emem` and `i32.cls`) are also valid declspec modifiers.

Table 4.1. Declspec Modifiers

Modifier	Refer to	Description
addr32	Section 4.3.3	32-bit address
addr40	Section 4.3.3	40-bit address
aligned	Section 4.2.7	alignment
atomic	Section 4.3.9	atomic variable
dram	Section 4.3.3	MEM (legacy)
export	Section 4.3.5	global data
gp_reg	Section 4.3.2	general purpose register
import	Section 4.3.5	global data
local_mem	Section 4.3.3	local memory
mem	Section 4.3.3	generic MEM
nn_local_reg	Section 4.3.2	next neighbor register
nn_remote_reg	Section 4.3.2	remote next neighbor register
packed	Section 4.2.7	packing of structures
packed_bits	Section 4.2.7	packing of bit fields
read_reg	Section 4.3.2	read transfer register
read_write_reg	Section 4.3.2	read-write transfer register
remote	Section 4.4	reflector I/O

Modifier	Refer to	Description
scope	Section 4.3.5	global data allocation
shared	Section 4.3.2	shared data
signal	Section 4.3.7	signal variable
signal_pair	Section 4.3.7	signal pair struct
sram	Section 4.3.3	SRAM (legacy)
unaligned	Section 4.2.9	alignment
visible	Section 4.4	reflector I/O
write_reg	Section 4.3.2	write transfer register
xfer_read_reg	Section 4.3.2	read transfer register
xfer_read_write_reg	Section 4.3.2	read-write transfer register
xfer_write_reg	Section 4.3.2	write transfer register

4.2 Standard Data Types

4.2.1 Basic Data Types

The compiler supports the following standard scalar data types:

- char 8-bit signed and unsigned

- short 16-bit signed and unsigned

- int 32-bit signed and unsigned

- long 32-bit signed and unsigned

- long long 64-bit signed and unsigned

- enum 32-bit signed and unsigned

- 32-bit pointers typed by memory type

- 40-bit mem pointers typed by memory type.

As per the C standard, chars are the smallest addressable units, and pointers to successive chars differ by one. The compiler supports chars and shorts and pointers to them, although at some potential performance cost. Users are recommended to avoid usage of char and short when possible, because access of quantities less than 32 bits (64 bits in MEM) generally involves additional operations to extract the appropriate bytes from the longword or quadword. Access through pointers to 8-bit and 16-bit types may also require runtime alignment of data, which is even more inefficient.

4.2.2 Pointer Representation

All pointers are represented as byte addresses irrespective of the memory region pointed to. The compiler keeps track of the memory region that a pointer can point to and issues error messages on inconsistent use. For example, assignment of a pointer to CLS to a pointer to MEM will be flagged as a user error. Pointers with no specified memory region are assumed to point to IMEM. Further, when performing pointer arithmetic, the compiler will modify the byte value by the appropriate value. For example, when incrementing a pointer to a long long by one, the compiler will add 8 to the pointer value. If the pointer is pointing to a user defined data type, the compiler will also do the right thing. See Section 4.2.9 for more information on how the compiler handles alignment issues.

4.2.3 Bitfields

The compiler implements arrays and structs. Since the Netronome NFP-6xxx network processors handle packets of communication data that are often defined in terms of bit fields, the compiler supports efficient manipulation of structs with bit fields. Bit fields are supported using standard C syntax. The compiler also supports packed bit-fields through __declspec(packed_bits) as described in Section 4.2.7. With this declspec, structures containing bitfields are laid out such that there is never any padding inserted between a bit-field and its previous member in the structure. __declspec(packed_bits) is helpful in mapping packet header structures accurately onto C structures. See Section 4.2.7 for more information on packing bit fields within a structure.

4.2.4 Floating Point Types

The compiler does not support floating point types. The lack of hardware support and limited code space make it virtually impossible to provide any floating point support-nor is any needed for the type of applications envisioned.

4.2.5 String Literals

String literals are placed into MEM and accessed through a pointer to MEM. It is an error to use a string literal in a position which expects a pointer to a non-MEM memory region, unless a static initialization of a character array is being performed.

Example:

```
void foo(__declspec(cls) char *str_in_cls) { ... }

foo("string"); // ERROR: "string" is in IMEM and cannot be passed to foo()

foo((__declspec(cls) char *)"string");    // RUNTIME ERROR: address of "string" is
not a valid CLS address
{
  __declspec(cls) char *ptr = "string";    // ERROR: "ptr" must be a character
array
```

```
  __declspec(cls) char arr[7] = "string";  // CORRECT: static initialization of
character array
  foo(arr); // CORRECT: type of parameter matches
type of argument
}
```

4.2.6 Size of Data Types

The size of data types, as reported by the sizeof() built-in function, are in bytes, thus:

```
sizeof(int) == 4
sizeof(long long) == 8
```

Table 4.2 lists the standard C built-in data types and indicates which are supported by the compiler.

Table 4.2. Summary of Data Types

Data Type	Supported	Size (in bits)
char	Yes	8
short	Yes	16
int	Yes	32
long	Yes	32
long long	Yes	64
enum	Yes	32
pointers	Yes	32 or 40
float	No	N/A
double	No	N/A
struct	Yes	Variable
union	Yes	Variable
array	Yes	Variable

4.2.7 Alignment of Data Types

The compiler aligns all chars on 1 byte boundaries, shorts on 2 byte boundaries, int, enum, long, and pointer data types on a 4 byte boundary, and all long long data types on an 8 byte boundary. In general, pointer values can be assumed to be aligned based on the type of the data pointed to.

Bit fields are stored within longwords. The next bit field starts at the next bit position within the current longword if, and only if, the entire bit-field element fits within the current longword (4 bytes). If the bitfield is too wide to fit, then the remaining bits in the current longword are padded and the bit-field begins at the next longword.

The bit field padding can be avoided if the structure that contains the bit field is declared with a __declspec(packed_bits) qualifier. In this case, the overflow bits of the bitfield wrap to the next longword and the bitfield is split between two longwords.

Aggregates (array, union, structure) assume the strictest alignment of any of their members. Hence an aggregate is aligned on a 8 byte boundary if it contains a long long member; otherwise, it is aligned on a 4 byte boundary. Aggregates allocated explicitly in SRAM/Cluster Local Scratch/local memory, are additionally aligned at least on a 4 byte boundary. Similarly, aggregates allocated explicitly in MEM, are aligned at least on an 8 byte boundary.

The compiler inserts padding between elements of structures to ensure that each element is aligned based on its type. A final tail padding is added to each structure to make its size a multiple of its required alignment. This guarantees that when you have an array of structs no additional padding is needed between array elements to align all the element structs. However, this padding can be avoided if the structure has been declared with the __declspec(packed) qualifier.

For more information about the __declspec(packed) qualifier, see Section 4.2.8.

> **Note**
>
> The __declspec(packed) qualifier implies the __declspec(packed_bits) qualifier.

If a structure contains one or more bit fields, its size is rounded up to the multiple of 4 bytes. For example, the following structure is declared with the _declspec(packed_bits) qualifier and contains two bit fields, so its size is 4 bytes:

```
typedef __declspec(packed_bits) struct {
    unsigned int a:4;
    unsigned int b:4;
    unsigned char c;
    unsigned char d;
} bfEg;
```

> **Note**
>
> The __declspec(...) qualifiers must be placed to the left of the "struct" keyword, as in the example above.

The alignment of a given structure can be changed with the __declspec(aligned(n)) directive, where "n" is a power of two, up to 2048 for memory and 64 for local memory. (For compatibility, __declspec(align(n)) may also be used.) If the structure's natural alignment is less than the word size of the structure's storage region (16 for MEM, 4 for other types of storage), the performance of whole structure copies can be improved by increasing the alignment value (padding) to the word size.

In the following example, the natural alignment is 1, but it can be changed to 4 by using the align directive:

```
typedef __declspec(cls, aligned(4)) struct // overrides natural alignment
{
    char c;         // natural alignment is "1" because of this element
    char s;
```

```
} str;
...
str x,y;
...
x = y;              // copy performance improved by manually setting alignment
```

Structure alignment is not optimized automatically because of the possibility that the structure may be embedded inside an array or another structure, which calls for the use of the structure's natural alignment.

If a structure is allocated in network processor local memory, setting the alignment to the closest power of two greater than the size of the structure (see Section 4.3.8.5) will allow the compiler to generate faster, offset-based addressing for the structure members. The disadvantage of doing this is an increase in the amount of wasted space needed to pad such objects.

> **Note**
>
> CLS and MEM support access on longword or quadword granularity respectively. Extraction or modification of bytes within a longword or quadword involves generation of additional instructions, and consequently results in some performance degradation.

4.2.8 Packed Aggregates

Aggregates (structures, unions, and arrays) are normally aligned on byte boundaries. Padding (up to 64 bytes) of aggregates is an automatic compiler function to improve performance. (This is discussed in Section 4.2.7.) Under some conditions you may wish to block padding. A __declspec(packed) qualifier can be used to avoid the padding between members of the aggregate and to avoid tail padding.

> **Note**
>
> __declspec(packed) implies __declspec(packed_bits).

The naturally aligned data for a network processor for Cluster Local Scratch and SRAM is 4 bytes; for MEM on the Netronome NFP-6xxx network processor it is 8 bytes. Cluster Local Scratch rings require alignment to 512 bytes.

Local memory natural alignment is 4 bytes, but when referencing big structures, the performance will improve with bigger alignment (up to 64 bytes). The reason is the hardware uses an "OR" to calculate the local memory address. For example, when you use *(p+16), assuming that p is your base pointer; if p is aligned to 16, then p|16 (p OR 16) is the same as p+16. If p's alignment is smaller than 16, we have to perform an ADD and reset the local mem base pointer, which is very time consuming.

4.2.9 Pointer Alignment Assumptions and Unaligned Pointers

Normally declared pointers will be assumed to point to data with correct alignment based on the natural alignment of the type they point to. The compiler generates code that correctly dereferences these pointers only in the case that they are correctly aligned. If they are not correctly aligned, the behavior is undefined.

Example:

```
    char *pc;       // pc can have any byte address
    int *pi;        // pi must be zero mod 4.
    short *ps;      // ps must be zero mod 2.
```

Additional alignment assumptions are made on values of variables declared as pointers to aggregates in memory. By default, variables declared as pointers to aggregates allocated to SRAM, Cluster Local Scratch or local memory are assumed to point to objects with a 4-byte minimum alignment. Variables declared as pointers to aggregates allocated to MEM, are assumed to point to objects with an 8-byte minimum alignment.

By using __declspec(unaligned) all alignment assumptions on the value of the pointer is avoided and the compiler will generate correct code to dereference the pointer. This code is considerably larger / slower than the code for aligned pointers.

Any pointer to a component of a packed aggregate is an unaligned pointer. An unaligned pointer cannot be assigned to an aligned pointer of the same type. Hence pointers to packed variables can be assigned only to unaligned pointer variables but not to regular pointer variables. An aligned pointer can be assigned to an unaligned pointer of the same type, since it is less restrictive.

Example:

```
int *pi;
__declspec(unaligned) int *pui;
__declspec(packed) struct {char x; int y} z;

pi = &z.y;             // Error since &z.y is an unaligned pointer
pui = &z.y;            // Ok since pui is an unaligned pointer
```

In this example, "pui" can point to any byte address, whereas "pi" can only point to 4-byte-aligned addresses.

The __declspec(aligned(n)) attribute can be used to declare the alignment of the objects that a given pointer will reference. "n" must be a byte value and a power of two. If you know that a pointer that is normally unaligned (a pointer to char, for instance) will only reference objects that are aligned on word boundaries, declaring the pointer with a higher alignment (__declspec(aligned(4)) for CLS, __declspec(aligned(8)) for MEM) will allow the compiler to generate faster code when dereferencing such pointers.

Example:

```
__declspec(packed) struct // structure is 4 bytes long but 1-byte aligned
{
    char a, b;
```

```
    short c;

} *ptr;
...
ptr x, y;
...
*x = *y; // unaligned copy
```

Since the natural alignment of this struct is 1 (single-byte aligned), the compiler will make no assumptions about the position of the structure in memory, and will generate code for the copy operation that will take into account the fact that the structure might span across two memory words, taking up only part of each word. If __declspec(aligned(4)) is added to the structure definition:

```
__declspec(packed, aligned(4)) struct
{
    char a, b;
    short c;
} *ptr;
...
ptr x, y;
...
*x = *y; // optimized copy
```

The compiler now assumes that the structure fits entirely into a single word in memory, and generates code that reads and writes only a single word.

4.3 Data Allocation

You can declare data with or without allocation attributes. Allocation attributes can describe where the variable is allocated (allocation region) or the scope of the variable. These allocation attributes are provided as __declspecs. The allocation region attributes indicate a choice of register or specific memory types where the data needs to be allocated.

For pointers, allocation attributes can be specified both for the pointer itself as well as for the object it points to. This implies that pointers are only compatible if they point to the same allocation region. Pointers declared to types without any allocation region attribute point to MEM by default.

In the absence of a region allocation attribute, variables are allocated to registers in the following circumstances:

- They are 64 bytes (128 bytes in 4-context mode) or less in size, and

- Their address is not taken, or if taken, the address reference is optimized away, (note: array references with non-constant indexes implicitly take the address of the array beginning), and

- There are enough registers to accommodate the variables

In the absence of a region allocation attribute, variables are allocated to local memory or IMEM in the following circumstances:

- If there are not enough registers to accommodate all user variables, some are spilled to local memory or IMEM. This includes global variables as well as those local to a function, function arguments, and return values.

- Variables larger than 64 bytes (128 bytes in 4-context mode) are generally allocated to local memory or IMEM.

- An array is allocated in local memory or IMEM, if it uses an index that is computed at runtime. There is no way to index variables in a GPR. The compiler can not use T_INDEX to index into xfer registers for various reasons (T_INDEX is not per-context, availability, performance, etc.). Similarly addressed variables are allocated to IMEM or local memory if the address reference cannot be optimized away.

The command-line option, -Qperfinfo=2, provides user-defined variables to register/memory mapping. (Refer to Table 3.1 for more information.)

NOTE: A Micro-C variable must be less than 0x20000000 bytes in size.

4.3.1 Register Model

Each microengine supports 256 General Purpose Registers (GPR) split into two banks (A and B), and 512 Transfer Registers (XFR) that are used to communicate with memory and I/O devices. The transfer registers are designated as follows:

In NFP Mode:

- 256 Transfer_In registers for I/O

- 256 Transfer_Out registers for I/O.

In addition to these registers, there are also 128 Next Neighbor registers for communication between neighboring microengines. Refer to Figure 4.1 for more information.

The microengines (MEs) support two modes for accessing registers: relative and absolute.

In relative mode, the registers are divided equally between the eight contexts so that each context effectively has its own set of registers. Each context may refer to relative general purpose registers in banks A and B and relative transfer registers without conflicting with the registers of another context.

In absolute mode a context may refer to any of the 256 GPRs. In this mode, some of the registers may be shared among contexts, and others may be context specific.

By setting up a microengine to run in 4-context mode, each context can access twice as many context relative registers. In this mode, odd contexts 1,3, 5, and 7 are disabled and the even contexts 0, 2, 4, and 6 have full access to their registers.

GPRs are located in two separate registers banks (A/B). Only one register from each bank may be read or written in any one clock cycle. Therefore, a typical binary instruction (w=r0+r1) may only reference alternating banks for their read operands. The compiler enforces this restriction by potentially adding register moves.

Absolute (ME shared) registers may require extra assembler move instructions as the instruction set is asymmetric (i.e. many instructions do not take absolute registers as operands).

4.3.2 Register Regions

The following __declspec() modifiers can be used to specify allocation to registers:

- __declspec(gp_reg) to allocate to a general purpose register

- __declspec(read_reg) to allocate to a read transfer register

- __declspec(write_reg) to allocate to a write transfer register

- __declspec(read_write_reg) to allocate to a read transfer register and to a write transfer register with the same register number.

- __declspec(nn_local_reg) to allocate to a next neighbor register local to this ME

- __declspec(nn_remote_reg) to declare the name of a next neighbor register in the next neighbor Microengine (this is to be used in -Qnn_mode=0, i.e., NEIGHBOR mode). The linker patches the physical register number for each reference.

NOTE: You may prepend `xfer_` when specifying a transfer register, so __declspec(xfer_read_reg), __declspec(xfer_write_reg) and __declspec(xfer_read_write_reg) are also valid.

When you declare a variable with the __declspec(read_write_reg) modifier, the compiler will allocate two registers with the same register number to it. Therefore, the variable is not initialized for reading when it is defined.

Consider the following example:

```
main()
{
    __declspec(read_write_reg) int a;
    int b;
    a =100;
    b = a;
    ...
}
```

In this example, the value of variable b is not 100, because variable a is not initialized for reading. In this case, you should either specify b = 100 directly, or use the mem_read() intrinsic function to initialize variable a before trying to read its value.

The Netronome NFP-6xxx network processor cannot allocate a variable to a register under certain situations in which case an error message is produced and compilation aborts. Example reasons for error are:

- If your program takes the address of such a variable

- If the variable is too large (greater than 64 bytes, or 128 bytes in 4-context mode)

- If there are too many variables requiring allocation to registers.

In such cases the compiler in addition to reporting this as a user error, reports an indication why it was not able to allocate the variable to a register.

In general, pointers to and arrays of register objects are not guaranteed to compile successfully, because the compiler needs to be aware of exactly which registers are being accessed at any given time. If the compiler cannot resolve a register pointer access into a "fixed" register access, an error will be generated.

4.3.2.1 General Purpose Registers

The qualifier gp_reg, if used in a variable declaration, causes the compiler to allocate general purpose register(s) for that variable.

4.3.2.2 Transfer Registers

The qualifiers for read/write transfer registers, if used in a variable declaration, indicate that you want to associate the variable with specific class of transfer register.

Transfer Registers (abbreviated as "Xfer Registers") are used for transferring data to and from the ME and locations external to the ME (for example MEM, CLS, etc). There are two types of transfer registers.

1. Transfer_In

2. Transfer_Out

Transfer_In Registers, when used as a source in an instruction, supply operands to the execution datapath. The specific register selected is either encoded in the instruction, or selected indirectly via T_Index. Transfer_In Registers are written by external units based on the Push_ID input to the ME.

As shown in Figure 4.1, the mux between the A and B push buses allow data arriving on the buses to be written to any of the Transfer Registers.

Transfer_Out Registers, when used as a destination in an instruction, are written with the result from the execution datapath. The specific register selected is encoded in the instruction, or selected indirectly via T_Index. Transfer_Out Registers supply data to external units based on the Pull_ID input to the ME.

As shown in Figure 4.1, the mux between the Transfer Registers allows data from either set to be put onto either the A or B pull buses.

Typically, the external units access the Transfer Registers in response to commands sent by the MEs; the commands are sent in response to instructions executed by the ME (for example, the command instructs a DRAM controller to read from external DRAM, and place the data into a Transfer_In register). However, it is possible for an external unit to access a given ME's Transfer Registers either autonomously, or under control of a different ME, or IA Core, etc. The ME interface signals controlling writing/ reading of the Transfer_In/Transfer_Out registers are independent of the operation of the rest of the ME.

> **Note**
>
> The names "Pull" and "Push" are from the perspective of the unit external to the ME. Pull data is data being transferred from the ME to an external unit and would traditionally be thought of by a programmer as a "write" despite the pull operation reading from the Xfer Registers. Similarly, a push operation is (usually) return of data from what is traditionally thought of as a "read" despite the push operation writing the data into the Xfer Registers.

> **Note**
>
> Writing and reading of Transfer Registers between the ME and external units is sometimes synchronized by software. This is normally done by having the external unit set an Event Signal to indicate when it has written/read the Transfer Register, and it may therefore be used by instructions as a source/destination. Another method is to write the data to the Transfer Register, and periodically read it to see when the data has arrived. When this method is used, the reader may get either the previous value or the new value. However, the hardware guarantees that the reader will get either of these values even if the read and write coincide on the same cycle (that is, the hardware will guarantee that the reader does not get an intermediate value where some of the bits have been updated and others have not yet been updated).

4.3.2.3 Next Neighbor Registers

Each microengine has 128 Next Neighbor (NN) registers that can be written in one of two ways as selected by the NN_MODE bit in the CTX_ENABLE CSR. When the NN_MODE bit is 0, a write to a NN register goes out of the ME to the corresponding NN register of the next neighbor ME. In this mode, the NN registers of an ME are read-only. When NN_MODE bit is 1, and a NN register is specified as a destination in an instruction, the selected NN register in the same ME is written. When an ME writes to its own neighbor register, it must wait 5 cycles (or instructions) before it executes the instruction that reads the same register in order to get the newly written value.

> **Caution**
>
> Changing the NN_MODE bit at runtime is not allowed, and the behavior is undefined.

The NN register can be specified directly as a context-relative register or indirectly through an index register. In the direct access mode the 128 registers are partitioned between the 8 contexts (or 4 contexts in 4-context mode) each addressing its own set (0-15 or 0-31) of context relative registers. In the indirect mode, one of the 128 NN registers is selected through either a local CSR NN_Put (*n$index++ used as destination) or NN_Get (*n$index or *n$index++ used as source). The compiler supports the indirect access mode through intrinsic functions.
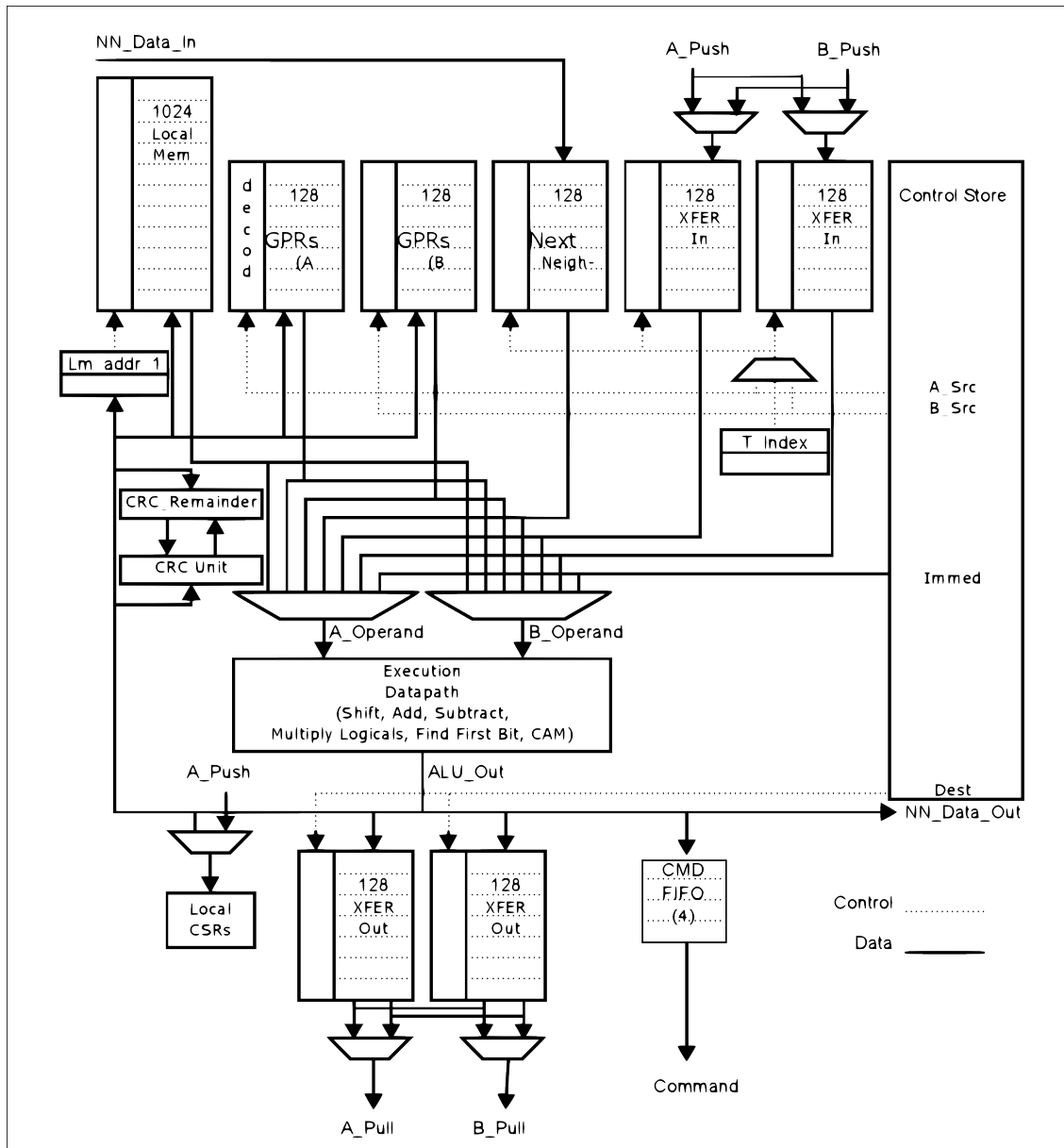
Figure 4.1. Microengine Block Diagram for Netronome NFP-6000 Network Processor

The nn_local_reg qualifier causes allocation of the variable to a next neighbor register in this ME. If the NN_MODE is 0 (NEIGHBOR), this variable is read-only within this ME program. In this mode, the previous neighbor ME program may declare the same variable with an nn_remote_reg qualifier and can only write into it.

When -Qnn_mode is 1 (SELF) a variable with the nn_local_reg qualifier cannot be modified from another ME. (NOTE: there is a 6-cycle latency between a write instruction and a subsequent read instruction from the same NN register with a new value). It can be both read and written within this ME program. In this mode one cannot use the nn_remote_reg qualifier.

In the SELF mode all next neighbor registers should be declared with __declspec(nn_local_reg). Given this, the following is to be followed. In SELF_mode, a nn_local_reg variable can be read or written, and a nn_remote_reg

variable should not be used. In NEIGHBOR mode, a nn_remote_reg variable can only be written, and a nn_local_reg can only be read.

> **Note**
>
> If -Qnn_mode=0 (neighbor mode) variables declared as nn_local or nn_remote must be declared in global scope (i.e., they cannot be declared as local variables within a function) because they are referred to from a neighbor microengine.
>
> When -Qnn_mode=0 (neighbor mode) is specified, the compiler will enforce subword access restrictions appropriate to transfer registers. A nn_local_reg variable may therefore be unsuitable when bitfield-style access is desired.

4.3.2.4 Volatile Registers

The volatile attribute can be applied to variables in transfer registers or to signal variables. This attribute indicates that the register can be read or written by instructions not explicitly referencing the register. An example of this is transfer registers that are read or written by reflect operations from another microengine.

When a register or signal variable is declared volatile, the register or signal that the compiler assigns to that variable will not be used for any other purpose within the scope of that variable. If the variable is global, then its register or signal will be assigned to that variable for the entire program. If it is local to a function, then the register or signal will be assigned to that variable for the scope of that function. This insures that external MEs or hardware will be able to access the variable even though that variable may not be "live," or actively in use, within the currently executing segment of code. If the volatile variable is local to a function, the user will need to provide a synchronization mechanism to insure that the function does not return until the data is no longer needed by the external MEs or hardware. Function-local volatiles may allow the compiler to allocate registers more efficiently than global volatiles, since the register can be reused for other variables once the function returns. In general, volatile variables limit the efficiency of register allocation, and should only be used when necessary. If you wish to indicate that a register or signal variable can be accessed by an external entity only within a certain region of the program, the __implicit_read(), __implicit_write() and __implicit_undef() intrinsics can be used instead of the volatile attribute.

4.3.3 Memory Regions

The Netronome NFP-6xxx network processor has these memory regions:

- Local Memory
- SRAM (this has been deprecated but is still supported for backwards compatibility)
- MU (Memory unit) includes IMEM, EMEM and CTM
- Cluster Local Scratch (CLS)

The compiler allows the following __declspec modifiers to specify memory to allocate in:

- __declspec(local_mem) to allocate to LOCAL MEMORY.

- __declspec(sram) to allocate to SRAM

- __declspec(emem) to allocate to closest available EMEM. Additionally a specific EMEM can by targeted with __declspec(ememN). N can be value 0-2 (Actual island is N+24). 40-bit address by default unless the -compat32 option is specified.

- __declspec(imem) to allocate to closest available IMEM. Additionally a specific IMEM can by targeted with __declspec(imemN). N can be value 0-1 (Actual island is N+28). 40-bit address by default unless the -compat32 option is specified.

- __declspec(ctm) to allocate to closest available or local CTM. Additionally a specific CTM can by targeted with __declspec(ctmN). N is the target island ID. Non-local targets(ctmN) are 40-bit addresses by default, unless the -compat32 option is specified.

- __declspec(cls) to allocate to closest available or local cluster local scratch. Additionally a specific CLS can by targeted with __declspec(clsN). N is the target island ID. Non-local targets(clsN) are 40-bit adresses by default, unless the -compat32 option is specified.

- __declspec(mem) is used to declare a generic memory pointer. This can either be used a user-defined pointer (i.e. must include correct island bits) or it can obtain the island informations from the address of symbol/variable. 40-bit address by default unless the -compat32 option is specified.

- __declspec(addr40) is used to declare a symbol with a 40-bit address. This is for use with imem(N)/emem(N)/ctm(N)/cls(N)/mem. This will ensure a pointer/variable is declared with 40-bit address regardless of the configured default.

- __declspec(addr32) is used to declare a symbol with a 32-bit address. This is for use with imem(N)/emem(N)/ctm(N)/cls(N)/mem. This will ensure a pointer/variable is declared with 32-bit address regardless of the configured default.

Examples:

```
__declspec(local_mem) struct msg_header header;
```

declares a variable of type struct msg_header which resides in Local Memory.

```
__declspec(mem) buffer * buf_ptr;
```

declares a pointer to a buffer data type in MEM. buf_ptr is assigned to a register.

```
__declspec(mem) buffer * __declspec(cls) buf_ptr_1;
```

declares a pointer to a buffer data type in MEM. The pointer resides in CLS.

> **Note**
>
> The memory type modifier applies to the type to its right, thus the first one indicates that the buffer is in MEM, while the second indicates that the pointer to it is in CLS. Variables declared in memory using the above syntax can be used with standard C syntax. When the C code reads or writes a variable, the compiler automatically guarantees synchronization. Typically, if the compiler reads a variable, it issues a context swap and waits until the data is available. This swap may be delayed until after other computations that do not depend on the read value to complete. Writes also generate context swaps to prevent read-after-write or write-after-write

interference. The compiler may issue multiple writes and reads before a context swap in cases where it can disambiguate the references and guarantee that no conflicts occur.

> **Note**
>
> CTM and IMEM cannot have different MU engines working on the same 64 byte chunk of memory (also 64 byte-aligned). This will typically lead to corruption int the cache-line.

4.3.3.1 Local Memory

Each microengine has a local memory area that is private to it. This memory holds 1024 longwords and can be addressed through one of two local memory address CSRs. These CSRs can be configured as either being local to each context or shared among contexts. The local memory can be accessed as operands in microcode instructions (with some restrictions) by setting up one of the two local memory pointers. Hence, it serves as a register set that can be addressed indirectly. There is a 3-cycle latency that must be observed between the setup of the local memory address CSR and its use in dereferencing local memory.

For details about local memory layout and allocation, refer to Section 4.3.8.

4.3.3.2 External Memory

External memory accesses are asynchronous. When memory is read, the thread must do one of the following:

- Swap itself out, allowing other threads to run.

- Wait until the operation signals completion before using the data read.

Similarly, when memory is written, care must be taken not to read it or write a new value before the write has completed.

Memory is also divided into three separate regions, each with its own address space. These are:

Memory Region	Speed	Size	Description
MEM (IMEM/EMEM/CTM)	slowest to fastest	largest to smallest	Directly addressable on quadword boundaries (64-bits) only (See Note 1).
SRAM (legacy)			Directly addressable on longword boundaries (32-bits) only (See Note 1).
Cluster Local Scratch			Directly addressable on longword boundaries (32-bits) only (See Note 1).

> **Note**
>
> 1. The memory is only accessed on a 32-bit (SRAM, Cluster Local Scratch) or 64-bit (MEM) boundary. However, the address specified is a byte address, with the lower 2 bits (SRAM, Cluster Local Scratch) ignored by the memory subsystem.

2. You always read and write memory through transfer registers (XFRs). The transfer registers are divided into read transfer and write transfer registers. The read XFRs are written by external units then used as source, while the write XFRs are read by external units after they are used as a destination (see Figure 4.1).

4.3.3.3 Signals

The Netronome NFP-6xxx network processor architecture provides a set of 15 signals per context that can be associated with certain hardware events. These signals might be used to notify the execution context that a certain request has been completed. The choice of hardware signal to use is specified in the microcode. Hence these signals are like hardware registers that can be allocated and used by software. Certain events such as MEM access, or accesses that involve a pull and a push from transfer registers require specification of two signal registers (an even-odd pair) on which the event completion is signaled.

4.3.3.4 Reflector

The Netronome NFP-6xxx network processor architecture supports an operation called Reflector, which provides the ability for one microengine thread to read its transfer register from the local CSR of another microengine context or write its transfer register to the transfer register or local CSR of another microengine context. Access completion signals can optionally be requested for one or both of the sending and receiving threads.

4.3.3.5 Indirect Register Access

The MEv2 architecture supports indirect register access, using the T_INDEX, NN_PUT, and NN_GET registers. Because the value of these registers generally cannot be determined at compile time, the use of indirect register access in inline assembly is not recommended. The compiler will perform register allocation and live range analysis without taking the indirect register access into account, and, as a result, the values read or written using indirect access may be incorrect. For access to the next-neighbor registers in ring mode, you can use the intrinsic functions designed for this purpose.

4.3.3.6 Threading Model

The programming model for the Netronome NFP-6xxx network processor architecture involves programs running on multiple microengines, each running multiple threads. Each microengine can be configured to run either in 4-context mode or 8-context mode by setting a bit in the CTX_ENABLE CSR. In the 4-context mode, twice as many context relative registers are available to each of the 4 threads. The multithreading is explicit; that is, you must partition the tasks across threads. You also need to partition the program across microengines and manage all interthread and interprocess communication.

The Netronome NFP-6xxx network processor is designed to handle a very large number of packets of data in communications routing applications. The threading model contributes to this bandwidth by allowing useful work to be done by another thread while one thread is waiting for completion of a memory or I/O transfer. Thus, the usual case is, when you read or write memory, your thread is swapped out, allowing other tasks to run.

4.3.3.7 Features Not Supported

Several features generally found on all general-purpose processors are not supported on the Netronome NFP-6xxx network processor microengines. There is:

• No support for either a data stack or a subroutine call stack

• No data or instruction cache

• No traps or exception support

• No misalignment support

• No direct support for byte aligned access (direct access must be aligned on longword (SRAM, Cluster Local Scratch) and quadword (MEM)).

A stack could be implemented in IMEM with software, but due to the long memory latency and multiple instructions required for stack manipulation it would be prohibitively slow. Local memory is not well suited for implementing the stack either, because of its limited size (80 longwords per context). Hence this architecture is not amenable to function recursion and standard caller/callee register partitioning. Consequently, the compiler does not support recursion, and resorts to whole program register allocation to avoid or reduce the overhead of spilling/filling registers.

Function calls are implemented by loading a register with the return address and jumping to the function. The load of the return address is placed in the delay slot of the jump instruction to minimize the overhead.

4.3.4 Shared Data

Because the processor supports multiple contexts, you can declare data to be shared between contexts or to be local to each context on a microengine. By default, all variables (both within and outside of a function) are local to a context, thus they are physically duplicated for each of the eight contexts on the ME. In other words, each context has a separate copy of the variable to work with no matter where the variable is allocated (i.e., registers or memory). In addition to this, you sometimes need variables that are shared by all eight contexts on a processor.

The compiler supports this with another __declspec modifier:

```
__declspec(shared)
```

The shared attribute can be combined with a memory region attribute in a single __declspec, example: __declspec(shared imem) int x.

Without a memory region, the shared attribute declares a potential register candidate that is shared by all contexts and is subject to the normal register restrictions.

4.3.5 Global Data

You can declare data in MEM, or Cluster Local Scratch memory that is shared by all the microengines on a network processor. One microengine program will "export" (and optionally initialize) the variable with:

```
__declspec(export imem) int i = 42; // initialization is optional
```

The other microengine programs will "import" the variable with:

```
__declspec(import imem) int i; // no initialization possible
```

Although the variable is not "attached" to the microengine that exports it, the export/import qualifiers are needed to prevent multiple initializations of the variable.

The exact scope of a global variable may be specified with `__declspec(scope(global))` or `__declspec(scope(island))`.

4.3.6 Load Time Constants

Load-time constants (i.e. constants that are bound to fixed values at load time and used at run time) are supported through an intrinsic __LoadTimeConstant(string).

Example:

```
C = a + __LoadTimeConstant("LTC");
```

For each call to __LoadTimeConstant("LTC"), the compiler generates a trio of immed[t, 0] instructions, each with a linker directive on source bits, and uses the temporary variable "t" in the expression (see above example) for the constant "LTC". The linker uses a 64-bit constant for "LTC" to patch those bits when the program is loaded, but the compiler currently uses only the lower 48 bits. See the documentation for this function in the MicroC Reference Manual for a list of predefined import variables which the linker will fully resolve at link time.

4.3.7 Signals

The compiler exposes hardware signal and signal pair registers as special predefined data types, SIGNAL and SIGNAL_PAIR respectively. Alternatively, you can apply a __declspec(signal) or __declspec(signal_pair) to a variable of type int or a struct comprised of two ints respectively. You can declare signal variables and pass them by reference to various intrinsic function calls.

The Netronome NFP-6xxx network processor microengine provides 15 signals for each execution context. You may have more than 15 signal variables so long as no more than 15 are in use simultaneously in the program. This restriction is imposed because there is no efficient mechanism to temporarily store signals in memory or other registers.

The following example illustrates the use of signal variables.

```
   SIGNAL sig;
   SIGNAL_PAIR sp;
...
   mem_read(dst1, src1, 1, sig_done, &sp);
   cls_read( dst, src, 4, sig_done, &sig );
   while ( ! signal_test(&sig) )
```

```
    {
/* do something*/
    }
    ..= dst;
    __wait_for_all(&sp);
    ..= dst1;
```

Signals can be shared across functions or across microengines in a limited way using the support provided to determine the signal number allocated to a signal variable or by creating a signal mask. This support is provided through intrinsic functions (__signal_number(), and __signals() respectively). To associate a signal variable with a specific number, you need to call the __assign_relative_register intrinsic. When you use these functions you need to convey additional information to the compiler by using the intrinsics __implicit_read(), __implicit_write() and __implicit_undef() to indicate the lifetimes of the signals involved.

4.3.7.1 Signal Variable Restrictions

Signal variables have special properties that make them unlike normal variables. The read access of a signal variable if the signal has been delivered has the potential side effect of clearing this signal. Consequently, there are certain restrictions imposed on the usage of signal variables. These restrictions are listed below.

- These variables cannot be assigned to or used in any way other than as addressed arguments to specialized intrinsic functions or through inline assembly.

- You cannot take the address of a signal variable, except when passing it as an argument to an intrinsic function. However, you can get a signal's address as an int through the __signal_number() intrinsic or as a mask through the __signals() intrinsic. This can then be passed to a function as an int type parameter. You may also have to insert calls to __implicit_read()/__implicit_write() to convey the live ranges of indirectly accessed signal variables, normally in the caller around the call site if there are no other reference to the signal variable. This is because __implicit_read() and implicit_write() accept signal variable, and callee function taking such int type parameter usually does not know what signal variables that int contains.

- You cannot create an array of signals or __declspec any aggregate variable to be a signal. A signal variable if declared explicitly using __declspec(signal), must be declared as a 4 byte quantity such as an int or long. Similarly __declspec(signal_pair) can only be applied to an 8 byte data type.

- You cannot read/write, copy, or pass signals as arguments or return values across user function boundaries. Furthermore, as hardware signal registers are context specific, one cannot declare a signal that is shared across contexts. Signal variables can, however, be used for cross-thread communication by declaring them as __declspec(remote) or __declspec(visible), or by explicit user allocation using the __assign_relative_register() intrinsic.

- A Netronome NFP-6xxx network processor microengine, when executing the ctx_arb instruction with the OR token (__wait_for_any() intrinsic), does not clear any of the signals that are asserted. After the use of signal variables in a __wait_for_any() intrinsic, you must clear them with calls to __wait_for_all() or signal_test() intrinsics, prior to reusing the signals.

4.3.8 Local Memory Allocation

4.3.8.1 Overview

You can allocate variables to local memory by applying the __declspec(local_mem) type qualifier. These variables may be used in all situations where variables declared in other memory regions may be used, with a few restrictions. First, data in local memory cannot be exported to other microengines. Also, local memory is limited in size: each ME only contains 1024 32-bit words of local memory, which must be shared among the running contexts (specified with the -Qnctx= or Qnthreads= command-line options) on that microengine.

In addition to allocating user-qualified variables to local memory, the compiler may also spill (copy) some variables that normally reside in registers to local memory when not enough registers are available for computation. The -Qperfinfo=1 command-line option will indicate which variables, if any, are being spilled.

4.3.8.2 Placement of Variables

The compiler will make decisions on the allocation and layout of variables declared to be in local memory. The programmer is also allowed to perform manual allocation and access local memory through addresses hard-coded in the program provided this space has been reserved using the "-Qlm_start=<n>" command-line option. No assumptions can be made about the layout or relative ordering of individual variables allocated in local memory by the compiler. For example, with the following declaration:

```
__declspec(local_mem) int x, y;
```

You cannot assume that y is allocated at the next word following x. Any such assumptions should be confined to be between data members of an aggregate (struct/union/array), where the layout of the aggregate is defined by the C language definition.

If one or more instructions reference two variables allocated in local memory, then both those variables can be addressed using the same local memory base register value, provided that the local memory pointer is aligned on a 16-word (i.e. 64 byte) boundary, and the two variables are within the same 16-word aligned block. Therefore, to minimize the cost of instructions involved in setting up a local memory base register, decisions on the placement of variables in local memory relative to one another are optimized based on the usage pattern of the variables. Variables are grouped into "buckets," where all the variables in a bucket are indexed with the same base pointer, to minimize the number of times when a local memory base register has to be reassigned through the local_csr_write[] instruction. However, the Netronome network processor architecture's OR-based offsetting forces buckets to be aligned on an appropriate word boundary, which can create unused "gaps" between the buckets.

4.3.8.3 Thread Local vs. Shared Storage
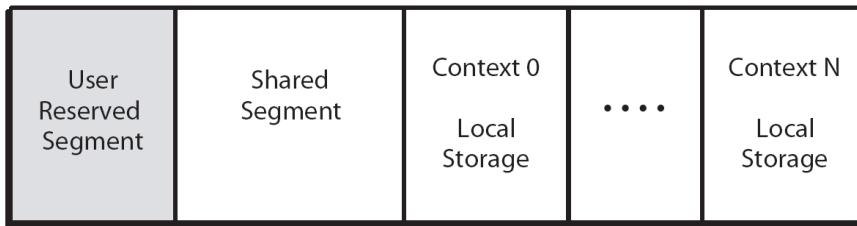
Figure 4.2. Local Memory Layout

The preceding figure illustrates local memory layout. Each "segment" contains "buckets," which are addressed, in the case of thread-local storage, using one base pointer value for each context, and in the case of shared storage, one base pointer value for all contexts. The first segment is optional - you might request that a portion of local memory be excluded from compiler allocation by using the "-Qlm_start=<n>" command-line option. This tells the compiler not to allocate variables to local memory addresses in the range [0, n]. If you do not specify this option, the compiler will start allocating local memory at address 0. The next segment, which the compiler allocates, is the shared segment, which holds all the local memory objects shared by multiple contexts, i.e. the variables specified with the "shared" qualifier. The remaining segments contain the thread-local variables for each context running on the microengine.

4.3.8.4 Viewing Local Memory Usage

The allocation of local memory can be examined with the following compiler option:

-Qperfinfo=16

For example, for this program, which allocates an array of 10 integers in thread-local storage and a single integer in shared memory:

```
__declspec(local_mem,shared) int x;
void main() {
__declspec(local_mem) int a[10];
...
a[9] = x;
....
}
```

The allocation information from "-Qperfinfo=16"will be output as follows:

→ User reserved: 0 bytes, Shared segment: 64 bytes, Local page (including gap): 64 bytes

→ Gap between context pages is 24 bytes. The data on the page is 40 bytes.

Direct access local mem group 0x180d900

Maximum offset used: 36 Alignment: 4

Num members: 1 Total size: 40

[This group contains thread local symbols]

lmem.c(8): a allocated at offset 0

Direct access local mem group 0x180d9cc

Maximum offset used: 0 Alignment: 4

Num members: 1 Total size: 4

[This group contains shared symbols]

lmem.c(15): _x allocated at offset 0

This information corresponds to the following memory layout:
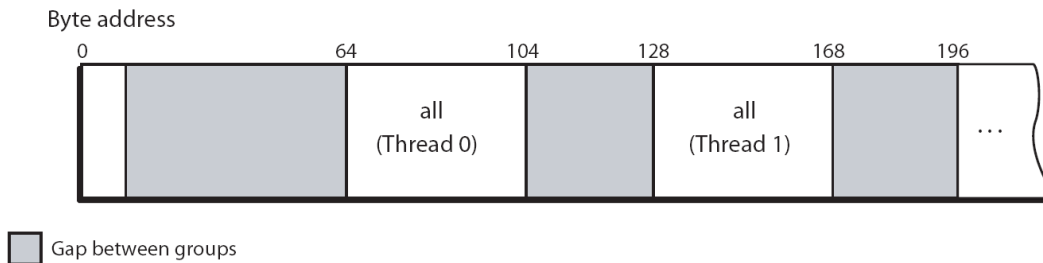
Figure 4.3. Local Memory Layout for Program 1

The first part of the allocation information describes the size and placement of each segment in local memory. Assuming that you have not reserved any local memory for manual allocation, the shared segment starts at address 0, and is 64 bytes long. Since there are only four bytes of shared storage (corresponding to the variable x) that are actually allocated, the other 60 bytes of the shared segment remain unused. Each thread-local storage segment (called a "local page") is also 64 bytes long. 40 of these bytes are data (the 10-word array) and 24 bytes are the "gap," or unused space.

The next sections of the allocation information describe the object groups (buckets) contained in each segment. As described above, a group contains either shared data, or thread-local data, but not both. If a group contains shared data, all the objects in the group are accessed by all threads using one base pointer, OR'ed with the same offset. If a group contains thread-local data, all the objects in that group are accessed using a different base pointer on each thread, OR'ed with the same offset. So if "group 0" was assigned to thread-local data and contained two objects, a and b, each thread would contain a base pointer that would point to the beginning of that thread's "group 0," each a would be accessed using the same offset for each thread, and each b would be accessed using the same offset for each thread (but a different offset than a).

The first group in this example contains one thread-local object, the array, a[], declared at source line 8 in the `lmem.c` file. The maximum constant offset used in the group is 36 bytes (corresponding to a[9] in the above code). This affects the spacing between groups, as each group must be aligned so that all its indexed elements are addressable

with OR-based indexing. The alignment value specified is the C-language-specified alignment of the group's objects; it is 8 bytes for groups containing 64-bit primitive types and 4 bytes for groups containing 32-bit or smaller primitive types. The group is 40 bytes long. The given offset ("0" in this case) of the object a is the object's byte offset from the beginning of the entire local page for a given thread, therefore the offset of the first object in a group identifies the alignment of the group.

The next group described is the group containing the shared variable, x. The information for this group reads similarly to the other group—the maximum offset used is 0 (since there is only one word of storage used), the alignment is 4 bytes, there is one object in the group, and the group is 4 bytes long. The object "x" is allocated at offset 0 from the beginning of the shared data segment.

4.3.8.5 Alignment Information for Local Memory Pointers

In certain cases multiple accesses to local memory objects might be driven by a single local_csr_wr instruction. The compiler will initialize base register once and then use different offsets to access different data items allocated in close proximity of that base. To do that, however, the compiler must know the alignment properties of the base address. Without this knowledge, the compiler will not be able to drive multiple pointer accesses through the same base value. Legal values for local memory alignment are 8, 16, 32 and 64. All these values will be interpreted by the compiler as byte addresses. For example:

```
typedef struct
{
    int a;
    int b;
    int c;
    int d;
} MyStruct;

void foo(MyStruct __declspec(local_mem aligned(16)) * P)
    ... = P->a
    ... = P->b
    ... = P->c
    ... = P->d
}

main()
{
    MyStruct __declspec(local_mem aligned(16)) X;
                foo (&X );
}
```

In the example given above the compiler might be able to generate just one local_csr_wr instruction initializing the base with the value of pointer and then use that base for four different accesses to the fields of the structure. Without having alignment information on pointer P, the compiler will have no choice but to drive each single access to a field of the structure through separate base. Therefore, it will generate four local_csr_wr instructions and this might result in poor performance. Whenever the same local memory pointer might be used as a base for multiple different accesses, you should declare the pointer with some alignment information.

4.3.8.6 Suggestions for Improving Local Memory Use

To assist the compiler with optimizing local memory usage, you can apply several techniques:

1. Place large objects in the shared segment. Reducing the alignment restrictions on thread-local data will provide space savings for each thread, because all threads have the same layout in their thread-local storage segments.

2. Use __declspec(align(n)) judiciously as described in Section 4.3.8.5. Larger alignments may cause fragmentation in local memory but it may help the compiler to group several accesses with constant offset (e.g. p->x, or p[3]) using a single local_csr_wr[] to base pointer. Variable addressing (e.g. p->[i], where i is a variable) will cause the compiler to do an index calculation and generate a local_csr_write[] instruction plus three nops, which trades off runtime performance for space savings.

4.3.9 Atomic Variables

4.3.9.1 Overview

Atomic variables can be allocated in MEM or CLS by using the __declspec(atomic) type qualifier. If the memory region specifier is omitted, then CLS is assumed by default. Atomic variables have the benefit of optimizing the following C operators: ++, --, +=, -=. Keeping the constant value being added or subtracted within 7 bits, will allow the compiler to further optimize by using the immediate form of the atomic commands (which do not need transfer registers or signals). This however carries the risk that when instructions are issued at a very high rate, the microengine may stall, causing the execution of all threads to suspend until the backlog has been processed.

4.3.9.2 Usage

The following code snippet show basic use and declaration of atomic variables.

```
__declspec(atomic) int x; //Cluster local scratch
__declspec(atomic, mem) short y;
__declspec(atomic, addr40, mem) unsigned int *z;
__declspec(atomic, cls) unsigned int array[8];
x++;
y += 20;
*z -= 40;
array[2]--;
```

4.3.9.3 Supported Data Types

The atomic modifier supports int and long long as well as arrays and pointers to the aforementioned types. Chars and shorts are also supported but arrays of them are not. Furthermore, it also offers support for enums.

All atomic variable addresses must also be aligned to 4-byte boundaries for 32-bit variables and 8-byte boundaries for 64-bit variables. Any deviation from this will lead to unpredictable results.

4.3.10 40-bit MEM Pointer (Deprecated)

A 40-bit MEM pointer can be declared by applying the __declspec(mem ptr40) type qualifier. This is provided as a mechanism to provide further information about the type of memory being accessed in a command. The top two bits of a 40-bit memory unit encode the memory addressing type. Refer to programmer reference manual for more information on recommended addressing modes. Please use addr40 for future code.

> **Note**
>
> For more information on recommended addressing modes, see *Netronome Network Flow Processor 6xxx: Microengine Programmer's Reference Manual.*

4.3.10.1 Usage

The following code snippet show basic use and declaration of 40-bit MEM pointers.

```
__declspec(mem ptr40) unsigned int *ptr = (__declspec(mem ptr40) unsigned int *)0x2002030400ULL;

__declspec(packed) struct s
{
   char c;
   long long ll;
};
__declspec(mem) struct s dr;
__declspec(mem ptr40) char *dr_ptr = (__declspec(mem ptr40) char *)&dr;

*ptr = 1;
dr.c = 0xab;
```

4.3.10.2 Supported Data Types

The 40-bit MEM pointer supports in the same way as the regular 32-bit pointer. For example, all C standard scalar data types pointed to, including signed and unsigned char, short, int, long, long long as well as structs and unions are supported. The pointer arithmetic and alignment of all the types are also supported.

4.4 Reflector Inputs/Outputs

A reflector operation involves read/write of transfer registers across MEs and hence across programs. Such variables are transfer register variables that are visible across MEs. Hence they are declared with the following declspecs:

```
__declspec(visible read_reg/write_reg);
__declspec(remote read_reg/write_reg);
```

A variable that has a __declspec(remote read_reg) refers to a context relative read transfer register in another ME. Note that unlike normal exported variables, which are by definition shared across the contexts of an ME, remote/visible transfer register variables are context relative. Also, remote or visible variables must be declared outside function to avoid name mangling which may confuse the nfld linker when linking.

Signals used for signaling the remote or both MEs are also declared similarly.

```
__declspec(visible) SIGNAL/SIGNAL_PAIR;
__declspec(remote) SIGNAL/SIGNAL_PAIR;
```

When you use sig_both to signal both MEs, you must ensure that the same signal number is used for both signals involved. This is done by manually allocating the same signal number to both signal variables involved, using the __assign_relative_register() intrinsic:

ME0 Program	ME1 Program
__declspec(write_reg) me0_x;	__declspec(visible read_reg) int me1_x;
__declspec(remote read_reg) int me1_x;	__declspec(visible) SIGNAL me1_sig;
__declspec(remote) SIGNAL me1_sig;	
__assign_relative_register(&me1_sig, SIG_USE);	__assign_relative_register(&me1_sig, SIG_USE);
	__implicit_write(&me1_sig);
	__implicit_write(&me1_x);
reflect_write(&me0_x, ME1, me1_x, CTX, 1, sig_remote, sig_done, &me1_sig);	
__free_write_buffer(&me0_x);	__wait_for_all(&me1_sig);
	... =me1_x; //use me1_x

> **Note**
>
> A remote read transfer register can only be written in the micro-engine declaring it as remote, and a remote write transfer register can only be read in the micro-engine declaring it as remote.

Calls to __implicit_write() need to be inserted in the ME1 program to indicate the earliest point in the ME1 program execution, where ME0 might write the registers associated with visible variables me1_x and me1_sig.

4.5 Summary of Allowed Data Attribute Combinations

Table 4.3. Summary of Allowed Combinations of Attributes on Data

	Thread Local	Shared	Export	Remote/Visible
GP Register	Yes	Yes	No	No
Transfer Register	Yes	No	No	Yes
Signal Register	Yes	No	No	Yes
Next Neighbor	Yes	No	No	Yes
Local Memory	Yes	Yes	No	No
SRAM (legacy)	Yes	Yes	Yes	No
MEM	Yes	Yes	Yes	No
Cluster Local Scratch	Yes	Yes	Yes	No

4.6 Expressions

In general, all C expression syntax involving the supported data types is supported. Remote XFR register variables can only be used in reflect inline asm and reflect intrinsics. Signal variables can only be used as intrinsic arguments or in inline asm. Function pointers are not supported.

> ### Note
>
> A special note for the implementation of integer divide-by-zero. Because microengines do not support signals nor exceptions, evaluating an expression such as x/0 or x%0 for any integer x, signed or unsigned, returns 0xffffffff for 32-bit integers or 0xffffffffffffffff for 64-bit integers.

4.7 Statements

The compiler supports all C statements involving supported expressions.

4.8 Functions

4.8.1 Supported

The compiler supports C functions including:

- Local variables with memory regions (equivalent to static locals).

4.8.2 Not Supported

- Recursion

- Variable length argument lists

- Pointers to function

- Passing aggregates larger than 64 bytes (or 128 bytes in 4-context mode. In NFP Enhanced Mode, 128 bytes in 8-context mode, 256 bytes in 4-context mode.) as function arguments or return value.

The implementation on recursion and variable length argument lists on the Netronome NFP-6xxx network processor impacts performance and is therefore not supported. The restriction on function pointers allows the compiler to determine the call-graph exactly and optimize every function call. The use of function pointers requires that all functions that might be called through a pointer have a standard argument passing and return value mechanism. Since aggregates larger than 64 bytes (or 128 bytes in 4-context mode. In NFP Enhanced Mode, 128 bytes in 8-context mode, 256 bytes in 4-context mode.) are never allocated to registers, and function arguments and return values are passed only in registers, the compiler gives an error message on function arguments/ return values that are larger than 64 bytes (or 128 bytes in 4-context mode. In NFP Enhanced Mode, 128 bytes in 8-context mode, 256 bytes in 4-context mode.).

4.8.3 NFP Enhanced Mode Function Attributes

The following function attributes can be applied to functions to define their characteristics with respect to inlining.

- __noinline func();

- __forceinline func(), __inline func();

The attribute __noinline indicates that the function should not be inlined. The attribute __inline is a hint to the compiler to inline the program. The attribute __forceinline is a strong hint for the compiler to inline the function. Unless optimization or inlining is turned off the function will be inlined by the compiler. __forceinline functions are by default static functions. See Section 2.3.2 for information on inline __forceinline functions.

4.8.4 Optimizing Pointer Arguments

It is possible to improve the speed of access to function arguments passed in with pointers. For example:

```
void foo(MyStruct *p)
{
some code using *p and assigning *p
}

main()
{
```

```
    MyStruct x;
    ...code ...
    foo(&x);
    ...code...
}
```

In the preceding code, you wish to use the function foo to modify the contents of the structure x, by passing the address of x to foo. Since general-purpose registers cannot be accessed with pointers, the compiler cannot place the structure x into registers, which significantly slows access to the data contained in x.

One way to write the preceding code, which still allows "x" to be placed into registers, is as follows:

```
MyStruct CompilerTemp;

void foo( void )
{
some code directly using and assigning CompilerTemp
}

main()
{
MyStruct x
    ...code...
    CompilerTemp = x;              /* copy -in */
    foo();
    x = CompilerTemp;              /* copy-out */
}
```

In this code, the program copies x into a global temporary structure that is accessible to both main and foo, allows foo to perform operations on this temporary structure, and copies the results back into x. Both the temporary structure CompilerTemp and the structure x can be placed into registers, with a significant performance gain over the first example.

4.8.4.1 The "restrict" Qualifier

The compiler can automatically perform the structure copy optimization described above if the "restrict" qualifier is applied to the function definition of "foo":

```
void foo(MyStruct * restrict p)
{
alias-free code using *p and assigning *p
}
main()
{
    MyStruct x;
    ...code ...
    foo(&x);
    ...code...
}
```

The "restrict" qualifier must come directly before the variable name. This qualifier informs the compiler that the memory pointed to by the attached pointer parameter is not accessed through "unknown" means—either through another pointer whose definition is ambiguous, or from another thread. The optimization described above is only guaranteed to be possible, and safe, when the following conditions exist:

- The memory location pointed to by the "restrict" pointer parameter is only accessed using a dereference of that particular pointer, or with copies of that pointer which are defined within the function. The pointer is not assigned to a non-restricted pointer, and the restricted pointer copies, if they exist, are only assigned to once (similar to "const" variables).

- The memory location pointed to by the "restrict" parameter is only accessed from a single thread and a single microengine while the function is executing.

- The "restrict" pointer parameter is not cast to another type of pointer.

- The "restrict" parameter is dereferenced with a constant offset. For example, in the preceding code, the function body can contain *p, p->field, and *(p+2), but not *(p+i) where "i" is not a constant.

The compiler can check for simple violations of the above rules and will not perform the "structure copy optimization" in those cases, but you must determine whether the "restrict" qualifier is appropriate (otherwise the qualifier would not be needed). The option "-Qperfinfo=256" will tell the compiler to print out information on any "restrict rule violations" that it finds.

4.9 User Assisted Live Range Analysis

Register allocation and other compiler optimizations depend on correct live range information of register variables to make the right decision. A register variable is defined as a variable that could be assigned to a register (transfer register, signal register, general purpose register). A live range of a register variable is the period between the definition of this variable and the last use of the defined value. When a register variable has multiple definitions in the program, and each definition has sequential reads, multiple live ranges are assigned to the same variable. Each live range covers one definition and its sequential reads.

The live range of a register variable begins with a write into the variable; and it terminates at the point where there is no subsequent read of that value, i.e., the last read point. A register variable has the same physical register assigned to it for the span of one live range. It could have different physical registers assigned to it across different live ranges. You cannot have another write into the same variable in the middle of a live range (otherwise the live range would be split), but you could have multiple reads in the middle of a live range. Once past the last read, the live range is terminated and the physical register can be released.

A register variable can have a write without a read, or a read without a write. For example:

Example A	Example B	Example C

`main()`
`{`
`__declspec(write_reg) x;`
`  SIGNAL s1;`
`  `
`  // read x without`
`  // write to x first`
`mem_write64(&x, &p, 1,`
	`main()`
	`{`
	`__declspec(read_reg) x;`
	`  SIGNAL s1;`
	`  `
	`  // write into x`
	`  // no read of x`
	`  mem_read64(&x, &p, 1, ctx_swap,`
		`main()`
		`{`
		`__declspec(read_reg) x;`
		`  SIGNAL s1;`
		`  `
		`  // write into x`
		`  men_read64(&x, &p, 1, ctx_swap,`
		`&s1);`

Example A	Example B	Example C

```ctx_swap, &s1);```
 ```};```
	```&s1);```
	 ```}```
		```// write into x again```
		 ``` mem_read64(&x, &q, 1, ctx_swap,```
		 ```&s1);```
		 ``` // read x```
		 ```... = x```
		 ```}```

In example A, x is read without being written, so the compiler assumes the live range starts from the beginning of the program. In example B, x is written but not read; the compiler concludes that the write is redundant and can be removed during optimization. Example C is the variation of example B. X is rewritten before the value from the first `mem_read` is used. The compiler can remove the first `mem_read` as a redundant instruction. Even if it is not removed for other reason, the compiler will release the physical register x immediately after this instruction.

In summary, a write always terminates an old live range, and may start a new live range if there is a preceding read. And, a read always extends the live range of a register at least to the read point.

One of the challenges this process brings to the compiler is the implicit read/write of a register. Normally, you can only modify or reference a register through explicitly expressed names, like register x in our previous examples. This process provides you with ways to implicitly read and write a register without referring to the register name. The scenarios include but are not limited to the following:

- A signal/Xfer register is defined on a remote ME and used on local ME. The definition/write is not visible from the local program.

- A signal/Xfer register is defined locally and used on a remote ME. The reference/read is not visible from the local program.

- A signal/Xfer register is assigned an absolute register number. It could be read/written without referring to the symbolic name.

- A signal has the signal number exposed through signals() or signal_number(). It could be read/ written without referring to the symbolic name, for example, though local_csr write.

- A xfer register has address taken and used in indexing reference through T_INDEX (not supported in PR3).

- A NN register that being referenced indirectly though NN register ring.

- A synchronized I/O operation with sig_done. For example:

```
__asm mem_write64[x, &p, 0, 2], sig_done[s1];
...
__asm ctx_arb[s1];
```

Even if this instruction is the last use of the xfer register(s), you should not release the xfer register(s) immediately. The semantics of I/O instructions requires you to hold the xfer register(s) until the signal arrives. So in our example, the live range of x needs to be extended to pass `ctx_arb`.

Under certain scenarios the compiler cannot detect the correct live range of a register. For example, if the compiler prematurely terminates the live range of a register, it could overwrite the value currently in use; or if the compiler prolongs the live range of a register, it could run out of register space unnecessarily. You must supply live range directives for the compiler to make the right decision.

There are four compiler directives for liveness computation:

- `__implicit_read()` will prolong the live range to at least the point where `__implicit_read()` is called. If there are other reads of the same register after this point, then adding `__implicit_read()` does not have any effect on the program.

- `__implicit_write()` will terminate a live range, and start a new one if there are following reads. If `__implicit_write()` is followed by another write, and there is no read in between, then this `__implicit_write()` has no effect on the program.

- "`__implicit_undef()` is similar to `__implicit_write()`. It also terminates a live range at a specified point. The difference is that `__implicit_undef()` does not start a new live range even if the program has reads of the variables following it. The user can assume that at the point `__implicit_undef()` is called, the value of the variable will become undefined and its register(s) can be released.

- `__free_write_buffer()` will free the I/O buffer (xfer register) after this point. See Section 7.4 for examples of the use of `__free_write_buffer()` and `__implicit_read()`.

> ### Note
>
> For syntax, please see the intrinsics in the *NFP-6xxx Micro-C Standard Library Reference Manual*.

4.10 Viewing Live Ranges

The command-line option -Qliveinfo displays live-range information about a set of register classes. You can use -Qliveinfo or -Qliveinfo=all to display live-range information for all register classes, or use -Qperfinfo=<reg_class,...> to display live-range information for a selected set of register class. Table 3.1 explains the syntax of the option in detail.

If a variable is live at some point in a program, this means that the variable's value is used somewhere after that point in the program, and therefore storage (registers in this case) must be reserved for that variable. If too many variables are "live" at the same point in a program, not all of them will be able to be stored in registers, and some of them will have to be "spilled," or demoted, into local memory, NN registers, cluster local scratch or IMEM. This can adversely affect performance. When this happens, -Qliveinfo can help you analyze your program and determine which code segments have a high "register pressure" and thus need to be restructured.

Every object in your program that is assignable to registers, including user variables, is represented in a common format, the "virtual register." Virtual registers have a class, ID number, and optionally a user variable name. Register allocation is the process of mapping virtual registers to physical ones. The - Qliveinfo printout provides information on these virtual registers in the following format:

```
cls.ID(variable_name)
```

where "cls" is one of the register class names specified in the -Qliveinfo parameter set, "ID" is the ID of the VR, and "variable_name" is the name of the corresponding user variable, if any.

-Qliveinfo prints the live-range info for each register class separately. For a given register class, the first part of its live-range display is a mapping from the virtual registers of that class to the corresponding user variable names.

Not all virtual registers can be mapped to a variable name. For those that cannot be mapped, an ellipsis (...) is used instead. Some VRs will correspond to compiler-generated variables (usually of the form "cgt.nnn"). Global variables in your code will have their names modified slightly.

The following is an example of the virtual register map:

```
Virtual gpr Registers to User Variables Mapping:
   nn_inl_01a.c(30)    _x        gr.123
   nn_inl_01a.c(32)    _y        gr.328
   nn_inl_01a.c(32)    _y+4      gr.329
   nn_inl_01a.c(32)    ...       gr.330
```

After the virtual register map, the live-range info is printed for each pseudo-assembly instruction in each function. At the top of each function display, three sets of VRs are printed: the live-in set, the live-out set, and the live-through set. The "live-in" set is the set of VRs that are live at the points where the function is called. "Live-out" is the set of VRs that are either "live-in" or defined in the function, and are used after the function returns. "Live-through" is the set of VRs that are "live-in" and "live-out," but not referenced inside the function. "Live-through" reflects the register pressure from the function's callers. The function-level liveness information is then followed by the live range information for each pseudo-assembly instruction (pseudo-assembly instructions are used internally by the compiler, and correspond roughly with NFP assembly instructions, but may have different syntax).

An example of the live-range info for a function is:

```
Live info. of gpr registers for Function test1b:
   Live in(1):
   gr.671(..)

   Live out(0):

   Live through(0):

   Live set(1): gr.671(..)

   /******/     puthi(S1); put('\n');
1       immed[gr.348(val) , 65535, <<0]
   Live set(2): gr.348(val) gr.671(..)
2       immed_w1[gr.348(val) , 32767]
   Live set(2): gr.348(val) gr.671(..)
3       .mcall[_puthi#, gr.663(..) ]
   Live set(1): gr.671(..)
4       immed[gr.328(c) , 10, <<0]
   Live set(2): gr.328(c) gr.671(..)
5       .mcall[_put#, gr.655(..) ]
   Live set(1): gr.671(..)

/******/ puthi(S1/(1<<0)); put('\n');
6       immed[gr.348(val) , 65535, <<0]
   Live set(2): gr.348(val) gr.671(..)
```

Each pseudo-assembly instruction is preceded by a set of the VRs that are live at the point before the instruction executes. The numbers that occur after the set names ("Live set(n)") indicate the number of relative registers needed to allocate the live VRs at that point. "Shared" VRs or variables will count as a fraction of a register, because several absolute registers can be mapped into a single relative register. In the above example, the VR gr.348 is live between

instructions 1 and 3, and live after instruction 6. Instruction 1 writes to the VR, which makes it live afterward (recall that liveness is an indication of whether a VR's value is needed). Instruction 3 is a function call to puthi(), which uses the value of gr.348 (this fact would be apparent on examination of the liverange info of the puthi() function). After the function call, gr.348 is no longer live because its value was only needed for the function call. gr.348 becomes live again after instruction 6, because the VR is rewritten with different value for the next puthi() call (which is not shown).

4.10.1 Limitations and Restrictions on Viewing Live Ranges

- Pseudo instructions do not exactly reflect the assembly instructions in the final list file. Register allocation, local memory allocation, scheduling, and other optimizations might delete, add, modify, or reorder some instructions. Some of the pseudo instructions are compiler directives only, and will not produce any physical instructions in the final list file.

- At a given program point, register pressure slightly greater than the number of available registers does not necessarily mean that a spill (demotion to another storage class) will be generated for one of the live variables. Some variables may be found to be equivalent to each other and may share the same register. Conversely, register pressure which is slightly smaller than the number of available registers does not guarantee that no spilling is needed, because additional registers may need to be allocated to perform operations such as memory I/O, or to temporarily hold values from other register classes.

- Not all virtual registers can be mapped back to user variables.

- Some variables have modified names that cannot be found in the original source code. They are either global variables or compiler-generated variables.

- When the compiler allocates registers to variables, transfer registers are allocated first. If there are not enough transfer registers to hold all the live transfer register values at a given program point, some of the values will be stored (spilled) in GPRs instead. This may in turn cause spills (demotion to NN registers or memory) in the GPRs that are live at that point. Since the -Qliveinfo output is printed before register allocation, it might show a GPR register pressure which is smaller than the number of available registers at a given program point, even though a spill is generated at that point. If this happens, the register pressure of the transfer registers should be examined. Any transfer register pressure larger than the number of physical registers should be added into the GPR register pressure.

- If your code contains local memory variables, 2 GPRs are reserved from the allocation pool, so that local memory address calculations can be performed.

- Variables declared as volatile are marked as live at every point within their scope. Functionlocal volatile variables are live within the scope of their defining function, and global volatile variables are live within the scope of the entire program.

4.11 Unsupported ANSI C99 Features

The compiler is largely based on the ISO/IEC 9899:1990 (C89) standard, with selected features from ISO/IEC 9899:1999 (C99). The majority of the new features in C99 are not supported in the current implementation of the compiler, which include the following:

- variable-length arrays

- flexible array members

- static and type qualifiers in parameter array

- complex and imaginary data types

- compound literals

- designated initializers

- preprocessor arithmetic done in intmax_t/uintmax_t

- mixed declarations and code

- new integer constant type rules and integer promotion rules

- macros with a variable number of arguments

- trailing comma allowed in enum declaration

- inline functions (compiler does implement __inline, which is similar to the inline keyword defined in C99)

- boolean type (compiler defines bool as int, which is not the same as _Bool defined in C99)

- idempotent qualifiers

- empty macro arguments

- new struct type compatibility rules

- _Pragma preprocessing operator

- __func__ predefined identifier

- VA_COPY macro

- LIA compatibility annex

- conversion of an array to pointer not limited to lvalues

- relaxed constrains on aggregate and union initializations

- flag error on return without expression in functions return value (and vice versa)

- no implicit function declaration.

In addition to these unsupported features, the compiler has the following restrictions:

- It does not support any floating type, including float, double, and long double.

- It does not support recursive function calls.

- It does not support variable function arguments.

- It does not support setjmp/longjmp.

- It does not support any signal() functions.

- It does not support taking function address and using function pointers

- Due to the size of control store, it does not support translation limit (such as the number of identifiers, number of nested blocks, etc.) defined in C89 and C99.

- An array subscription on register variables (gp_reg, nn_reg) can only take a constant value. Transfer registers can take variable indexes but there are limitations. Please refer to Section 4.3.2.

- The address operator (&) can not be applied to register variables (gp_reg, nn_reg or xfer_reg), except as an argument to intrinsic function calls.

- Read xfer register variables can not be used as an Lvalue except in a call to intrinsic functions.

- Write xfer register variables can not be evaluated (used as Rvalue) except in inline assembly.

- Most standard library headers defined in C89/C99 are not supported at this time. Only `<memory.h>`, `<string.h>` and `<stdlib.h>` are implemented with significantly fewer functions supported than required by C89/C99.

- A limited number of standard library functions are implemented in the compiler. Please refer to `<memory.h>`, `<string.h>`, and `<stdlib.h>` in the *Netronome Network Flow Compiler LibC: Reference Manual* for information on which functions are implemented.

5. Inline Assembly Language

The compiler supports inline assembly language through the `C __asm` statements and blocks, providing full access to the compiler microengine instruction set. Inline assembler instructions must refer symbolically to variables declared in C.

All references must be symbolic—the compiler then assigns the registers. It does not allow references to specific machine registers.

As a relatively new feature, file-scope __asm blocks are also supported. These allow selected assembler directives, mostly those accessing linker/loader features, to appear in the list file. It is important to note that these "file-scope __asm blocks" (as they are always called here) are completely different than regular __asm statements and blocks. The file-scope __asm blocks support only non-executable directives (in other words, assembler pseudo-ops) whereas regular __asm statements and blocks support only executable code ("microwords"). File-scope __asm blocks are covered in Section 5.7. Elsewhere, when __asm statements and __asm blocks are mentioned, the reference is always to regular __asm statements and blocks.

Various non-standard preprocessor directives which are supported for the microcode assembler are not supported by the C compiler.

> **Note**
>
> The compiler checks for hazards in the inline assembly sequence and flags any hazards found as errors. The philosophy in the compiler is to not interfere with inline assembly code in a way that would make the code worse from a performance point of view. The assumption is that users who write inline assembly are sophisticated in programming at the microcode level and generally do not want the compiler to modify or rearrange inline assembly. The only exception to this is that the compiler may perform some low level optimizations and will try to fill delay slots.

5.1 Single __asm Instruction

For a single instruction, the following form is accepted:

```
__asm <instruction>;
```

5.2 Block of __asm Assembly Code

For a block of assembly code, the following is accepted:

```
__asm __attribute(CONSTANT)
{
label: <instruction>; comment
```

```
    .
    .
    .
}
```

The allowed values for the "CONSTANT" block attribute are ASM_HAS_JUMP and LITERAL_ASM. The LITERAL_ASM attribute disables all compiler optimizations in the assembly block and allows the use of the defer[] keyword. The ASM_HAS_JUMP attribute is used when the assembly block contains the jump[] instruction and is further described below. Assembly block attributes can be combined with the OR (|) operator.

> **Note**
>
> The label, comment, and attribute value are optional. The attribute value, if specified, must be a constant. The semi-colon begins a comment in the asm block that continues up to and including the end-of-line character.

The following fragment shows nested __asm blocks being used in a macro definition. (Semi-colons would be out of place here since they begin comments.)

```
#define yield(a,b,l) \
__asm __attribute(LITERAL_ASM) { \
  __asm { alu[--,a,-,b] } \
  __asm { bne[l] } \
  __asm { ctx_arb[voluntary],br[l] } \
l: \
}
```

5.3 Instruction Format

An inline instruction has the following format:

```
opcode [operand, operand...], keyword,...
```

The allowed operands and optional keywords depend on the opcode.

Function calls are supported using a pseudo instruction CALL that takes a single argument of the function name:

```
CALL[foo]
```

The compiler expands this into storing the return address and branching to the function.

Inline assembly blocks containing jump[] instructions need to be marked with the attribute constant ASM_HAS_JUMP (defined in `ixp.h`). The jump[] instructions themselves also need to contain a list of possible targets. The maximum number of jump targets is 120.

For example:

```
__asm __attribute(ASM_HAS_JUMP)
{    alu_shf[offset, K, +, I, <<1]; // example offset calculation
     jump[offset, base], targets[L1, L2, L3] // jump to L1, L2, or L3
     L1:
     ...
     L2:
     ...
     L3:
     ...
}
```

This added information allows the compiler to properly allocate registers and optimize code within the assembly block.

5.4 NFP Enhanced Mode Instructions

Refer to the "NFP Enhanced Mode Instructions" section in the *Netronome Network Flow Processor 6000: Network Flow Assembler System User's Guide* for detailed information on these instructions.

5.5 Operand Syntax

Each operand to the instruction specifies one of the following:

- a register

- an immediate value (constant)

- a label

- a keyword, such as memory or ALU operation

The specific type of each operand is determined by the opcode.

5.5.1 Register Operands

When a register operand is called for, you must supply a special hardware register, such as a CSR name (where this is applicable) or a C variable that is in a register, that is either a local, an argument, or a global variable without a memory attribute (MEM, CLS or local memory).

The register variable must be an integral type (no char or short or __int64). After setting up the local memory address CSRs, you can use *l$index0 or *l$index1 as register operands to address local memory. If the variable specifies a struct or union type, you can use the "." notation to select a field. The field must be a full 32-bit item. Fields of type char or short, and bit fields cannot be referenced in this manner. The variable name cannot be the same as a reserved token in the microcode language. This includes, for example, all the tokens that arise from the

ALU opcode such as B, A, CARRY, AND, OR, XOR, IFSIGN or common names like csr and inp_state etc. These tokens are case insensitive and hence neither variable "a" nor "A" is permitted as a register operand. Note that B, and A are reserved tokens since "B-A" is an ALU opcode and white space is permitted between "B", "-", and "A". Other examples of reserved tokens are (but not limited to) "csr", "state", and "inp_state".

Objects in registers can be addressed with a byte offset by adding a "+n" to the object name. For example:

```
__declspec(gp_reg) int thing[10];
__asm {
      alu[thing+12, --, B, 4]; // thing[3] = 4
}
```

If the object is assigned to general-purpose registers, any byte offset can be used, regardless of alignment. The compiler will generate the proper mask-and-shift operations to perform an unaligned access, if necessary. Objects in transfer registers or remote next neighbor registers cannot be accessed with unaligned offsets.

When setting up an argument for a function call, you can refer to the registers used to pass the argument using the following notation:

```
funcname.arg_n
```

where n = 0 for the first (leftmost) argument in the function header.

To get the return value from a function after a call, use the following:

```
funcname.ret
```

If an argument or return value is a struct, you can cast it to the struct type and select a field as follows:

```
funcname.arg_n:structname.field
```

The example in Section 5.5.3 depicts usage of C variables, structure field access, and function call setup within inline assembly.

When calling functions inside inline assembly blocks, you must take care to pass the arguments in the way that the function expects. The compiler will treat all such arguments as untyped register values and will not perform type checking.

For operands that span more than one register, you can use an offset of 4 bytes for each additional register, for example:

```
foo+8
```

refers to the third register occupied by foo.

5.5.2 Immediate Operands

Immediate operands are constant expressions using C syntax. All of the C operators are supported, including the sizeof() macro. An "offsetof(struct, field)" macro is also supported, which returns the byte offset of a given field in a struct.

Simple "constant folding" is supported. For example, "3+4" will be replaced with the constant "7".

The address of a memory variable can also be specified as an immediate operand, using the & operator. Since immediate operands can be no more than 16 bits, you can get the high order 16 bits of an address by using the >> operator as follows:

```
&buffer >> 16
```

> **Note**
>
> See the *Netronome Network Flow Processor 6000 Databook* for allowable opcodes, operands, and keywords.

> **Note**
>
> A non-zero constant beginning with zero (0) is interpreted as an octal number. Thus, for example, the byte mask in __asm {ld_field[dest, 0011, src, >>16]} will be interpreted as 9. To express the mask in binary use the "b" or "B" suffix, as in __asm {ld_field[dest, 0011b, src, >>16]}. To express the mask in hex, use 0x or OX as a prefix, as in __asm{ld_filed[dest, 0x11, src, >>16]}.

5.5.3 Usage Examples

Example

This example illustrates how to call a C function from inline assembler code and access structure fields.

```
typedef struct
  {
     int a;
     int b;
  } a_struct;

  int a_func(int x, a_struct str)
  {
     return x + str.a + str.b;
  }

  int call_a_func(int y)
  {
```

```
    int ret;
    __asm
    {
    alu [a_func.arg_0, --, B, y]; pass y as x
    alu [a_func.arg_1:a_struct.a, --,B,1]; pass 1 as str.a
    alu [a_func.arg_1:a_struct.b, --,B,2]; pass 2 as str.b
    call [a_func]

    alu [ret, --, B, a_func.ret]; return value
    }
    return ret;
}
```

Example

This example illustrates different errors in usage.

```
    __declspec(imem) int x;
    int y;
    __declspec(local_mem) B, C;
    struct { char c; short s;} st;

    foo(&y, &C);
    __asm {
       alu [x, C, +, C]; Error since x has been allocated to IMEM
       alu [st.c, st.s, +, st.s]; Error since s.c and s.s are of
                                  type char, and short
                                  respectively
       alu [B, C, +, C];          Error since B is a reserve
                                  keyword in microcode
    }
```

5.6 Restrictions on Use Of Assembly Language

The compiler can not handle all possible uses of assembly language. The following restrictions are in place:

- The target of a jump instruction must reside in the same inline assembly block as the branch.

- A goto statement in code cannot branch into inline assembly code.

- Instructions that override the register address for transfer registers resulting in the registers that are not local to the current thread do not work properly.

- funcname.arg_n can not be used as a source operand, funcname.ret can not be defined. Sequences of funcname.arg_n definitions must be followed by inline-asm call to funcname without other intervening call(s) in between. All parameters to inline-asm call must be defined before call.

- The DEFER keyword can only be used in inline assembly blocks that are tagged with the LITERAL_ASM attribute.

- Code within an __asm block may not depend on the value of processor flags (e.g. +carry, condition codes), generated by C code outside the __asm.

- Register operands must be of integral type (no char, short, or __int64)

- The br[] token on the ctx_arb[] instruction is not supported

- Local memory variables cannot be accessed in inline assembly using their "C names", because the access may need to be expanded to several instructions. Local memory can be accessed directly using the local memory pointer CSRs.

- Code that depends on a specific clock cycle timing to read or write different CSR values may not function correctly. For example:

```
local_csr_write[active_lm_addr_0, a0]
local_csr_write[active_lm_addr_0,b0]
nop
nop
alu[$1, a2, +, *l$index0] //expecting to use the old value
alu[$2, a3, +, *l$index0] //expecting to use the new value
```

The compiler may insert additional NOP instructions to maintain the 3-cycle latency between the CSR writes and the CSR accesses, which would disturb the timing of the above code.

In general, array indexing in inline assembly language statements is best avoided since support is incomplete. If used, correctness should be verified on a case by case basis.

The optional tokens `gpr_wrboth` and `predicate_cc` are not supported in inline assembly language statements. (The token `predicate_cc` was originally partially supported.)

Further restrictions may be specified in the future.

5.7 File-Scope __asm Blocks

In addition to the __asm statements and blocks described in preceding sections, special file-scope __asm blocks are also supported. Whereas regular __asm statements and blocks must appear within functions and may contain only executable code, file-scope __asm blocks must appear outside functions and may contain only assembler directives (pseudo-ops). The following example shows a file-scope __asm block:

```
__asm {
  .alloc_mem res0 i24.emem+0x10000000 global (0x100000*32) reserved
  .alloc_mem res1 i24.emem+0x20000000 global (0x100000*32) reserved
  .alloc_mem res2 i24.emem+0x30000000 global (0x100000*32) reserved
  .alloc_mem res3 i24.emem+0x40000000 global (0x100000*32) reserved
}

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

Although reference is made throughout this section to "__asm blocks", file-scope __asm statements (in one-line form, without braces) are also permitted.

The overall idea of a file-scope __asm block is to provide compatible support in the compiler for a subset of assembler directives, thus allowing the same configuration, declaration and initialization statements to appear in both microcode and Micro-C programs. Users are accordingly referred to the Assembler User's Guide for documentation of the supported directives. What appears here relates mostly to the basic approach, and to limitations and exclusions.

As a general principle, whereas the assembler will process directives in context, taking into account other directives already enountered, the compiler, in file-scope __asm blocks, processes directives only in terms of their syntax, and line-by-line in isolation. Some assumptions are therefore made. For instance, a basic assumption is that a symbol enountered in a bracketed expression refers to a linker variable rather than a load-time constant.

A file-scope __asm block should, in general, not refer to Micro-C variables and other structures (and it may not even be possible to do so). For example, if allocated memory is needed for some directive such as `.declare_resource`, use should be made of `.alloc_mem`. A special Micro-C variable should not be set aside.

A compiler intrinsic function, `__link_sym`, is supplied to provide access from Micro-C to linker-allocated resources set up within file-scope __asm blocks and elsewhere. This works somewhat like `__LoadTimeConstant` but for linker symbols.

The following assembler directives are supported in file-scope __asm blocks: `.alloc_mem`, `.alloc_resource`, `.declare_resource`, `.init`, `.init_csr`, `.init_mu_ring`, and `.load_mu_qdesc`. The `.init` support is limited to memory.

Only these operators are supported in bracketed expressions:

Unary operators: `! ~ + -`

Binary operators: `* / % + - << >> < <= >= > == != & ^ | && ||`

Some further things to bear in mind are:

- Integer constants must be in standard C form.
- There is no support for assembler functions.
- No case-folding is done on identifiers and other names.

6. Compiler Optimizations

The compiler optimizes for size, speed, or debugging. It optimizes at the function level or on a whole program level.

6.1 Machine Independent Optimizations

The compiler does several machine independent optimizations including:

- Inlining and whole program constant propagation.

- Traditional scalar optimizations to clean up the code and remove redundant computations.

- Loop unrolling/peeling—reduces taken branches and provides optimization opportunities.

- Constant and copy propagation:

  Example:

  `y=5; x=y; foo(x);` In `foo(x)` `z=z<<(x+1)` is optimized as `z=z<<6`.

- Removal of dead stores—stores to memory locations that are not referenced are eliminated. These are not applied to variables declared volatile. The assignment to x is eliminated in the following example:

  `main() {__declspec(imem) x; x=5; return}`

6.2 Network Processor Specific Optimizations

6.2.1 Registrations

All local and global variables are kept in registers unless:

- The address of the variable is taken and can not be propagated to the dereferencing site or used for something else than simple dereferencing.

  Example:

  `int x, *p; p=&x; /* p is in a register, x is in IMEM */`

- The variable is declspec-ed to a memory region.

  Example:

  `__declspec(imem)int x, *p; /* p is in a register, x is in IMEM */`

- Registers need to be spilled.

- Transfer registers are spilled to GPRs
- GPRs spilled to local memory or IMEM.

6.2.2 Read/Write Combining

Multiple reads or writes can be combined into one read or write respectively, if the reads/writes are to contiguous memory locations. This is done with external memory operations specifying a reference count greater than one.

Example:

```
struct{
int a, b;
}
__declspec(imem) s;

x=s.a+s.b; //Only one IMEM read is generated with a two-word count
```

Although the two accesses to the IMEM structure are distinct in the C code, they are combined by the compiler into a single IMEM read instruction. One IMEM read of two words will complete faster than two separate IMEM reads, especially when the reads are blocking (ctx_swap) reads.

6.2.3 Peephole Optimization

Some instructions on the Netronome network processor architecture can perform multiple operations simultaneously. The compiler will try to use these instructions whenever possible.

Examples:

- Combining shift and logical operations.

  `dest = x & (j << 2)` becomes `alu_shf[dest, x, AND, j, << 2]`
- Combining add and mask operations:

  `dest = x + (j & 0xff)` becomes `alu[dest, x, +8, j]`

6.2.4 Defer Slot Filling

Some multi-cycle instructions on the Netronome network processor architecture contain "defer slots" that allow other instructions to be executed while the slower instruction is being processed. The compiler will take advantage of these defer slots when possible.

For example:

```
z += 5;
if (x) goto label;
```

becomes:

```
alu[x, --, B, x]
beq[label#], defer[1]
alu[z, z, +, 5]              ; always executed
```

6.2.5 Local Memory Grouping

The compiler attempts to allocate local memory variables used in the same control flow region into one continuous group and controls access to those variables through a single local_csr_write instruction. For information on how this is done, refer to Section 4.3.8.

6.2.6 Local Memory Autoincrement/Autodecrement Conversion

The compiler will generate autoincrement and autodecrement addressing for local memory accesses in loops whose addresses vary by a constant amount. For example:

```
__declspec(local_mem) int arr[10];
for (i = 0; i < 10; i++)
{
    arr[i] = i;          // address of LM access increments by
                         // 1 word on every iteration
}
```

becomes:

```
    local_csr_wr[active_lm_addr_1, arr]
    nop
    nop
    nop
l_2#:
    alu[*l$index1++, --, B, i]
    alu[i, i, +, 1]
    alu[--, i, -, 10]

    blt[l_2#]
```

Previously, such accesses were performed by writing the local memory index register on each access, which would cause up to a four-cycle delay on every loop iteration.

This optimization is disabled by an option, "-Qlm_unsafe_addr". This should be used when local memory pointers are written with invalid values and accessed conditionally. For example:

```
__declspec(local_mem) int p[100];
int i;

for (i = -100; i < 100; i++)
{
    if (i > 0)
    {
    ... = p[i];
    }
}
```

For the first 100 iterations of the loop, "p[i]" is not a valid local memory address. The above example is correct code because the loop checks if "i" has a valid value before performing the array access, but the local memory address optimizer may generate:

```
local_csr_wr[active_lm_addr_1, "&(p[-100])"]
nop
nop
nop
top:
    alu[--, --, B, i] // if (i > 0)
    ble[skip#]
... [..., *l$index1] // ... = p[i]
skip:
    alu[--, --, B, *l$index1++] // p++ and i++
    alu[i, i, +, 1]
alu[--, i, -, 100]
blt[top#]
```

The local memory index register is initialized to "&(p[-100])", which may be negative, and incremented by 1 word every iteration. This is "functionally correct" but may not work correctly on actual Netronome network processor hardware, since signed arithmetic is not guaranteed to work on local memory index registers.

If you have code such as the above, where a local memory array or pointer expression is intentionally set to point to a value outside of the allowable 0-1024 address range, you should turn off local memory postinc/postdec optimization with -Qlm_unsafe_addr.

6.2.7 Scheduling

The compiler may re-order instruction sequences to reduce nops; otherwise, it needs to maintain correct latencies between performance. Listed below are some cases where nops are needed.

• Between writes to local_csr and the use or read of the same local_csr, included, but not limited to the following:

  • Writes to lm_addr CSRs and *l$index0 or *l$index1

  • Writes to T_index CSRs and *$index or *$$index

  • Writes to BYTE_INDEX CSRs and byte_align_be or byte_align_le

  • Writes to NN_Put/NN_get and *n$index

• Between instruction setting condition code and bcc on it with defer[3]

- Between crc operations, and between the last crc and the use of crc remainder

- Between definition of nearest neighbor and use of the same nn in self-mode

- Between two local_csr writes to the following 3 local_csrs: next_neighbor_signal, prev_neighbor_signal, and same_me_signal

The scheduler assumes that you write serialized code without knowledge of pipeline latencies, which might change from generation to generation, which results in non-portable code. The Hazard Detector inserts proper nops in -Od when the scheduler is turned off.

6.2.8 I/O Parallelization

Multiple I/O operations with the ctx_swap token can be converted to use sig_done followed by a ctx_arb to wait a mask of their signals, if instructions between I/O and the inserted ctx_arb, if any, do not access other global states, and if doing so won't create more spilling. The compiler may use an existing ctx_arb rather than inserting a new one if one is found and the above rules are satisfied. If I/O with the ctx_swap token is generated by the compiler from C statements (as opposed to intrinsics or inline-assembly), meaning user has no expectation of ctx_swap, the restriction about accessing global states may be relaxed. For example:

Original sequence of code:

```
cls[read, sr, &mem1, 0, 1], ctx_swap[sig1]
mem[write, dw, &mem2, 0, 4], ctx_swap[sig2]
mem[read, dr, &mem3, 0, 2], sig_done[sig3]
ctx_arb[sig3]
```

Can be converted into:

```
cls[read, sr, &mem1, 0, 1], sig_done[sig1]
mem[write, dw, &mem2, 0, 4], sig_done[sig2]
mem[read, dr, &mem3, 0, 2], sig_done[sig3]
ctx_arb[sig1 sig2 sig3]
__free_write_buffer(&dw);
```

The CLS read and write operations can be executed in parallel with the MEM read. This is possible because the code between the MEM read and the ctx_arb instruction meets the conditions described above. Namely, the transfer register "sr" is not read, there are no global states accessed, and no spills will be caused by extending the live ranges of the transfer registers and signals.

7. Tips for Optimization, Troubleshooting, and Debugging

This chapter has two goals: to help you develop high performance code, and to provide you with troubleshooting and debugging information. The chapter will cover three main topics:

- Potential pitfalls for ANSI C programmers new to programming network processors. These are topics for which unaware programmers can unwittingly prompt the compiler to generate incorrect code in relation to their actual intent.

- Tips to help you elicit the best possible code from the compiler. These topics are for you to keep in mind when coding, although *not* implementing them will not necessarily result in incorrect code, but simply suboptimal

- Compile time and run time features of the network processor development tools that are invaluable for you to debug your Microengine C code. These features are part of both the compiler and the Programmer Studio GUI.

> **Note**
>
> You should have a strong understanding of the network processor architecture as well as a firm knowledge of Microengine C syntax. If not, please consult the latest appropriate hardware reference manuals and programmer reference manuals (see Section 1.2).

7.1 Software Implementation Considerations

The Microengine C compiler provides a high-level-language programming environment for the network processors to reduce application development time and reduce the need for specialized knowledge. There are specific considerations that you need to be aware of to program with confidence in terms of correct Microengine C behavior and performance. Many of the issues that are discussed in this chapter do not exist within a general-purpose compiler that targets a generalpurpose processor, and therefore might not be self-evident.

7.1.1 Variable Live Range Analysis

Register allocation and other compiler optimizations depend on having correct live range information for register variables. A live range of a register variable is the period between the definition of this variable and the last use of the defined value. When a register variable has multiple definitions in the program and each definition has sequential reads, multiple live ranges are assigned to the same variable. It follows that multiple reads in the middle of live range are fine, but a write into the same variable in the middle of a live range will split it.

The compiler automatically calculates the live range of a register variable through code analysis based on the fact that a live range always starts with a write into the variable and terminates at the point where there is no subsequent read of this written value (i.e. the last read point). A register variable has the same physical register assigned to it for the span of one live range; however, it could have different physical registers assigned to it across different live ranges.

(For more information on how the compiler calculates variable live ranges, refer to Section 4.9.)

There are times, however, where the compiler will not be able to calculate the live range of a variable correctly. Specifically, you will have to intervene when a variable is implicitly read or written at a point in the code where the variable is not referred to by name. For example, some asynchronous memory operations or event signaling can be done in such a manner. In these cases, the compiler has no way to determine the true start or end of the live range through code inspection. This situation can lead to suboptimal register allocation, or worse, incorrect code generation.

There are some major code constructs that merit the use of the __implicit_read() and __implicit_write() intrinsic functions to help the compiler with live range analysis. These intrinsics do not generate any new code, but rather, allow you to manually extend or shorten the live range of a variable by providing a clue to the compiler as to when a register or signal is being accessed outside of the scope of a particular thread's code.

The following sections provide several specific examples as to when the you need to intervene in live range analysis.

7.1.1.1 Asynchronous I/O Operations

Asynchronous I/O operations are those that read or write into a variable that is not explicitly under compiler control. Such situations can arise with the use of the "sig_done" token with memory intrinsics, and also in cases where a signal or transfer register is defined on one microengine (ME) but accessed from another.

Example:

```
SIGNAL sig1;
SIGNAL_PAIR sig_pair;
__declspec(read_reg) int sr1[4];
__declspec(read_reg) int sr2[4];
cls_read(sr1, 0, 4, sig_done, &sig1);
mem_read(sr2, 0, 4, sig_done, &sig_pair);
wait_for_all(&sig1, &sig_pair);
sum += sr1[0] + sr2[0];
```

At first glance, nothing seems amiss, but in reality the compiler will determine that the program is not using the entirety of the buffers sr1 and sr2, and truncate the live range of individual members of an aggregate accordingly. That is, we write into these variables with the CLS and MEM reads, but do not consequently read all of the array elements (in this snippet, only sr1[0] and sr2[0] are read). So, it may attempt to conserve registers by assigning overlapping register ranges to sr1 and sr2. For example, transfer registers $0 through $3 may be assigned to sr1, and $1 through $4 may be assigned to sr2.

This is correct if the two memory reads complete sequentially (in order), since the sum operation only uses sr1[0] and sr2[0], which are $0 and $1, respectively. However, if the cls_read() and mem_read() operations complete *out of order*, then this assignment will cause problems. Specifically, when the mem_read() operation completes, the data the program needs will be read into $1. But when the cls_read() operation completes, the contents of $1 will be overwritten by the four-word read operation starting at $0. In short, the compiler does not rely on characteristics of specific implementations of the network processor, but rather makes decisions based upon the general network processor architecture and the syntax of the C language.

You must avoid this situation by informing the compiler that the entirety of both transfer buffers is being used with the __implicit_read() intrinsic.

One example is as follows:

Example:

```
wait_for_all(&sig1, & sig_pair);
sum += sr1[0] + sr2[0];
__implicit_read(sr1);
__implicit_read(sr2);
```

In this way, the compiler will extend the live range of all array elements of sr1 and sr2, and thereby not overlap the register allocation.

Of course, if this example only included the one asynchronous cls_read and wait_for_all pair, there would be no problem. It is usually only in the presence of multiple asynchronous memory operations that there could be a problem. As a general rule of thumb, when using a read memory intrinsic with the sig_done token, you should place an __implicit_read after the matching wait_for_any or wait_for_all, to manually extend the live range of the memory variable to the correct point in the code.

Other asynchronous reads and writes to transfer registers or signal variables can occur in the following situations:

- A signal or transfer register that is defined on a remote ME and used on local ME—the definition and write is not visible from the local program

- A signal or transfer register is defined locally and used on a remote ME—the reference and read is not visible from the local program

- Special chip hardware that is designed to "push" data into a transfer register or send a signal

Most I/O instructions can overwrite the ME/CTX/XFER through the use of the indirect_ref token, causing the signal and transfer register to be used in the operation another ME or context—and therefore must be defined there. Additionally, the use of the reflector hardware implicitly will read or write from the transfer register of one thread to another. In the case of signal variables, asynchronous "reads" or "writes" can also be the result of using local or chip-wide thread signaling mechanisms.

> **Note**
>
> A "write" of a signal variable is defined to be the point at which a program asks for a signal to be generated from a chip resource, such as a memory controller. A "read" of a signal variable is defined to be the point at which a program waits on a signal to return (ctx_arb), or branches on a signal (br_signal or br_!signal).

7.1.1.2 Indirectly Referencing a Variable

Normally, all access to a variable declared to be in a register or signal storage class in Microengine C is done by explicitly referencing the variable name. For example, one can declare a variable in a transfer register and assign it a value with the following code:

Example:

```
__declspec(write_reg) U32 wr_xfer0;
wr_xfer0 = 0x1234;
```

However, access to some types of variables can be done without referring to the variable's name, through special hardware support. Some of the software techniques used in conjunction with this support is used for performance reasons, while in some cases, it is actually required to perform a certain hardware function. These hardware/software constructs and features hold a special challenge for the compiler to correctly determine the live range of the variables in question.

The following cases fall into this category:

• A signal or transfer register is assigned an absolute register number and read/written without referring to the symbolic name

• A signal has the signal number exposed through signals() or signal_number() and read/written without referring to the symbolic name for example, though local_csr_wr

• A transfer register has the address taken and used in indexing reference through T_INDEX

• An NN register that is being referenced indirectly though NN register ring

• A transfer register is read or written, or a signal is sent via the cap calculated addressing instruction from another thread.

Example:

```
__declspec(write_reg) U32 wbuf[4];
__declspec(cls) U32 cls_fun[16];
U32 ti0 = ((__ctx()<<4) | __xfer_reg_number(wbuf)) << 2;
__asm {
        local_csr_wr[T_INDEX, ti0]
        nop
        nop
        nop
        alu[*$index++, --, B, cls_fun[index]]
        alu[*$index++, --, B, cls_fun[index+1]]
        alu[*$index++, --, B, cls_fun[index+2]]
        alu[*$index++, --, B, cls_fun[index+3]]
        }
cls_write(wbuf, addr, 4, ctx_swap, &sig);
```

In this case, the compiler would make the determination that the live range of the wbuf array elements would start and end at the cls_write call. That is, it would only determine that wbuf was being read—as a result of the cls_write call, but not ever written into—as the write happens through the use of the indirect addressing capabilities of the hardware. Consequently, the compiler would only allocate a physical register to wbuf on the one line, missing the write into wbuf via the TINDEX CSR.

The correct course of action is to place an __implicit_write() intrinsic before the point of writing into wbuf indirectly, as in the following example:

Example:

```
__implicit_write(wbuf);
__asm {
        local_csr_wr[TINDEX, wbuf]
        nop
        // etc..
```

Now the live range of wbuf will start at the __implicit_write call and correct register allocation will take place.

Example:

```
SIGNAL sigRxEvent;
__declspec(read_reg) rsw_pos_phy_t rsw[2];
int rxEventAddr= __signal_number(&sigRxEvent);
int rsw0Addr = __xfer_reg_number(&(rsw[0]));
rxFreeListReg = rsw0Addr | (thread << THREAD_BITS) |

(this_me << ME_NUM_BITS) | (rxEventAddr << SIGNAL_NUM_BITS);

// Add the thread to the receive freelist
AddThreadToFreelist(rxFreeListReg);

// Wait for signal from MSF receive hardware
wait_for_all(&sigRxEvent);
```

Another alternative to using the __implicit_write() and __implicit_read() intrinsics is use of the volatile keyword. The volatile keyword essentially extends the live range of the variable through the entirety of the program, bypassing live range issues altogether.

However, this method is not recommended in general for variables to be allocated to registers or local memory, as these scarce resources will be eaten up very quickly this way! That said, for variables that truly need a live range over the whole program, declaring them to be volatile (as in the case of the rsw variable in the above example) is the alternative to placing a pair of __implicit_write() and __implicit_read() at the beginning and end of a program.

So, in the previous example, declaring the signal and transfer registers used with the receive hardware to be volatile assures correct code behavior, as shown in the following example:

Example:

```
volatile SIGNAL sig_RxEvent
volatile __declspec(read_reg) rsw_pos_phy_t rsw[2];
```

7.1.1.3 Memory Intrinsics with Variable ref_cnt

The various memory intrinsics provide unfettered access to all of the special features provided by the memory controllers. For intrinsics that take a "count" parameter, you should use a constant reference count in the arguments. If not, the compiler will generate a memory reference with an indirect_ref token, which may or may not lead to the desired results. The reason is that the compiler does not know the exact size of the transfer register buffer to be used in the operation and will make a conservative guess of one without programmer intervention.

Example:

```
void main()
        {
          __declspec(read_reg) mdata[4];
          unsigned int data_cnt;
          mdata[0] = 0x29;
          mdata[1] = 0x39;
          mdata[2] = 0x49;
          mdata[3] = 0x59;
          // Assume data_cnt has not been optimized to a constant
          cls_write(mdata, addr, data_cnt, ctx_swap, sig_srwr);
          // More code...
        }
```

When the compiler calculates live range for the mdata array elements, it makes the conservative guess that only mdata[0] is being read with the cls_write intrinsic. Unless mdata[1], mdata[2] and mdata[3] are read later by another instruction, their live range will begin and end with their assignment, and unknown data will be written to CLS.

There are two possible courses of action. One is to use an __implicit_read to extend the live range of all members of the mdata array, as shown in the following example:

Example:

```
cls_write(mdata, addr, data_cnt, ctx_swap, sig_srwr);
__implicit_read(mdata);
```

The other option is to use the indirect reference form of the memory intrinsic:

Example:

```
srsw.refcount = data_cnt;
srsw_ind.ov_ref_count =1;
cls_write_ind(mdata, addr, 4, srsw_ind, ctx_swap, sig_srwr);
```

The third parameter is the maximum number of longwords that could be written, and must be a constant. The compiler will use that value as the potential length of data buffer used in the operation—even if at run time less longwords are actually written to memory.

Finally, note that the -Qperfinfo=128 command-line option will warn you if the compiler cannot determine the size of a memory option and needs to generate an indirect_ref token. An example of the output is as follows:

Example:

```
myfile.c(45): warning: cls_write(): Size of data access cannot be
determined at compile-time. __implicit_read/write may be needed to
protect xfer buffer. Use of cls_write_ind() is recommended instead.
```

Again, the goal of this warning is to remind you to more precisely provide the liverange information for the variable used in the intrinsic. It is recommended that this option be used for all compilations.

7.1.1.4 Live Range Summary

The live range of a variable starts at the first point in the code it is written to and ends at the last read. In the case of signal variables, this should be interpreted as the period extending from when one asks for a signal to the time when that signal is consumed.

In special cases, the compiler cannot determine the live range of a variable correctly by itself. The most common cases are asynchronous I/O references and access to variables through indirect addressing. For these instances, your program needs to intervene with the use of either the __implicit_read() or __implicit_write() intrinsics or the volatile keyword to extend or truncate the live range of the variables in question.

7.1.2 Data Alignment

The question of how and where data is placed into various storage types is of utmost importance for two reasons. First is to make the most efficient use of a given, and presumably scarce, storage type. The second is to guarantee the correct behavior of the compiler accessing variables indirectly (i.e. through pointers) or data not allocated by the compiler (i.e. packet data off the wire).

In short, misunderstanding how the compiler allocates variables to a storage type and with what alignment not only has performance implications, but could also affect the proper behavior of your program.

You should review the sections of this manual that discuss alignment, then consider the examples in the remainder of this section:

Example:

```
Structure declaration:
    typedef struct
        {
            char member1;
            int member2;
        } data1_t;
    declspec(cls) data1_t good_data;
```

The natural alignment for the data1_t structure is on a four-byte boundary because the compiler will place the head of a structure aligned on a boundary relative to its storage type. Large objects in MEM (>= sixteen bytes) will be aligned on a sixteen-byte boundary while all others, including anything declared in CLS, local memory or any type of registers will be aligned on an eight-byte boundary.

In addition, the compiler adds padding between elements to maintain individual member natural alignments. In this case, three bytes of padding will be inserted between member1 and member2 so that member2 will fall on a four-byte boundary (since it is an "int").

The resulting good_data variable will be laid out in CLS memory as shown in Table 7.1:

Table 7.1. Example Variable Memory Layout

Bytes	Memory Layout
0	member1
1	
2	padding1
3	
4	
5	member2
6	
7	

There are two possible issues in this situation. First is the insertion of three bytes of padding between member1 and member2; and second is the potential performance penalty for accessing a structure on a boundary that is less than the address granularity of the storage type. Although in this example the structure is aligned well relative to its storage type (CLS), there are examples in the following sections to highlight this concern.

7.1.2.1 Packing Structures

The compiler adds the padding to improve structure access performance. The most obvious issue with the compiler adding padding to a declared structure is that the size of the structure will be increased accordingly. This might not be a concern when the structure is being allocated to a larger storage area like MEM, but if this structure is destined for registers, it could lead to running out of a storage resource prematurely.

Another (less obvious) issue with the compiler adding padding to a structure is that overlaying this structure type on top of non compiler-allocated data will result in incorrect access to members of that structure.

Example:

```
typedef struct
{
        U32 mac_addr[3];
        U16 prot_type;
        U32 src_addr;
        U32 dest_addr;
} hdr_t;
__declspec(mem) long long *p_packet;
__declspec(mem) hdr_t *p_header;

// Assign p_packet a packet buffer address.
// Then receive a packet into the RBUF.

mem_rbuf_read_ind(p_packet, ...);
p_header = (__declspec(mem) hdr_t *) p_packet;
if(p_header->dest_addr == OK_ADDRESS) { ... }
```

Consider that the compiler will add two bytes of padding between prot_type and src_addr to make sure that scr_addr is aligned on a four-byte address. When mapped onto the packet data from the RBUF, there will be some misalignment of structure members.

The following table shows a hdr_t data structure as the compiler expects to see it:

Bytes==>	0...11	12...13	14...15	16...19	20...24
	mac_adr	prot_type	padding	src_addr	dest_addr

Next is an actual header from a packet moved from the RBUF to MEM (with no padding):

Bytes==>	0...11	12...13	14...17	18...21
	mac_adr	prot_type	src_addr	dest_addr

These tables show that accessing src_addr and dest_addr using the hdr_t pointer will return incorrect data. Specifically, p_header→src_addr will return the last two bytes of the actual src_addr and the first two bytes of the actual dest_addr concatenated and p_header→dest_addr will return the last two bytes of the actual dest_addr and the next two bytes of the rest of the packet.

The compiler provides a means for your program to specify that no padding should be inserted between bit fields or between any members of a structure. Specifically, __declspec(packed) and __declspec(packed_bits) provide this functionality. For the preceding example, you should declare hdr_t as shown in the following example:

Example:

```
typedef struct __declspec(packed)
{
        U32 mac_addr[3];
        U16 prot_type;
        U32 src_addr;
        U32 dest_addr;
} hdr_t;
```

With this declaration the compiler will not insert any padding between prot_type and scr_addr, matching the "real," non-compiler maintained data from the wire. Access to all members of the structure will be logically successful.

One of the drawbacks to packing data structures is that accessing members that cross an addressing boundary of the storage type (i.e. 4-byte for registers, CLS; 8-byte for MEM) will incur extra overhead of bit extraction and/or concatenation, and possibly the reading/writing of extra memory locations. However, it is a necessary technique for the kind of situations mentioned in this section.

7.1.2.2 Overriding Natural Alignment

Consider the application of a __declspec(packed) modifier to the structure as declared previously. Now, since padding between member1 and member2 will be removed, access to the second member of this structure will most likely lead to less than optimized code. The compiler will need to account for the fact that member2 will span two 32-bit words. Still, access to the head of the structure (as part of an array, for example) will be aligned to the storage type.

However, there are cases where access to both the structure members and the whole structure itself is not optimized. This is most likely to happen when dealing with packed structures full of small elements, as in the following example:

Example:

```
typedef struct __declspec(packed)
{
    char a_count;
    short s_count;
    char b_count;
} count_t;
__declspec(i32.cls) count_t *pkt_count1;
__declspec(i33.cls) count_t *pkt_count2;
...// Assign values to pkt_count1;
*pkt_count2 = *pkt_count1;
```

The natural alignment of this structure is on a byte boundary, so the compiler will make no assumptions about the position of the structure in memory. Consequently, it will generate code for the copy operation that will take into account the fact that the structure might span two memory words. Structure alignment is not optimized automatically because of the possibility that the structure will be embedded inside an array or another structure, calling for the use of the structure's natural alignment.

However, you can override the default alignment of a basic or aggregated data type using the "aligned(n)" __declspec modifier. If the unmodified structure's natural alignment is less than the addressable granularity of its storage region, the performance of whole structure copies can be improved by increasing the alignment to at least this granularity. That is, if the structure is being allocated to registers, CLS memories, performance would be improved if the structure was aligned on at least a four-byte boundary, and similarly on an eight-byte boundary for MEM. In relation to the original example, a better way to declare the count_t structure would be:

Example:

```
typedef struct __declspec(packed aligned(4))
{
    char a_count;
    short s_count;
    char b_count;
} count_t;
*pkt_count2 = *pkt_count1; // copy performance is improved
```

More information on the syntax of the aligned(n) modifier can be found elsewhere in this manual.

7.1.3 Efficient Structure Access

Structures make for convenient vessels to access related pieces of data. With free use of structures in most Microengine C programs, the compiler absolutely needs to be able to access fields of a structure (including bit fields) efficiently. There are several things you can do when you design and access data structures to help the compiler access them most efficiently.

7.1.3.1 Sizing of Structure Members

Structure members with the following characteristics will produce the most efficient access because of the sizing of registers in the network processors:

- A multiple of four bytes in size

- Falling on a four byte offset from the start of a structure

- Do not cross a 4 byte boundary.

For members lacking in one or more of these conditions, the compiler will need to generate extra instructions or memory accesses to extract or concatenate data from one or more registers every time this data is referenced. The following two examples will demonstrate and discuss the repercussions for variables in this category.

Example:

```
typedef struct __declspec(packed)
{
    char mem1;
    int mem2;
    int mem3;
} mem_test_t;
```

This structure does not follow the three guidelines previously mentioned for any of its members. In fact, the members with the most overhead for access are mem2 and mem3, since both do not start on a four-byte offset and both cross a four-byte boundary (mem2 lies between byte offset 1 and 4, and mem3 lies between byte offset 5and 8). The first member, mem1, is not a multiple of four bytes in size, but is an example of the second best structure construct. Members (including bit fields) of byte multiples in size (8, 16 or 24 bits) that follow the other two edicts only require a single ld_field instruction to operate on such data. And, although this is not shown in this example, note that bit fields between 1 and 8 bits can be extracted with a single instruction with immediate mask. However, an insert will take 2 instructions.

Of course, by default (i.e. without the packed keyword), the compiler would have placed 24 bits of padding between mem1 and mem2 in order to align mem2 and mem3 on their natural four-byte boundaries. This would have yielded optimal access to all members of those sizes in the structure at the cost of restrictions on how your program could use this structure with non-compiler maintained data. In either case, the total size of the structure would still be 12 bytes.

Example:

```
typedef struct
{
    int mem2;
    int mem3
    char mem1;
} mem_test_t;
```

This data structure allows the best possible access for structure members of those sizes, simply by reordering their declared positions. Here, all members are aligned on byte boundaries and are of proper size to warrant optimal

treatment by the compiler, similar to a non-packed version of the first structure—but without the padding between members.

Of course, you can create structures with any member sizing and order and the compiler will do its best to optimize access, but the best possible performance will be obtained with a little forethought. For some, total structure size or unpadded data will be a concern, and so packing structure members will be necessary at the cost of a few extra instructions to pack and unpack data. If at all possible, a software architect should adhere to the three guidelines concerning structure layout presented above.

7.1.3.2 Using Unions

As an alternate (or perhaps a supplement) to the structure member sizing strategies outlined in the previous section, unions can be used to streamline access to several structure members at once. This can be critically important if your structure resides in a high latency memory.

Example:

```
typedef struct {
        union
        {
            struct {
                    U32 a1:16;
                    U32 a2:16;
                    U32 a3:16;
                    U32 a4:16;
            } a_params;
            struct {
                    U32 b1;
                    U32 b2;
            } b_params;
};
} params_t;
volatile __declspec(cls) params_t param_set;
void main()
{
            volatile U32 p1 = 0x1111;
            volatile U32 p2 = 0x2222;
            volatile U32 p3 = 0x3333;
            volatile U32 p4 = 0x4444;
            // Assign each member separately
            param_set.a_params.a1 = p1;
            param_set.a_params.a2 = p2;
            param_set.a_params.a3 = p3;
            param_set.a_params.a4 = p4;
                // Format data for a1 and a2 and write into b1
                param_set.b_params.b1 = (p1<<16) | p2;
                // Format data for a3 and a4 and write into b2
                param_set.b_params.b2 = (p3<<16) | p4;
}
```

The compiler can certainly figure out how to combine the "a_params" code above, but it might generate four separate writes to CLS, one for each of the a_params assignment statements:

Example:

```
/******/              param_set.a_params.a1 = p1;
alu_shf[$0, --, B, a6, <<16]
cls[write, $0, a3, 0, 1], ctx_swap[s2]
/******/              param_set.a_params.a2 = p2;
alu[$0, --, B, b7]
cls[write, $0, a3, 0, 1], ctx_swap[s2]
/******/              param_set.a_params.a3 = p3;
alu_shf[$0, --, B, a7, <<16]
cls[write, $0, a3, 4, 1], ctx_swap[s2]
******/              param_set.a_params.a4 = p4;
alu[$0, --, B, b0]
cls[write, $0, a3, 4, 1], ctx_swap[s2]
```

A less straightforward example would include other code between each assignment, making it more difficult for the compiler to even consider any memory access optimizations.

In any case, a reduction in memory accesses is guaranteed through the use of the union in the params_t structure. Here, the four 16-bit a_params members are declared in a union with two 32- bit b_params members. By using the latter as an alias to write into the former, four CLS writes become two. Of course, you will need to format the data "by hand," explicitly performing the shifting and logical bit operations in code versus letting the compiler generate such code automatically. The tradeoff in code complexity to save accesses to memory is well worth it—just be sure to comment the code appropriately to avoid confusion.

Example:

```
/******/              param_set.b_params.b1 = (p1<<16) | p2;
alu_shf[$0, b7, OR, a6, <<16]
cls[write, $0, a3, 0, 1], ctx_swap[s1]
/******/              param_set.b_params.b2 = (p3<<16) | p4;
alu_shf[$0, b0, OR, a7, <<16]
cls[write, $0, a3, 4, 1], ctx_swap[s1]
```

7.1.4 Miscellaneous Considerations

These tips did not fall under one of the main headings, but are helpful to maximize code performance.

7.1.4.1 Signed vs. Unsigned Integers

The various basic data types, such as integers ("int") are, by default, signed entities. In some cases, a small performance benefit can be derived by using the unsigned versions wherever possible, especially in bitfield structs. Otherwise the compiler may generate unnecessary ASR instructions to handle sign extension when extracting fields.

Example:

```
typedef struct three_fields
    {
        int a1:16;
        int b2: 8;
        int c3: 8;
    } three_fields_t;
    int result;
    three_fields_t my_var;
result = my_var.b2; // implemented with 2 instr, ASR, ALU_SHF
```

Instead of using this code, the preferred method would be to declare three_fields_t with unsigned integer bit fields as such as shown in the next example:

Example:

```
typedef struct three_fields
{
    unsigned int a1:16;
    unsigned int b2: 8;
    unsigned int c3: 8;
} three_fields_t;
```

And so the following access to a field of such a structure will only take 1 instruction, instead of two:

Example:

```
result = my_var.b2; // implemented with 1 instr, LD_FIELD
```

7.2 Tuning Established Code

If you follow the good programming practices presented in this manual, you should be able to program logically sound code for the network processors. This next section is intended to bring you to the next level in terms of coding conventions that will lead to best compiler performance, and hence, best program performance. It will highlight techniques to handle register spillage, to improve access to variables in memory, to optimize function calls, and more.

7.2.1 Register Spillage

Not all register candidate variables will be allocated to actual registers by the compiler, either because there simply are not enough registers to handle all variables live at a certain point in the code, or because the address of a variable was taken. In such a case, the default behavior of the compiler is to automatically "spill" these variables to one of the following resources based upon the -Qspill=n command-line option:

* Next Neighbor registers

* Local Memory

- IMEM Memory

- Cluster Local Scratch Memory

Please see -Qspill command-line option (Table 3.1) for more information.

When register spillage occurs, the compiler will provide information about which variables were spilled, and to which storage type if the -Qperfinfo=1 command-line option is used. In addition, the -Qliveinfo option provides liveness information for all register variables in the program.

7.2.1.1 Handling Register Spillage

One option is to turn off the automatic spillage feature of the compiler altogether, using -Qspill. In this case, if the compiler cannot allocate all variables without explicit storage declarations to registers, the compilation will fail and the programmer will have to perform the rearrangement of data by hand. This is a good option if you want to have absolute control over the storage regions for all variables at all times. However, there are less severe options that can give a variable-by-variable or code-section-by-code-section level of control for the compiler's opportunities for register spillage.

First, when your program absolutely needs an individual variable to be placed in a register at all times in the program (ex. frequently accessed variables), you can explicitly declare the variable with a gp_reg storage type.

Example:

```
unsigned int count1;                  // a register candidate
__declspec(gp_reg) unsigned int count2; // must go to a GPR!
```

If for some reason the compiler cannot allocate the count2 variable to a register, then the compilation will fail. You will then take this as an opportunity to rearrange the code to use less register variables during the live range of the failed variable.

There could be specific sections of code for which you do not want to see any variables spilled. The __no_spill_begin() and __no_spill_end() intrinsic functions provide this functionality. In this way, the no-spilling directive to the compiler is done relative to a section of code, rather than on a per variable basis. For example, a variable with several live ranges could spill in one section of code, but not in any __no_spill regions. Again, if the compiler cannot figure out how not to spill variables in a __no_spill region, the compilation will fail.

> ### Note
>
> The compiler does not spill variables accessed inside a no_spill region for the entirety of the program (i.e. not just inside the no_spill region).

7.2.2 Functions

Due to the lack of a stack in the network processors, the compiler has to incur some overhead for function calls. Also, some features cannot be supported, such as recursive functions and function pointer to some extends. Although

the compiler has many optimizations to affect run time performance as little as possible, several programming techniques will provide the compiler with the greatest opportunity to do so.

7.2.2.1 Inlining Functions

The function calling convention in Microengine C is to pass as many register-compatible arguments as possible, saving the return PC to a register, then performing a hard branch to the function. This can be quite a bit of overhead, especially in relation to functions with few lines of actual code.

The alternative is to have a function call inlined at the place in the code in which it was called. In this case, the compiler does not waste a register unnecessarily for saving the PC, or waste execution time branching to the function and back.

The inlining of functions is controllable through compiler options as well as through the use of directives in the C source code. The __forceinline keyword forces the compiler to inline the function regardless of the size of the function (providing that inlining has not been turned off via the -Obn compiler options or in debug code via the -Od option). The __inline keyword allows the compiler to decide whether or not to inline the function based on cost/benefit analysis performed by the compiler when explicit inlining is enabled (-Ob1).

On the other hand, you can prevent the compiler from inlining a particular function while still enabling the general inlining capabilities of the compiler through the use of the __noinline keyword preceding the function prototype and definition. Although the compiler tries to balance control store versus performance based upon command-line compiler options, the use of this keyword allows you to precisely control inlining on a function by function basis.

In general, you should use the -Ob2 command-line option to allow the compiler to inline shorter functions automatically based upon compile time heuristics.

> ### Note
>
> The compiler will still inline functions defined with the __forceinline keyword even with the –Od option specified. This behavior is so to support backward compatibility for older versions of software, but this is subject to future changes.

7.2.2.2 Optimizing Pointer Arguments

It is sometimes possible to improve the speed of access to function arguments passed in with pointers, as in the following example:

Example:

```
void foo(MyStruct *p_x)
{
        // some code using *p_x and assigning *p_x
}
void main()
{
        myStruct_t x;
        ...
        foo(&x);
```

```
        ...
}
```

In this example, you wish to use the function foo to modify the contents of the structure x, by passing the address of x to foo. Since general-purpose registers cannot be accessed with pointers, the compiler cannot place the structure x into registers. Rather, x will be allocated into other storage regions such as local memory or MEM, slowing down—potentially significantly—access to the data contained in x.

If you can guarantee that the pointer parameter of the function is not accessed through "unknown" means (for example through another pointer whose definition is ambiguous, or from another thread), then you can place the "restrict" qualifier directly before the parameter in the function definition. In doing so, the compiler will automatically perform a "structure copy optimization" which will copy the structure to be passed to a global temporary structure accessible by the function foo. Both the original and temporary structures can be placed into registers, with a significant performance gain over a non-restricted pointer parameter. In this case, the function definition would look like the following example:

Example:

```
void foo(MyStruct * restrict p_x)
{
        // Alias-free code using *p_x and assigning *p_x
}
```

Again, the restrict keyword should only be used if you can guarantee controlled pointer access to the data structure in question. Section 4.8.4.1 provides a list of allowable operations for a restricted pointer and a command-line option (-Qperfinfo=256) to help you determine any violations of these rules. But remember that ultimately you are responsible for the safe application of the restrict keyword.

7.2.3 Miscellaneous Optimizations

These tips did not fall under one of the main headings, but are helpful to maximize code performance.

7.2.3.1 Conditional Statements

Compare to zero (==, !=, <, >) rather than the explicit value when possible. This allows the condition codes to be tested as opposed to the compiler generating a subtract and then testing of the condition codes.

Example:

```
if( queue_entry->current_buf.sop_flag == 1 )
{
    do_something();
}
```

In this case, the compiler will need to generate an extra ALU instruction to perform the subtractcompare. A better implementation is as follows, and uses one less instruction:

Example:

```
if( queue_entry->current_buf.sop_flag != 0 )
```

7.2.3.2 Compiler Defer Slot Filling

In general, the algorithm used in the compiler to fill defer slots is limited to looking in the basic block immediately above or below a branch or context swap.

Example:

```
void cool_function(U32 pass)
{
    if(pass)
    {
        ...
    }
    else
    {
        ...
    }
    gl_foo = gl_a + gl_b;
    gl_bar = gl_x + gl_y;
}
```

Presumably, the if statement will be translated into a branch if equal (BEQ) instruction, which allows for up to three instructions in the branch shadow to be deferred. However, the code in the previous example does not give the compiler any instructions as candidates to place into the branch defer slots. This, of course, supposes that the code in the if and else blocks cannot be moved or are otherwise not candidates to lie in the branch shadow. But if the gl_foo and gl_bar assignments can be moved freely in the code from a logical standpoint, then placing them directly above the if statement will provide the compiler with two more opportunities for optimization.

Example:

```
gl_foo = gl_a + gl_b
gl_bar = gl_x + gl_y;
if(pass)
{
// etc...
```

The following sections contain suggestions on how you can optimize, troubleshoot, and debug your code.

7.3 Optimization Techniques

The C compiler performs automatic optimization at various levels but there are some coding techniques that will make your program perform better.

• Minimize variable allocation to memory.

- Taking the address of a variable or declaring it with a memory region attribute causes allocation to memory.

- Structures/arrays larger than 64 bytes (or 128 bytes in 4-context mode) are allocated to memory.

- Minimize access to memory variables.

- When possible, it is preferable to declare a variable that does not require initialization as a local variable in main() rather than as a global or a static variable. This is because a global/ static declaration implies a default initialization to zero. Such variables when allocated to registers or local memory are initialized at runtime. By making it a local variable the unnecessary initialization is avoided.

- When using the ctx() intrinsic, it is preferable to use it directly in a branch condition. This is because the architecture supports the BR=CTX and BR!=CTX instructions.

Example:

```
if (ctx() == 0) { ... }
 else if (ctx() == 1) { ... }
  ...
```

or the alternative:

```
switch (ctx())
 case 0:
       ...
       break;
 case 1:
       ...
       break;
```

are both preferable to:

```
unsigned int c = ctx();
if (c == 0) { ... }
else if (c == 1 ) { ... }
```

Use unsigned types where signed types are not needed.

- Arithmetic right shift is more expensive than logical right shift.

When writing into bit fields, do not mask unnecessarily.

Example:

```
sram_read_write_ind_t ind;
ind.xadd_reg & 0x1f; /* mask is not needed as xadd is 5 bits */
```

Force inlining of very short (1 or 2 microword) functions with __forceinline directive.

- The UC assembler preprocessor supports the use of loops to evaluate constant expressions at assembly time. The use of such loops is not recommended. The compiler will "fold" simple expressions at compile time (example: "i = 4 + 5" will be folded to "i = 9"), but loops will not be pre-evaluated in this manner.

- Whenever possible, use __declspec(aligned(n)) to inform the compiler about guaranteed run time characteristics of the pointer variable. This information allows the compiler to generate significantly better code. It is your responsibility to specify correct values for alignment boundaries. Incorrect alignment information might force the compiler to generate incorrect code.

- Use the "restrict" qualifier on pointers when there are no other means to access the memory it points to. For details, refer to Section 4.8.4.1.

7.3.1 Critical Path Annotation and Code Layout

On the Netronome network processor architecture, there is a penalty paid for each branch that is taken, i.e. each time the code does not proceed sequentially. This can sometimes be removed by use of branch defer slots, but the compiler is not always able to completely fill the defer slots.

Code layout is an optimization performed by the compiler that arranges the code in an order that reduces the number of taken branches. As an example, look at the following code:

```
if (condition)
{
<statement 1>;
}
else
{
<statement 2>;
}
```

Typically, the compiler would produce code similar to this:

```
    alu [--, --, b, condition]
    bne [lab1#]
    <statement_1>
    br [lab2#]
lab1#:
    <statement 2>
lab2#:
```

Notice that there is a taken branch on each path.

Now let's assume that the compiler knows that the 'else' clause is executed far more frequently than the 'then' clause. The compiler could arrange the code differently as follows:

```
    alu [--, --, b, condition]
    beq [lab1#]
    <statement 2>
lab2#:

    ...
lab1#:
    <statement 1>
    br [lab2#]
```

Now there are no branches when the 'else' clause is executed and 2 branches when the 'then' clause is executed. This is a win on the average since the 'else' clause is much more frequently executed. In fact, this optimization will always win when the 'else' clause is taken 2/3 of the time or greater.

Another case is the switch statement. A switch is fairly expensive to implement because it involves an indexed branch to a branch. In the case of a switch, if one leg of the switch is taken more than 30% of the time, a test for that leg before doing the switch will improve the code.

The compiler cannot do this optimization without direction from the programmer. This is because these transformations would hurt performance if the execution ratios were not what the compiler assumed. For this reason, the compiler provides an intrinsic to mark the "critical path" in the program, that is the path that is executed most frequently.

Use the intrinsic function:

```
__critical_path()
```

to indicate that this point in the program is on the critical path. For code that is on the critical path, you should mark the leg of a two way branch (e.g. if statement) that is taken 2/3 of the time or more, and for a switch statement you should mark the most important leg if it is taken 1/3 of the time or more. You should not mark any two paths that are mutually exclusive as both critical, as this provides no significant information to the compiler

The compiler is capable of inferring, from the __critical_path() directives that you insert, other parts of the program that are on the critical path. For example, if you have a series of nested if's you only need to mark the leg of the innermost one as being on the critical path. If one critical path will overlap another one, the user may want to set priorities on the paths. Please see Section 7.3.1.1 for more details.

Here are some rules you should take into consideration when marking the critical path:

- Put a critical path marker inside the main loop of the program, at the top.

- For an if with an else, mark one side or the other if it is executed 2/3 of the time or more.

- For a switch, mark the most often taken case if it is taken 1/3 of the time or more.

- If you have an if without an else, put a critical path marker in the 'then' clause if the if is taken 2/3 of the time or more

- If you have an if that ends with a return or goto statement, and the if (and hence the return or goto) is not executed 2/3 of the time or more, mark the statement following the statement or block controlled by the if.

- If you have a function that is used both on the critical path and off the critical path, do not put critical path markers in the function.

Basically, you want to walk through the code for your program following the most frequently taken path, and place a marker whenever the code makes a decision and one path is on the critical path and the others are not.

7.3.1.1 Multiple Critical Paths

If several critical paths overlap each other, the branches on the overlapping sections will be laid out in an arbitrary order. For example:

```
if (cond1) {
    __critical_path();
    // block 1: most frequent case
}
else {
    if (cond2) {
    __critical_path();
    // block 2: second most frequent case
}
    else {
    // block 3: infrequent case
    }
}
```

In the above segment of code, the user wants the "if (cond1)" statement to give preference to "block 1". The user also wants the "if (cond2)" statement to give preference to "block 2". If the __critical_path() directive is used as above, the critical path choice at "if (cond2)" will be extended upwards by the compiler and overlap with the critical path choice at "if (cond1)". The compiler will not know how to choose the default branch direction for "if (cond1)". This is by design; if the critical paths were not extended in this fashion the user would have to insert a directive inside every if statement surrounding a given frequently executed block.

When critical paths overlap, the user can tell the compiler which one to give preference to by assigning a priority to each path. The __critical_path() directive takes an optional integer argument, which specifies the priority of that path. For example:

```
if (cond1) {
    __critical_path(20);
    // block 1: most frequent case
}
else {
    if (cond2) {
    __critical_path(1);
    // block 2: second most frequent case
}
    else {
    // block 3: infrequent case
    }
}
```

The numbers can range from 0 to 100. The default is 100 if no argument is specified. The critical path with the higher number is given priority. In the above example, "block 1" will be placed as the default for "if (cond1)" because it has a higher priority (20) than the critical path that flows through "if (cond2)".

7.3.2 User-Guided switch() Statement Optimization

You can supply information that will determine how the compiler will perform certain optimizations. Among these are default case removal and switch block packing.

Given the following code:

```
void main()
{
    __declspec(cls) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
    case 0:
        x = 1;
        break;
    case 1:
        x = 2;
        break;
    case 2:
        x = 3;
        break;
    }
}
```

The compiler will generate code such as the following:

```
    cls[read, $0, a0, 40, 1], ctx_swap[s1]
    alu[--, 2, -, $0]
    ble[l_10#], defer[1]
    alu[a0, --, B, $0]
    jump[a0, l_21#], targets[l_23#,l_22#,l_21#]
l_21#:
    br[l_4#]
l_22#:
    br[l_6#]
l_23#:
    br[l_8#]
l_4#:
    br[l_10#], defer[1]
    immed[a2, 1, <<0]
l_6#:
    br[l_10#], defer[1]
    immed[a2, 2, <<0]
l_8#:
    immed[a2, 3, <<0]
l_10#:
    ....
```

The preceding code example shows two possible optimizations that the compiler can perform:

1. The code to test and handle the case where the switch() value does not match any of the other specified cases is not needed.

2. Instead of having the code `jump[]` to a jump table which then branches to the code to handle each case, the `jump[]` can go directly to the handler code, at an offset based on the value of x. This optimization can be performed because each case is handled by code that is approximately of equal length (two instructions). Therefore the offset for each handler is equal to the value of x times two.

The compiler will perform the above optimizations based on the input you supply.

7.3.2.1 Default Case Removal

The first optimization in this example, removal of the handler code for the unmatched ("default") case, requires that you select and provide the appropriate value of the switch() argument. You direct the compiler to remove the "default" case by creating an empty default case and annotating it with the intrinsic function __impossible_path(). For example:

```
void main()
{
    __declspec(cls) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
    case 0:
       x = 1;
       break;
    case 1:
       x = 2;
       break;
    case 2:
       x = 3;
       break;
    default:
       __impossible_path();// add default case, and annotate with intrinsic.
    }
}
```

7.3.2.2 Switch Block Packing

The second optimization in this example (switch block packing) should only be performed if all case handlers are approximately equal in length, with that length preferably a power of two. If the __switch_pack() intrinsic function is placed in the default case, the compiler will try to predict whether the code will benefit from switch block packing, and will perform the optimization if this is possible. For example:

```
void main()
{
    __declspec(cls) int mem[10] = {0,1,2,3,4,5,6,7,8,9};
    int x = 0;
    switch (mem[0])
    {
    case 0:
       x = 1;
       break;
    case 1:
       x = 2;
       break;
    case 2:
       x = 3;
       break;
    default:
       __switch_pack(swpack_auto);
    }
}
```

The possible arguments for the `__switch_pack()` function are described in the `swpack_t` enum in the `ixp.h` header file:

```
typedef enum {
    swpack_none,      // no pack, jump[] to a branch sequence
    swpack_lmem,      // no pack, but use local memory to hold branch table
    swpack_auto,      // auto pack when appropriate
    swpack_1,         // pack it, using up to 1 extra instruction
                      // for the offset calculation
    swpack_2,         // pack it, using up to 2 extra instructions
                      // for the offset calculation
    swpack_3,         // pack it, using up to 3 extra instructions
                      // for the offset calculation
    swpack_4,         // pack it, using up to 4 extra instructions
                      // for the offset calculation
    swpack_5,         // pack it, using up to 5 extra instructions
                      // for the offset calculation
    swpack_6,         // pack it, using up to 6 extra instructions
                      // for the offset calculation
    swpack_7,         // pack it, using up to 7 extra instructions
                      // for the offset calculation
    swpack_8,         // pack it, using up to 8 extra instructions
                      // for the offset calculation
} swpack_t;
```

Note that this optimization should not be performed if the case handlers vary widely in length, because the smaller handlers will have to be padded so that all handler offsets occur at the same intervals.

7.3.3 Creating Context Swap-Free Regions of Code

The __no_swap_begin() and __no_swap_end() intrinsics can be used to create a section of code where the compiler will not create any instructions that incur a context swap, or move any code into the region that will incur a context swap. This allows the user to write critical sections without incurring the overhead of explicit synchronization. Note that the other microengines on the network processor will still continue to execute in parallel. To create a context swap-free region, simply place the __no_swap_begin() and __no_swap_end() intrinsics at the beginning and the end of the desired section of code, as shown in the following example:

```
__no_swap_begin()
... critical section code ....
__no_swap_end()
```

If the code within the critical section contains a context swap operation, the compiler will generate an error message. This includes any access to data structures stored in memory. Function calls made in the critical section are also checked for this condition. Aside from this checking, the compiler will also guarantee that no other code that incurs context swaps will be moved into this region through compiler optimizations.

7.3.4 Loop Unrolling Control

When the -O2 (compile for maximum code speed) option is enabled, the compiler can perform an optimization called "loop unrolling" as shown in the following example.

Original loop:

```
for (i = 0; i < 10; i++) {
a[i] = i;
}
```

Loop unrolled by 2X:

```
for (i = 0; i < 10; i += 2) { // unroll by 2
   a[i] = i;
   a[i+1] = i+1;
}
```

The loop in the second code segment has been unrolled by 2X (the "unroll factor" is 2). Two iterations of the original loop will execute in one iteration of the unrolled loop. The total number of branches executed in the loop is halved. Also, the two statements in the loop body can be optimized together—computation can be reused and more scheduling and pipelining opportunities have been created. Loop unrolling therefore improves the performance of loops, at a cost in code size.

Loop unrolling is performed only for "for" loops. If loops are nested, only the innermost loop will be unrolled. The compiler automatically determines, using various heuristics, whether a benefit can be had for unrolling a given loop, and what the proper "unroll factor" should be. If you want more precisely controlled unrolling behavior, there are two #pragma directives that you can use, as shown in the following example:

```
#pragma nounroll// Don't unroll this loop
#pragma unroll (<unroll factor>)// Unroll this loop by the given unroll factor
```

These directives are placed directly before the loop to be managed. This loop must be a "for" loop, and must be the innermost loop in a series of nested loops. This is shown in the following examples:

```
#pragma nounroll
for (i = 0; i < 10; i ++)     // Don't unroll this
   ...

#pragma unroll(2)
for (i = 0; i < 10; i ++)     // Unroll this loop by 2X, exactly as in
// the above example
   ...

                    // NOT LEGAL, must be applied to

#pragma unroll(2)              // innermost loop
for (i = 0; i < 10; i++)
   for (j = 0; j < 6; j++)
```

```
   ...
   ...

#pragma unroll(2)              // NOT LEGAL, must be applied to for loop
while (1)
   ...
```

The "unroll factor" parameter is the total number of iterations of the loop body that will be in the final unrolled loop. If this parameter is 0 or 1, no unrolling will be performed.

NOTE: Loop unrolling cannot be performed involving inline assembler statements.

7.4 Things to Remember When Writing Code

- Transfer registers must be read or written in 32-bit blocks. On the NFP architecture, you cannot use chars, shorts, and bit fields in transfer registers. If these types are used, the compiler will emit the error message "read xfer reg buffer <name> is written.", in which *<name>* is replaced by the name of the register.

- Next Neighbor registers must be in "self" mode (-Qnn_mode=1) if you want the compiler to use them for local data storage.

- In certain cases, you might want to execute different code in different execution contexts. The context id is obtained by a ctx() call. It is preferable to use this call directly in every construct that controls execution flow. Note that when the program is compiled with 4 contexts enabled, the enabled contexts are the even ones (0, 2, 4, 6).

- To achieve optimized performance, you should declare integer-sized unions over bit fields and manipulate the integers instead of the individual bit fields. For the same reason, when copying a structure, you should copy the whole structure instead of the individual fields of the structure.

- The compiler's optimization can be more effective if you use the direct structure field access instead of the pointer field access. In some cases, if the "restrict" keyword is used, the pointer field access to a function's arguments can be optimized as well. For more information about pointer arguments optimization, see Section 7.2.2.2.

- You can use structa = structb as a clean way to copy blocks of registers (xfer to gpr, etc).

- The compiler globally resolves GPRs so that functions can return integral or aggregate values in registers to multiple callers.

- When possible, avoid multiple reads/writes of a struct in memory by first copying it to a temporary struct in local memory or registers, operating upon this struct, and finally writing it out to memory as needed.

- When debugging, always display instruction addresses. If there are no addresses displayed, the compiler may have removed your code.

- Use the -Qperfinfo compiler option to analyze register pressure, register spills, and register liveness information. You can also get symbol map and function size data. The data produced by the -Qperfinfo option, especially register spill information, is very important. The register allocator assigns variables to a limited number of hardware registers during program execution. When too many registers are in use simultaneously the compiler detects a "register conflict" and instead of assigning a variable to a register demotes it to local memory. In certain cases the compiler might allocate such a variable in IMEM, which may result in significant performance degradation.

- Be aware that qualifying variables with __declspec(gp_reg) will cause them to be regarded by the compiler as unspillable.

- When performing an asynchronous memory write operation (i.e. a write which waits on "sig_done"), you must call the __free_write_buffer() intrinsic to specify the point after which the operation can be considered to have completed. Without this call, the compiler will by default assume that the transfer registers involved in the write operation can be reused immediately after the operation has been issued. This behavior may cause invalid data to be written out to memory. For example:

```
__declspec(write_reg) int x, y;
unsigned int addr = 0x200;
SIGNAL s1, s2;

x = 10;
__asm cls[write, x, addr, 0, 1], sig_done[s1];
y = 20;
// x is still alive at this point
__asm cls[write, y, addr, 4, 1], sig_done[s2];
// y is still alive at this point
wait_for_all(&s1, &s2);
// the following calls are need to prevent x, y
// being released before I/O finished.
free_write_buffer(&x);// or __implicit_read(&x)
free_write_buffer(&y);// or __implicit_read(&x)
```

- Asynchronous reads may require the use of __implicit_read() or __free_write_buffer(), if not all the data is being used. Consider the following example:

```
SIGNAL sig1,sig2;
SIGNAL_PAIR sigpair;
__declspec(read_reg) int sr1[4];
__declspec(read_reg) int sr2[4];
cls_read(&sr1, 0, 4, sig_done,&sig1);
mem_read64(&sr2, 0, 2, sig_done,&sigpair);
__wait_for_all(&sig1,&sigpair);
sum += sr1[0] + sr2[0]; // only use the first element of each
```

The compiler will see that your program is not using the entirety of the buffers sr1 and sr2. It may attempt to conserve registers by assigning overlapping register ranges to sr1 and sr2. For example, $0 through $3 may be assigned to sr1, and $1 through $4 may be assigned to sr2. This is correct if the two memory reads complete in order, since the sum operation only uses sr1[0] and sr2[0], which are $0 and $1, respectively. However, if the cls_read() and mem_read64() operations complete out of order, this assignment will cause problems. When the mem_read64() operation completes, the data you need will be read into $1. But when the cls_read() operation completes, the contents of $1 will be overwritten by the four-word read operation starting at $0.

You must avoid this situation by informing the compiler that the entirety of both transfer buffers is being used, with the __free_write_buffer() or __implicit_read() intrinsics:

```
SIGNAL sig1,sig2;
SIGNAL_PAIR sigpair;
__declspec(read_reg) int sr1[4];
__declspec(read_reg) int sr2[4];
```

```
cls_read(&sr1, 0, 4, sig_done,&sig1);
mem_read64(&sr2, 0, 2, sig_done,&sigpair);
__wait_for_all(&sig1,&sigpair);
__implicit_read(sr1); // create a use of the whole four-word buffer
__implicit_read(sr2); // create a use of the whole four-word buffer
sum += sr1[0] + sr2[0]; // only use the first element of each
```

If the compiler sees that all the data in both sr1 and sr2 is being used, it will not attempt to overlap the register assignments for those buffers, and will assign different registers (8 total in this example) to each buffer.

- Intrinsics which perform atomic operations (test-and-set, test-and-clear, test-and-add, test-and-sub, hash, put-ring, swap) accept two transfer registers, which are required by the NFP architecture to have the same name (i.e. the read register $0 must be paired with the write register $0). This restriction may create unexpected behavior if the same variable is reused in more than one atomic operation. For example, if __cls_test_and_set() is called with arguments val and mask, and another __cls_test_and_set() call is made with arguments val2 and mask, the variables val and val2 will need to be assigned the same register because of their association with the variable mask. Different, but equally sized, variables should be used for each atomic operation if this register assignment behavior is not desired. The compiler will print an error message if this situation occurs.

- Passing an address of xfer/signal/gpr to a function is generally disallowed unless the callee is an intrinsic function. For __forceinline function compiled under command-line options other than -Od and -Ob0, the compiler does its best to propagate xfer/gpr/signal with the address taken at the call site to pointer deferencing inside callee, as if xfer/gpr/signal were directly used. If your program saves aside the pointer to another variable, however, the compiler is not always able to remove that statement, which causes an xfer/gpr/signal register violation from having its address taken. The general guideline follows:

If a user-defined function f takes the address of a xfer/gpr/signal variable in parameter p, then

- f must be inlined, and

- p can only be safely used in the following two cases:

  1. to de-reference in a form like "*((optional-cast)p + const-offset)";

  2. to compare p against another parameter q, which takes address of xfer/gpr/signal.

Pointer arithmetic on p, if any, can only happen in the above two cases.

Saving aside p in another user-defined variable r may cause the compiler to crash because statements like r=p (or r=&xfer after inlining) may not always be removed (especially combining with pointer arithmetic like r=p+1, or complex control-flow-graph between that and use of r, etc) and violates the rule that xfer/gpr/signal cannot have their addresses taken.

In user intrinsic functions, there should not be more than one return statement.

- Write transfer buffers should be fully initialized, with either an assignment or an __implicit_write() call, before they are used in I/O operations. Otherwise, the compiler will assume that the uninitialized transfer registers are actually initialized elsewhere, and will extend the live range of those registers above the declaration point of the buffer. This leads to inefficient register usage and possibly extra spills or a register allocation failure. Example:

```
__declspec(write_reg) int buf[10];
__implicit_write(&buf); // "initialize" the buffer
count = foo();
```

```
cls_write(&buf, addr, count, ctx_swap, &sig);
// I/O size determined at runtime: assumed to be max size
```

Limited forms of indirect register access are possible if you consider the live range computations that the compiler performs, and if you are careful to tell the compiler which registers will be accessed, by using the __implicit_read() and __implicit_write() intrinsics. For example:

```
__declspec(write_reg) int wbuf[10];
__implicit_write(&wbuf); // Tells compiler that wbuf is being written to
__asm {
... loop which writes "wbuf" using T_INDEX
}
cls_write(&wbuf, addr, 10, ctx_swap, &sig);
```

In the above code, if the __implicit_write() call were not present, the compiler would not know that the inline assembly code writes to the buffer wbuf. The cls_write() call would then appear to be using data defined elsewhere, which may cause incorrect program behavior.

Specifically, the compiler will assume that the values used in the cls_write() call are the previous values of the transfer registers allocated to the "wbuf" array, and may propagate those values into the cls_write() call. This will overwrite the values written by the T_INDEX loop, causing the wrong values to be written to CLS.__implicit_read() is necessary when reading registers with indirect accesses:

```
__declspec(read_reg) int rbuf[10];
cls_read(&rbuf, addr, 10, ctx_swap, &sig);
__asm {... loop which reads "rbuf" using T_INDEX
}
__implicit_read(&rbuf); // tells compiler that "rbuf" is read
```

In the above example, the compiler may remove the cls_read() call if it does not see any code which uses the "rbuf" transfer buffer. The __implicit_read() call informs it that the buffer is in fact "live" and its contents are needed.

Indirect accesses should not be used to read or write registers assigned to other threads.

- If the compiler cannot determine that the transfer size for a memory I/O or hash operation is a known constant (for example, calling cls_read() with a variable "x" for the size argument, where "x" does not have a constant value), the "indirect form" of the underlying I/O instruction will be generated, which allows the size to be determined at runtime. However, the compiler still needs to know how many transfer registers should be reserved for the I/O operation, to prevent values from being accidentally overwritten by other I/O operations. The indirect form of the I/O intrinsics contain a parameter, "max_nn", which allows you to specify this information. When the compiler itself generates the indirect form from a direct I/O call with a non-constant size parameter, the compiler will look at the transfer buffer argument passed in and assume that the entire buffer will be accessed. This assumption may cause problems if your program makes an I/O call with a non-constant size that only writes to part of a buffer, and if your program expects the rest of the buffer to retain its previous value. For example:

```
__declspec(read_reg) a[10];
__cls_read(&a, addr, 10...); // init "a" with 10 words
__cls_read(&a, addr, x....); // read "x" words into "a". Suppose "x" < 5,
```

```
                                      //but the compiler does not know it.
... = a[7]
```

In the above example, the second I/O call has a size parameter that the compiler cannot determine is a constant, for whatever reason. Suppose your program knows that this value is never greater than 5. Then you might expect that a [7] will never be touched by the second I/O operation, and that the value of a [7] will be the one read from the first I/O operation. The compiler, however, cannot perform this analysis, and will assume that all the elements of "a" are written to by the second I/O operation. Therefore, the first __cls_read() call will appear to be redundant and may be removed by the compiler, which will cause a[7] to have an unknown value.

The compiler cannot detect when the above situation is occurring (otherwise it would not be a problem). It is recommended that you compile your code with the -Qperfinfo=128 option, which generates warnings for all the instances that direct I/O operations are auto-converted into indirect form. You should examine each of the reported instances, determine if they are making the hidden assumption about partial buffer access described above, and, if so, manually change the I/O operation to the "true" indirect form (cls_read_ind() in the above example), where the maximum transfer size can be specified as a parameter.

- If inline assembly is used with a comment ";" additional C commands after the ";" are not seen.

Example:

```
__asm {cls [read, rd_xfer, address, 0, 1], ctx_swap[x];}
```

This is interpreted by the compiler as:

```
__asm {cls [read, rd_xfer, address, 0, 1], ctx_swap[x]
```

The trailing "}" is lost because of the comment start symbol ";".

- "In cases where the compiler cannot not determine the live range of a register variable, __implicit_undef() can be inserted at the point where user knows that the value is not initialized or not needed. For example,

```
main()
{
       int x;
       // ...
       for(;;)
       {
if(cond1) {
B1:    x = ...;
       }
       // ...
       if(cond1) {
B2:    ... = x; // only use the value previously defined in B1.
       }
       if(cond2) {
B3:    x = ...;
     }
        // ...
       if(cond2) {
B4:    ... = x; // only use the value defined in B3
       }
```

```
        }
}
```

Without user directives, the compiler cannot not determine an efficient live range of "x" since the assignments to "x" may or may not actually be executed. As a result, the compiler must assume that any of the assignments to "x", including the ones in the previous loop iteration, can reach any of the uses of "x". The live range of "x" is expanded into the entire loop. In other words, a register is reserved for "x" throughout the entire loop. You can insert an __implicit_undef() call into places where the value of "x" is clearly no longer needed. For example:

```
main()
{
        int x;
        // ...
        for(;;)
        {
                __implicit_undef(&x);
                // ...
                if(cond1) {
B1:             x = ...;
                }
                // ...
                if(cond1) {
B2:             ... = x; // only use the value previously defined in B1.
                }
                __implicit_undef(&x);
                // ...
                if(cond2) {
B3:             x = ...;
                }
                // ...
                if(cond2) {
B4:             ... = x; // only use the value defined in B3
                }
        }
}
```

The __implicit_undef() call at the top of the loop will prevent the compiler from assuming that any of the assignments to "x" in the previous loop iteration will reach the uses of "x". The call between "B2" and "B3" will prevent the compiler from assuming that the assignment to "x" in "B1" will reach the use of "x" in "B4". As a result, a register will be reserved for "x" only between "B1" and "B2", and between "B3" and "B4".

In the above case, __implicit_write() would be somewhat less efficient. Since __implicit_write() acts by overwriting the previous value of a variable and storing a new one, the compiler has to reserve a register to hold the "new" value. If the __implicit_undef() calls in the above example were replaced with __implicit_write() calls, a register would be reserved for "x" between the first call and "B1", and between the second call and "B3". Also, if the "if (cond1)" and "if (cond2)" statements had "else" cases, a register would have to be reserved for "x" in those areas as well even though the user might only be interested in the value of "x" in the code where the condition was true.

7.5 Queries and Pitfalls

In this section we consider some further points and potential pitfalls: this time in some depth, and based on actual queries and reports from developers using the compiler.

7.5.1 Read-Write Transfer Registers

The NFP convention, where a single symbol such as "$reg" may actually stand for a pair of physically separate transfer registers -- a "transfer in" (read) register and a "transfer out" (write) register -- may cause occasional problems for even experienced assembly language programmers. It is also something that doesn't necessarily fit the C language programming model very well.

Consider the following trivial Standard C program:

```
#include <stdio.h>

int main(void)
{
    volatile int number;
    int is_answer;

    number = 42;
    is_answer = number == 42;
    printf("%d\n", is_answer);
    return 0;
}
```

We would expect that to display the number 1 (standing for "true"). And it is easy to come up with a Micro-C version that produces equivalent output:

```
#define mailbox0 0x5c

int main(void)
{
    volatile int number;
    int is_answer;

    number = 42;
    is_answer = number == 42;
    __asm local_csr_wr[mailbox0, is_answer]
    for (;;);
}
```

However, suppose we were to qualify the definition of our volatile variable as follows:

```
#define mailbox0 0x5c

int main(void)
{
    volatile __declspec(read_write_reg) int number;
    int is_answer;

    number = 42;
    is_answer = number == 42;
    __asm local_csr_wr[mailbox0, is_answer]
    for (;;);
}
```

Now -- and with no change to the program logic -- our program would produce the output 0 (standing for "false").

The reason is clear if we examine the emitted code, which begins:

```
    immed[$0, 42, <<0]
    alu[--, $0, -, 42]
```

The number 42 is assigned to the WRITE (transfer out) register, but it is the READ (transfer in) register that is compared to 42: same symbol but different registers.

If that behavior seems a bit counter-intuitive -- or even to go against C semantics -- the good news is that it is often possible to avoid qualifying variables with __declspec(read_write_reg), at least in situations where doing so is likely to confuse things.

Consider, the following microcode fragment, where a value is read in from one memory location and then written out to another:

```
    .reg $0
    .reg r1, r2
    .sig s1

    immed[r1, 0]
    immed[r2, 1]

    mem[read8, $0, r1, <<8, 0, 1], ctx_swap[s1]
    alu[$0, --, B, $0]
    mem[write8, $0, r2, <<8, 0, 1], ctx_swap[s1]
```

To do the same thing in Micro-C, we should avoid a single variable qualified with __declspec(read_write_reg), as the microcode might suggest. Instead, a more idiomatic C version has the desired effect:

```
int main(void)
```

```
{
    __declspec(read_reg) unsigned int xr;
    __declspec(write_reg) unsigned int xw;
    unsigned int r1 = 0, r2 = 1;
    __declspec(signal) int s1;

    __asm mem[read8, xr, r1, <<8, 0, 1], ctx_swap[s1]
    xw = xr;
    __asm mem[write8, xw, r2, <<8, 0, 1], ctx_swap[s1]
    for (;;);
}
```

And the emitted code is:

```
.%init_reg A1 0x1
.%init_reg B1 0x0
_main#:
    mem[read8, $0, b1, <<8, 0, 1], ctx_swap[s1]
    alu[$0, --, B, $0]
    mem[write8, $0, a1, <<8, 0, 1], ctx_swap[s1]
l_8#:
    br[l_8#]
```

7.5.2 Transfer Registers and Subword Access

Write transfer registers cannot be read from -- it is always the contents of the read transfer register that is read -- and therefore assignments to write registers should be restricted to 32-bit words. Assignments to structure members and to bitfields typically require subword access, and the compiler must generate code that will (effectively) read the current value of a variable, AND off some old bits, OR in some new bits, and then write the value back. But code like this just doesn't work on transfer registers:

```
alu[a0, --, B, $0]
ld_field[a0, 0011, b0, <<0]
alu[$0, --, B, a0]
```

Assuming we have the structure:

```
typedef union {
    struct {
        unsigned h: 16;
        unsigned l: 16;
    } s;
    unsigned x;
```

```
} mybits;
```

then the following fragment

```
__declspec(write_reg) mybits xf;

xf.s.h = 0x7654;
xf.s.l = 0x3210;
__asm mem[write32, xf, &v, 0, 1], ctx_swap[s1]
xf.s.l = 0x0123;
__asm mem[write32, xf, &v, 4, 1], ctx_swap[s1]
```

will cause the compiler diagnostic

```
error: write xfer reg buffer xf is read
```

In the case where both read and write registers are involved

```
__declspec(read_write_reg) mybits xf;

xf.s.h = 0x7654;
xf.s.l = 0x3210;
__asm mem[write32, xf, &v, 0, 1], ctx_swap[s1]
xf.s.l = 0x0123;
__asm mem[write32, xf, &v, 4, 1], ctx_swap[s1]
```

the compiler presently issues the diagnostic

```
warning: subword assignments to transfer registers are not supported
```

though the intention is shortly to make this an error.

The following fragment, although it seems to take the long way around, by maintaining a separate variable for the purpose of manipulating the structure, both avoids problems and causes the compiler to emit better code:

```
__declspec(read_write_reg) unsigned xf;
mybits bf;

bf.s.h = 0x7654;
bf.s.l = 0x3210;
xf = bf.x;
```

```
__asm mem[write32, xf, &v, 0, 1], ctx_swap[s1]
bf.s.l = 0x0123;
xf = bf.x;
__asm mem[write32, xf, &v, 4, 1], ctx_swap[s1]
```

7.5.3 Endian Issues for 64-bit Scalars

The compiler, operating in big endian mode, is big endian not only in its (8-bit) bytes but also in its (32-bit) words. This means that when a "native" 64-bit scalar is spread across two 32-bit registers, the most significant bit will be in register 0 and the least significant bit will be in register 1.

In other words, in default big endian mode, "bw" will be set to 1, and "lw" will be set to 0, in the following Micro-C program:

```
#define mailbox0 0x5c
#define mailbox1 0x5d

int main(void)
{
    unsigned long long n = 0xfedcba9876543210;
    int bw, lw;

    bw = ((unsigned *)&n)[0] == 0xfedcba98 &&
        ((unsigned *)&n)[1] == 0x76543210;
    lw = ((unsigned *)&n)[0] == 0x76543210 &&
        ((unsigned *)&n)[1] == 0xfedcba98;
    __asm local_csr_wr[mailbox0, bw]
    __asm local_csr_wr[mailbox1, lw]

    for (;;);
}
```

The NFP hardware, however, assumes LWBE ("little endian word, big endian byte") order when operating on 64-bit quantities in external memory. This means that it will be necessary to do (32-bit) word swaps, when working with 64-bit scalars. An example follows:

```
#include <nfp.h>

__inline void put64(__declspec(xfer_write_reg) unsigned *p, U64 x)
{
    p[0] = (unsigned)x;
    p[1] = (unsigned)(x >> 32);
}

__inline U64 get64(__declspec(xfer_read_reg) unsigned *p)
{
    return (U64)p[1] << 32 | p[0];
```

```
}

int main(void)
{
    __declspec(write_reg) unsigned xw[2], xa[2];
    __declspec(read_reg) unsigned xr[2];
    __declspec(cls addr40) void *addr = 0;
    SIGNAL s1;
    U64 tmp;
    int ok;

    if (ctx()) return 0;

    put64(xw, 0x1ffffffff);
    cls_write(xw, addr, 2, ctx_swap, &s1);

    put64(xa, 0x100000001);
    cls_add64(xa, addr, 2, ctx_swap, &s1);

    cls_read(xr, addr, 2, ctx_swap, &s1);
    tmp = get64(xr);

    ok = tmp == 0x300000000;
    __asm local_csr_wr[0x5c, ok]
    return 0;
}
```

7.5.4 Ordering and __declspec directives

The order in which attributes may appear in a __declspec directive, and the order in which multiple __declspec directives may appear together, can be constrained by checking carried out by the compiler. For example

```
__declspec(imem aligned(4096)) int some_var;
```

is acceptable, but

```
__declspec(aligned(4096) imem) int some_var;
```

is not. The problem with the second directive is that alignment for generic variables may not exceed 2048, and the variable has not been assigned to a memory region at the point in the __declspec directive where the "aligned" modifier appears.

7.5.5 Indexing Registers

As documented in section 4 of this guide, the address cannot be taken of a register variable, and this means that it is mostly not possible to index variables that have been assigned to registers. If this is attempted, there may be either an explicit error, or the variable may end up being demoted from the specified register class to internal memory.

As an example of the latter behavior, the following source compiles

```
typedef struct var_t {
    unsigned val[3];
} var_t;

void fn(__declspec(gp_reg) var_t foo)
{
    unsigned i, x;

    for (i = 0; i < 2; i++) {
        x = foo.val[i];
        __asm local_csr_wr[0x5c, x]
    }
}

int main(void)
{
    __declspec(gp_reg) var_t foo;

    foo.val[0] = 0x54321;
    foo.val[1] = 0xa9876;
    foo.val[2] = 0xfedcb;
    fn(foo);
    for (;;);
}
```

but a look at the emitted code reveals that the variable has been assigned to internal memory. The -Qperfinfo=1 compile option will cause a warning to be issued, if desired.

7.5.6 Waiting for Signals

The compiler implements the intrinsic functions `__wait_for_all` and `__wait_for_any` which provide for the swapping-out of the currently-running context until activation of a specified signal or signals. These functions basically compile to NFP `ctx_arb` instructions. And, in the case of signal pairs, both even and odd signals are included in the `ctx_arb` mask.

As an example, the fragment

```
SIGNAL sig;
SIGNAL_PAIR sigpair;

wait_for_all(&sig);
wait_for_all(&sigpair);
```

produces the code

```
ctx_arb[s1], all
ctx_arb[s3, s2], all
```

In passing arguments to these intrinsic functions, it is possible to specify only the `odd` or `even` part of a signal pair. However, for technical reasons to do with type information available in the context of a variadic function, the compiler cannot distinguish between the arguments `&sigpair` and `&sigpair.even`. So the fragment

```
wait_for_all(&sigpair.odd);
wait_for_all(&sigpair.even);
wait_for_all(&sigpair);
```

results in the code

```
ctx_arb[s3], all
ctx_arb[s3, s2], all
ctx_arb[s3, s2], all
```

To make it possible to wait for just the `even` signal of a signal pair, two variant functions are available: `__wait_for_all_single` and `__wait_for_any_single`. So, for example, the fragment

```
wait_for_all_single(&sigpair.odd);
wait_for_all_single(&sigpair.even);
wait_for_all_single(&sigpair);
```

will produce the code

```
ctx_arb[s3], all
ctx_arb[s2], all
ctx_arb[s2], all
```

7.5.7 An Approach to Inline Assembly Language

While it will always be necessary to use inline assembly language statements in most non-trivial Micro-C applications, the suggested approach is nevertheless to use C where possible. Therefore, a fragment such as

```
__asm {
    alu[--, --, b, fl]
    beq[L1]
    alu[xw, --, b, data]
    mem[journal, xw, hi, <<8, 0, 1], sig_done[s1]
L1:
}
```

is best rewritten as

```
if (fl) {
    xw = data;
    __asm mem[journal, xw, hi, <<8, 0, 1], sig_done[s1]
}
```

In particular, assignments are best done in C. Use can then be made of intrinsic functions, and complex structure and array accesses -- which may not be properly supported by the inline assembler -- can be avoided.

7.5.8 Alignment in Thread Local Storage

Thread local storage (TLS) is used by the compiler for local variables: those, for instance, declared within a function and which have no __declspec that overrides their scope. The compiler has always had a policy of not strictly honoring specified alignment, when it comes to TLS variables. (Since many TLS variables end up in local memory, for instance, strictly honoring specified alignment could result in much of this scare resource being wasted.) Developers should take this compiler behavior into account, and not make use of variables that will end up in TLS where specified alignment needs to be strictly honored. (Variables declared, for instance, with __declspec(shared) or __declspec(export) will not end up in TLS and are not subject to the behavior described.)

7.5.9 Modifying Intrinsic Parameters

When writing __intrinsic functions, try to keep in mind what difference argument propagation is going to make, and think in terms of different possible arguments.

For example, the following function is intended to read 32 bits from MEM, given the high and low portions of an address:

```
__intrinsic unsigned
rd(unsigned hi, unsigned lo)
{
    unsigned xr;
    SIGNAL s1;

    hi <<= 24;
    __asm mem[read32, xr, hi, <<8, lo, 1], ctx_swap[s1];
    return xr;
}
```

The trouble with the function is that it modifies one of its parameters, which (strictly speaking) user __intrinsic functions should not do. On the other hand, because of the way the compiler's optimizers work, it is quite often possible to get away with doing things in Micro-C that are not fully legal, because the illegal parts will have been optimized away before the crunch actually comes. Of course, these functions may then work for most possible arguments but not for all.

When invoked as follows:

```
    hi = (unsigned long long)&foo >> 32;
    lo = (unsigned)&foo;
    x = rd(hi, lo);
```

the example function will compile correctly. But when invoked like this:

```
    x = rd(0, 0);
```

compilation will stop with the error "intrinsic parameter cannot be modified".

If you think in terms of

```
    hi <<= 24;
```

being replaced with

```
    0 <<= 24;
```

it is easy to see why.

7.6 Thinking About Intrinsics

The basic idea of an intrinsic routine, from a compiler perspective, is a routine that can't be implemented in the programming language itself. To give a rather dated example, the Pascal write procedure can't be written directly in Pascal. So the Pascal write procedure has to be built into the compiler. On the other hand, in classic C, pretty much all the library functions can be written in C.

Micro-C has a great many intrinsic functions. Clearly something like `__associate_read_write_reg_pair_no_spill` needs to be built into the compiler because there's no way anyone could sit down and hack out a C89 implementation.

The nfcc compiler also makes provision for user intrinsics. On the face of it, this seems nonsense: how can the user implement a function which, by definition, cannot be implemented using the programming language? Well, nfcc gets round this by providing not only C but also a limited "super language" where normal C semantics don't apply. And the way to go from C to this "super language" is to tag the function `__intrinsic`.

The basic difference between the standard C89 that nfcc provides and this "super language" lies in the way function arguments are handled.

In normal C, the code generated for a function is completely determined by the function definition. If I plan to write a function `myfn` and I sometimes want to use myfn on integers, and sometimes on short integers, and sometimes on long integers, then -- by the rules of C -- I have basically two choices: use a variadic function

```
int myfn(int type, ...);
```

where I pass type information using a preliminary argument, so

```
myfn(my_short, x);
myfn(my_int, y);
myfn(my_long, z);
```

or use a void pointer

```
int myfn(int type, void *p);
```

where, again, I have to pass separate type information using an argument:

```
myfn(my_short, &x);
myfn(my_int, &y);
```

```
myfn(my_long, &z);
```

The important point here is that I, as the programmer, have to know what types x, and y, and z are, and I have to decide on a convention to represent data types, and pass this information around "by hand". Standard C does not even provide facilities to do this

```
myfn(type(x), &x);
```

In other words, it does not expose meta data to the user.

Standard C semantics are based on the idea that each function is compiled only once into object code, and therefore must cater for all possible valid arguments. Now nfcc does support inlining of functions, and this does introduce the possibility of the compiler adapting the generated code to a given argument. So, for example, faced with the source code

```
__inline int twice(n) { return n + n; }
```

and

```
x = twice(2);
```

there is no reason why a compiler should not replace the function invocation with the obvious literal value 4.

However, in processing inline functions, the compiler is never allowed to generate code based on its own private knowledge of argument types and other attributes, if this private knowledge would mean it violates standard C semantics. Also, as already noted, standard C provides no user access to compiler meta data, like argument types and other attributes.

At this point we might well ask, "Well, why not just expose the compiler meta data by building the required intrinsic functions into the compiler?" And, of course, this is what nfcc has done with supplied functions like __is_ct_const, __is_in_cls and so on.

However, because of the rigid and very straightforward standard C semantics, this idea does not really get us as far forward as we'd like. To see why, let's consider the example of __is_ct_const, which returns true if its argument is a compile-time constant.

Suppose I write

```
int myfn2(int n)
{
    if (__is_ct_const(n))
```

```
        return PLAN_B;
    return PLAN_A;
}
```

using some nfcc-like compiler. At once, I run into problems with C89 semantics. The function `myfn2` will (in effect) be compiled into several lines of assembler code and these will then sit there, waiting to service all callers. The argument `n` is simply a variable and must be a variable and nothing else, if the function is to work for more than one possible argument. So, in any non-inlined context, `__is_ct_const` is always false and essentially meaningless.

Now, nfcc supports some features of C99, and one of these is inlining. If we change the function definition to

```
__inline void myfn2(int n)
{
    if (__is_ct_const(n))
        return PLAN_B;
    return PLAN_A;
}
```

then the compiler has more scope, and can consider arguments on a case-by-case basis. It will also take the function invocation into account. So

```
x = myfn2(42);
```

is clearly a matter for `PLAN_B`, whereas

```
z = myfn2(y);
```

is going to involve `PLAN_A`.

As shown, inlined functions will work correctly for trivial cases, but not for deep ones. Consider this program:

```
__inline int fn(int x) { return 1 + fn2(x); }
__inline int fn2(int x) { return 2 + x; }

int main(void)
{
    const int answer = 42;
    int x = answer;
    int y = fn(x);
    int z = fn(y);
    return z;
}
```

with specific reference to whether the arguments to functions are compile-time constants. It is easy to see that `x` is a constant, but determining whether `y` is a constant means examining `fn` and everything it calls. And the problem is worse with `z`.

We also have to bear in mind that function fn may not consist of one line but possibly thousands of lines, and it may itself call functions consisting of thousands more lines of code.

The nfcc compiler design is such that it needs to make decisions about some things at once, on entry to the code generator. And the way it makes the whole "extensively analyze call tree before doing anything" problem go away, and the way it also extricates itself from the straitjacket of C89 semantics, is to provide a category of function that goes two steps beyond C89. Step one is to inline these functions, and that we have already considered. Step two is to do wholesale argument propagation on these instantiated functions, from call tree root to call tree leaves, disregarding standard C semantics. And that is basically what distinguishes user intrinsics.

So, an `int n` argument might in effect be replaced by a literal 42, for a particular instantiation of an `__intrinsic` function (which doesn't violate C semantics). And a `void *p` argument might in effect be replaced by `volatile __declspec(mem, addr40) int *p` (which does violate C semantics).

This argument propagation, which is the defining characteristic of user intrinsics, makes the use of true intrinsics like `__is_ct_const` and `__is_in_cls` possible.

Just to clarify: a standard C function like

```
int whatis(void *p);
```

tells the compiler, "Behave as though you know nothing whatever about what `p` points to," and it is this aspect of standard C the argument propagation works around.

All of this also explains why user intrinsics may call only intrinsics (for instance, once you've propagated arguments, you can't "unpropagate" them and turn them into normal C arguments), and why true intrinsics like `__is_in_cls` need to be used inside user intrinsics (they rely on the argument propagation to work).

This picture is simplified and slightly idealized, but may provide a useful way of looking at these things.

7.7 Troubleshooting

7.7.1 Program Does Not Fit

- Compile for size (-O1).
- Use __noinline on any functions that are inlined and called from multiple places to prevent the compiler from inlining them.
- Reduce inlining (-Ob1, or -Ob0).

7.7.2 Program Does Not Run Correctly

• Compile at warning level 4 (W4) and check warning messages.

• Generate source level debugging information (-Zi).

> **Note**
>
> See Table 3.1 for more information on compiler command-line interface (CLI) options.

7.8 Debugging Techniques

7.8.1 Compile-Time Information

The Microengine C compiler can deposit debugging information in the `.list` file output to be passed to the linker, including source file to assembly code mapping. In addition, the compiler will produce a unique .dbg file for each source compilation. The compiler then reads back the .dbg during the whole-program compilation to generate a `.list` file (with debugging information at the end of the `.list` file). All source-level debugging information available to the Programmer Studio is in the `.list` file.

Specifically, the .dbg file contains variable scope information and datatype definition (for example, structure field layout) for use in the Data Watch window. In addition, an optional command-line option prints out various information to help you make optimal decisions on topics ranging from register spillage to "restrict" pointer violations.

7.8.1.1 Performance Information

The -Qperfinfo=n command-line option can provide valuable information about potential performance problems in an application. The –Qperfinfo values are described in Section 3.1.3. Note that each value above is actually a bit mask for the perfinfo option. That is, you can request multiple informational items from the above list for a given compilation by OR-ing several nvalues together. For example, if you would like to view one of the following: register candidates spilled; local memory allocation; and warnings for restrict pointer violations during a compilation, add -Qperfinfo=273 (alternatively: -Qperfinfo=0x111) to the compiler command line.

For all compilations you should include n=1 if register spillage is enabled (via the spill switch), as spilled variables will have various performance implications. Similarly, other options should be included if the code warrants it (i.e. use n=256 if there are restricted pointer parameters in your program).

These first three perfinfo options, along with the last one, were provided to help you manage register allocation in the program. However, -Qperfinfo=2 and -Qperfinfo=4 have been superseded by the command-line option -Qliveinfo=[gr,sr,sw,srw,dr,dw,nn,sig,all]. -Qliveinfo causes the compiler to print out liveness information for all register allocated variables in a more helpful and user-friendly manner. In fact, it uses a different algorithm, one

that more accurately reflects real register allocation than -Qperfinfo=2 or -Qperfinfo=4 did. You can display the register allocation information for only the register types of interest, by providing one or more of the -Qliveinfo options shown in Table 7.2:

Table 7.2. -Qliveinfo Options

Option	Result
gr	General purpose registers
xr	SRAM/xfer read registers
xw	SRAM/xfer write registers
xrw	SRAM/xfer read/write registers
dr	DRAM read registers
dw	DRAM write registers
drw	DRAM read/write registers
nn	NN registers (self mode only)
sig	Signals
all	All registers

In short, the -Qliveinfo option can help you analyze your program and determine which code segments have a high register pressure and need to be restructured

The first section of compiler output from the -Qliveinfo=gr command-line option details, on a function by function basis, the registers that are live when the function is called (Live in), those live upon completion of the function (Live out), and those live both in and out of the function (Live through).

Example:

```
: Live info. of gpr registers for Function meter_calculate_ebs_cbs:
   : Live in(11):
:        gr.554(_timestamp) gr.555(_timestamp+4)
   gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
   gr.645(entry) gr.685(result) gr.687(p_sram) gr.740(cgt.1090)
   gr.743(cgt.1093) gr.897(..) gr.899(..)
:    Live out(9):
:       gr.554(_timestamp) gr.555(_timestamp+4)
   gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
   gr.685(result) gr.687(p_sram)gr.740(cgt.1090) gr.743(cgt.1093)
   gr.899(..)
:    Live through(7):
:       gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093)
gr.899(..)
```

The registers are printed out in the following format:

```
cls.ID(variable_name)
```

where cls is one of the register classes mentioned above, the ID is a compiler-maintained virtual register number, and variable_name is the name of the corresponding variable. A variable_name of ".." implies a compiler-generated temporary variable is being used.

You can use this information to help determine which functions have maxed out, or are close to maxing out, register usage. For example, there are 32 available GPRs per thread, so seeing values close to 32 in the Live parentheses above advises you to pay special attention to those functions if there is a problem with register spillage.

Following the first section of the example is register liveness on a per-instruction basis. For each line of code in the program (Microengine C source with corresponding assembly), the "Live set" of registers is listed. This information lets you further refine the search for high register pressure areas of code.

Example:

```
: /******/ meter_params[entry].timestamp = timestamp;
:       alu[gr.955(..) , 4, +, gr.802(..) ]
: Live set(15): gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr) gr.651(tmp.27)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093)
gr.802(..) gr.803(..) gr.804(..) gr.897(..) gr.899(..) gr.955(..)
:
:       alu[??, --, B, gr.554(_timestamp) ]
: Live set(14): gr.554(_timestamp) gr.555(_timestamp+4)
gr.556(_meter_me_signal_csr) gr.557(_cache_signal_csr) gr.651(tmp.27)
gr.685(result) gr.687(p_sram) gr.740(cgt.1090) gr.743(cgt.1093) gr.803(..)
gr.804(..) gr.897(..) gr.899(..) gr.955(..)
: // etc...
```

Additionally, the -Qperfinfo=2048 option will provide a quick summary of the lines of code with the maximum physical register pressure. An example dump is shown in the following example:

Example:

```
Maximum live physical gprs (52) at myfile.c 245
Maximum live physical gprs (52) at myfile.c 203
Maximum live physical gprs (52) at myfile.c 196
Maximum live physical gprs (52) at myfile.c 192
```

The maximum number of live physical GPRs before spilling is printed along with the file name and line number of the high register pressure points.

7.8.2 Run-Time Debug Tools

When Microengine C-based programs are compiled to one or more MEs with debug information turned on, the Programmer Studio provides several debugging tools specifically for the compiler, allowing you to debug on a source code or assembly code level. The *Netronome Network Flow Processor 6000: Development Tools User's Guide* describes the use of all such tools, including source/assembly view toggling, data watches and breakpoints.

7.8.3 Debugging Inline Functions

When a function is inlined, the line number associated with the inlined code is the same as the call site. You cannot step into a inlined function.

7.9 Summary

You must be well versed in both the operation of the network processor hardware and Microengine C compiler options to write effective and performance minded code. Experience (i.e. writing code!) is the only way to become a great programmer, but this chapter is aimed at shortening the curve by providing explanations and examples for many of the common pitfalls and performance considerations for a new Microengine C user.

In addition, the explanations (in plain language) of the wealth of information provided by the compiler will help you make optimal decisions for data placement and code structure. A discussion of the compiler-specific debugging features and tools in the Programmer Studio will help you to debug code quickly from both a logic and performance point of view.

The tools and documentation are provided to make the process of writing and debugging code as easy as possible, but ultimately, it takes a knowledgeable and well-disciplined programmer to write effective Microengine C code.

8. Compiler Intrinsics

The compiler intrinsic functions are C-like functions built into the C compiler itself. Function declarations are in the `nfp.h` header file. The SDK standard library is a good place to look for examples of the use of these intrinsics. A number of them are also discussed in more depth elsewhere in this guide.

8.1 Intrinsic Data Types

```
typedef enum
{
    swpack_none,  // No pack, jump to a sequence of branch
    swpack_lmem,  // No pack, use local memory for branch table
    swpack_auto,  // Auto pack when appropriate
    swpack_1,     // Pack using up to 1 extra instruction
    swpack_2,     // Pack using up to 2 extra instructions
    swpack_3,     // Pack using up to 3 extra instructions
    swpack_4,     // Pack using up to 4 extra instructions
    swpack_5,     // Pack using up to 5 extra instructions
    swpack_6,     // Pack using up to 6 extra instructions
    swpack_7,     // Pack using up to 7 extra instructions
    swpack_8      // Pack using up to 8 extra instructions
} swpack_t;
```

8.2 Intrinsic Functions

Strict prototypes have been eschewed below as many intrinsics accept a wide range of argument types. To understand the pseudocode presented, it should be noted that where "VAR" appears in the parameter list, the address of the relevant variable is meant. (The same applies in the case of "XR" and "XW" parameters.)

8.2.1 __assign_relative_register(VAR, NUM)

Parameters

VAR   Variable of class nn local register, xfer register, or signal

NUM First physical register or signal number to allocate

Description

This function allocates physical registers or signals, as appropriate, to the variable for its lifetime, beginning from 'num'.

8.2.2 __associate_read_write_reg_pair_no_spill(XR, XW[, SIZE])

Parameters

XR   Variable of class read xfer register

XW   Variable of class write xfer register

SIZE Optional size parameter

Description

This function associates read xfer buffer and write xfer buffer variables, optionally restricting its operation to 'size'.

8.2.3 __critical_path([PRIORITY])

Parameters

PRIORITY Optional priority parameter in the range 0 to 100 (defaults to 100)

Description

This function marks a section of code as being on the critical path of the application. In the case of overlapping critical paths, those marked as having higher priority receive higher priority for code layout.

8.2.4 __ct_Qperfinfo(MASK, MSG)

Parameters

MASK This value will be AND-ed with the command line -Qperfinfo=n setting.

MSG   This string will be displayed as a warning.

Description

This function is for use in conjunction with -Qperfinfo for the generation of warnings, for example when the compiler cannot determine the size of a memory I/O transfer.

8.2.5 __ct_assert(EXPR, MSG)

Parameters

EXPR An integer expression of which the value is known at compile-time.

MSG   A string to be displayed as an error.

Description

This function can be used for compile-time checking. If the 'expr' argument evaluates to zero (false), an error is generated, and the 'msg' argument is output preceded by the string "__ct_assert fails".

8.2.6 __ctx()

Parameters

Description

This function returns the currently executing context as an integer (possible range 0 to 7).

8.2.7 __elem_size(VAR)

Parameters

VAR A variable about which information is sought

Description

This function can be used for compile-time checking and returns the element size of the relevant variable.

8.2.8 __free_write_buffer(XW)

Parameters

XW Variable of class write xfer register

Description

This function is used to mark the end of the lifetime of a xfer register write buffer, where necessary.

8.2.9 __global_label(NAME)

Parameters

NAME A unique label name as a string literal

Description

This function creates a named global label at the point the intrinsic is used.

8.2.10 __ilexpr(EXPR)

Parameters

EXPR An import variable name or expression as a string literal

Description

This function provides access to import variables and expressions and is a special-purpose version of the more general intrinsic `__LoadTimeConstant`. It is intended for use in cases where the associated numeric value is known to be in the range 0 to 255, and allows better optimization.

8.2.11 __implicit_read(VAR[, SIZE])

Parameters

VAR A signal or transfer register variable
SIZE Optional size parameter

Description

This function is used to indicate that the relevant variable is being read asynchronously or implicitly by the hardware, and allows correct lifetime analysis to be done by the compiler.

8.2.12 __implicit_undef(VAR[, SIZE])

Parameters

VAR A signal or transfer register variable
SIZE Optional size parameter

Description

This function is used to indicate that the variable can be considered undefined until the next time it is written to, even if the variable is read. This allows correct lifetime analysis to be done by the compiler.

8.2.13 __implicit_write(VAR[, SIZE])

Parameters

VAR A signal or transfer register variable

SIZE Optional size parameter

Description

This function is used to indicate that the relevant variable is being written asynchronously or implicitly by the hardware, and allows correct lifetime analysis to be done by the compiler.

8.2.14 __impossible_path()

Parameters

Description

This function, which should be placed only in the "default" handler of a switch statement, asks the compiler to perform default case removal optimization.

8.2.15 __intrinsic_begin()

Parameters

Description

For internal use only: deprecated.

8.2.16 __intrinsic_end()

Parameters

Description

For internal use only: deprecated.

8.2.17 __is_ct_const(EXPR)

Parameters

EXPR An integer expression

Description

This function checks whether the 'expr' argument is a compile-time constant, returning 1 (true) or 0 (false).

8.2.18 __is_in_cls(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in CLS, returning 1 (true) or 0 (false).

8.2.19 __is_in_ctm(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in CTM, returning 1 (true) or 0 (false).

8.2.20 __is_in_dram(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in DRAM (CTM, EMEM, or IMEM), returning 1 (true) or 0 (false).

8.2.21 __is_in_emem(VAR)

Parameters

 VAR A variable

Description

This function checks whether the 'var' argument is located in EMEM, returning 1 (true) or 0 (false).

8.2.22 __is_in_emem_cache(VAR)

Parameters

 VAR A variable

Description

This function checks whether the 'var' argument is located in EMEM_CACHE, returning 1 (true) or 0 (false).

8.2.23 __is_in_imem(VAR)

Parameters

 VAR A variable

Description

This function checks whether the 'var' argument is located in IMEM, returning 1 (true) or 0 (false).

8.2.24 __is_in_lmem(VAR)

Parameters

 VAR A variable

Description

This function checks whether the 'var' argument is located in local memory, returning 1 (true) or 0 (false).

8.2.25 __is_in_reg(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in registers, returning 1 (true) or 0 (false).

8.2.26 __is_in_reg_or_lmem(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in registers or local memory, returning 1 (true) or 0 (false).

8.2.27 __is_in_sram(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in SRAM, returning 1 (true) or 0 (false).

8.2.28 __is_nfp_arch(CHIP, STEP)

Parameters

CHIP A chip, specified using an `nfp.h` constant, for example `__nfp6xxx`.
STEP A stepping, specified using an `nfp.h` constant, for example `__REVISION_B0`.

Description

This function checks whether the compiler is set up to target the chip and stepping specified in the arguments, returning 1 (true) or 0 (false).

8.2.29 __is_nfp_arch_or_above(CHIP, STEP)

Parameters

CHIP A chip, specified using a `nfp.h` constant, for example `__nfp6xxx`.

STEP A stepping, specified using a `nfp.h` constant, for example `__REVISION_B0`.

Description

This function checks whether the compiler is set up to target the chip and stepping specified in the arguments, or alternatively some later chip and stepping combination, returning 1 (true) or 0 (false).

8.2.30 __is_read_reg(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in read xfer registers, returning 1 (true) or 0 (false).

8.2.31 __is_write_reg(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in write xfer registers, returning 1 (true) or 0 (false).

8.2.32 __is_xfer_reg(VAR)

Parameters

VAR A variable

Description

This function checks whether the 'var' argument is located in xfer registers, returning 1 (true) or 0 (false).

8.2.33 __link_sym(SYM)

Parameters

SYM The name of a linker-allocated symbol of the kind created, for example, by `.alloc_mem` or `.alloc_resource`, as a string literal.

Description

This function provides access to linker symbols, and may be used (for example) to set up a pointer to a variable in external memory.

8.2.34 __LoadTimeConstant(EXPR)

Parameters

EXPR An import variable name or expression as a string literal

Description

This function provides access to import variables and expressions, and resolves to a 40-bit integer value.

8.2.35 __load_expr(EXPR)

Parameters

EXPR An import variable name or expression as a string literal

Description

This function provides access to import variables and expressions, and resolves to a 40-bit integer value. It has a few advantages over the older `__LoadTimeConstant` intrinsic, in particular supporting a greater variety and complexity of expressions.

8.2.36 __n_ctx()

Parameters

Description

This function returns the number of contexts for which the compiler is set up. This is the command line `-Qnctx` setting or a default. The result is an integer in the range 1 to 8.

8.2.37 __nctx_mode()

Parameters

Description

This function returns the context mode for which the compiler is set up. This is the command line `-Qnctx_mode` setting or a default. The result is an integer, either 4 or 8.

8.2.38 __nfp_idstr2meid(IDSTR)

Parameters

IDSTR An NFP "ID string" indicating island and ME number, as a string literal: for example `"i32.me1"`.

Description

This function returns the microengine ID associated with the 'idstr' argument.

8.2.39 __nfp_meid(ISLAND, ME_NUM)

Parameters

ISLAND   NFP island number as an integer
ME_NUM NFP ME number as an integer

Description

This function returns the microengine ID associated with the supplied island and ME number.

8.2.40 __no_spill_begin()

Parameters

Description

For internal use only: deprecated.

8.2.41 __no_spill_end()

Parameters

Description

For internal use only: deprecated.

8.2.42 __no_swap_begin()

Parameters

Description

For internal use only: deprecated.

8.2.43 __no_swap_end()

Parameters

Description

For internal use only: deprecated.

8.2.44 __signal_number(VAR[, ME_NUM])

Parameters

VAR	Variable of class signal
ME_NUM	Optional integer specifying the ME number where the signal resides, needed for remote signals

Description

This function returns the physical signal number allocated to the variable in question.

8.2.45 __signals([VAR, ...])

Parameters

VAR Zero or more signal, signal pair, or signal mask variables

Description

This function returns a signal mask constructed from the signals specified by the arguments.

8.2.46 __switch_pack(TYPE[, MAX_NOP])

Parameters

TYPE   An enum of type `swpack_t` specifying how the packing should be done

MAX_NOP Optional integer parameter specifying the maximum number of NOPs to be used (defaults to -1 meaning "auto")

Description

This function asks the compiler to perform the switch block packing optimization. (Here "pack" refers to packing the case bodies to the same size suitable for use with the `jump[]` instruction.

8.2.47 __vlexpr(EXPR)

Parameters

EXPR An linker variable name or expression as a string literal

Description

This function provides access to linker variables and expressions and is a special-purpose version of the more general intrinsic `__link_sym`. It is intended for use in cases where the associated numeric value is known to be in the range 0 to 255, and allows better optimization.

8.2.48 __wait_for_all([VAR, ...])

Parameters

VAR Zero or more signal or signal pair variables

Description

This function generates a `ctx_arb` on the set of signals specified by the arguments. The AND form of the instruction is used which will wait until all signals are set. In the case of a signal pair, both even and odd signals are included among the signals waited for.

8.2.49 __wait_for_all_single([VAR, ...])

Parameters

 VAR Zero or more signal or signal pair variables

Description

This function generates a `ctx_arb` on the set of signals specified by the arguments. The AND form of the instruction is used which will wait until all signals are set. In the case of a signal pair, only the even signal is included among the signals waited for.

8.2.50 __wait_for_any([VAR, ...])

Parameters

 VAR Zero or more signal or signal pair variables

Description

This function generates a `ctx_arb` on the set of signals specified by the arguments. The OR form of the instruction is used which will wait until any signal is set. In the case of a signal pair, both even and odd signals are included among the signals waited for.

8.2.51 __wait_for_any_single([VAR, ...])

Parameters

 VAR Zero or more signal or signal pair variables

Description

This function generates a `ctx_arb` on the set of signals specified by the arguments. The OR form of the instruction is used which will wait until any signal is set. In the case of a signal pair, only the even signal is included among the signals waited for.

8.2.52 __xfer_reg_number(VAR[, ME_NUM])

Parameters

VAR   Variable of class xfer register

ME_NUM Optional integer specifying the ME number where the variable resides, needed for remote xfer registers

Description

This function returns the number of the first physical transfer register allocated to the relevant variable.

8.2.53 __signal_test(VAR)

Parameters

VAR Variable of class signal

Description

This function tests whether a signal is set, returning returning 1 (true) or 0 (false). A set signal is cleared as a side effect of testing.

8.2.54 ua_get_s8(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function reads a signed 8-bit value from memory (of any type) and returns a signed 32-bit integer result.

8.2.55 ua_get_s16(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function reads a signed 16-bit value from memory (of any type) and returns a signed 32-bit integer result.

8.2.56 ua_get_s32(PTR, OFF)

Parameters

PTR An unaligned pointer to memory
OFF An unsigned integer offset

Description

This function reads a signed 32-bit value from memory (of any type) and returns a signed 32-bit integer result.

8.2.57 ua_get_s64(PTR, OFF)

Parameters

PTR An unaligned pointer to memory
OFF An unsigned integer offset

Description

This function reads a signed 64-bit value from memory (of any type) and returns a signed 64-bit integer result.

8.2.58 ua_get_u8(PTR, OFF)

Parameters

PTR An unaligned pointer to memory
OFF An unsigned integer offset

Description

This function reads an unsigned 8-bit value from memory (of any type) and returns an unsigned 32-bit integer result.

8.2.59 ua_get_u8_cls(PTR, OFF)

Parameters

PTR An unaligned pointer to CLS
OFF An unsigned integer offset

Description

This function reads an unsigned 8-bit value from CLS and returns an unsigned 32-bit integer result.

8.2.60 ua_get_u8_mem(PTR, OFF)

Parameters

PTR An unaligned pointer to MEM (CTM, IMEM, or EMEM)
OFF An unsigned integer offset

Description

This function reads an unsigned 8-bit value from MEM and returns an unsigned 32-bit integer result.

8.2.61 ua_get_u8_sram(PTR, OFF)

Parameters

PTR An unaligned pointer to SRAM
OFF An unsigned integer offset

Description

This function reads an unsigned 8-bit value from SRAM and returns an unsigned 32-bit integer result.

8.2.62 ua_get_u16(PTR, OFF)

Parameters

PTR An unaligned pointer to memory
OFF An unsigned integer offset

Description

This function reads an unsigned 16-bit value from memory (of any type) and returns an unsigned 32-bit integer result.

8.2.63 ua_get_u32(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function reads an unsigned 32-bit value from memory (of any type) and returns an unsigned 32-bit integer result.

8.2.64 ua_get_u64(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function reads an unsigned 64-bit value from memory (of any type) and returns an unsigned 64-bit integer result.

8.2.65 ua_memcpy(DST, DST_OFF, SRC, SRC_OFF, LEN)

Parameters

DST   An unaligned pointer to memory (destination)

DST_OFF An unsigned integer offset (destination)

SRC   An unaligned pointer to memory (source)

SRC_OFF An unsigned integer offset (source)

LEN   An unsigned integer length

Description

This function does a `memcpy`-like operation from one memory location to another. The source and destination may be different memory types.

8.2.66 ua_set_8(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function writes an unsigned 8-bit value to memory (of any type).

8.2.67 ua_set_16(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function writes an unsigned 16-bit value to memory (of any type).

8.2.68 ua_set_32(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function writes an unsigned 32-bit value to memory (of any type).

8.2.69 ua_set_64(PTR, OFF)

Parameters

PTR An unaligned pointer to memory

OFF An unsigned integer offset

Description

This function writes an unsigned 64-bit value to memory (of any type).

9. Mutual Exclusion Library

9.1 Introduction

The mutual exclusion locks (mutexes) provided in this library are designed to prevent multiple contexts of an ME from simultaneously executing critical sections of code that access shared data. In other words, mutexes are used to serialize the execution of a microengine's contexts.

All mutexes must be global. A successful call for a mutex lock via `MUTEXLV_lock()` will cause another thread that is also trying to lock the same mutex to block until the owner thread unlocks it via or `mutex_unlock()`. Threads within the same micro engine can share mutexes.

The MUTEXLV uses the construct:

```
__declspec(shared gp_reg)
```

which is a shared general purpose register (across threads in a ME). If the compiler cannot allocate the `gp_reg` object in a register, it reports an error and aborts.

The MUTEXLV are implemented via macros, as they must manipulate the register object directly. A single MUTEXLV object is capable of 32 mutex(s), each referred to with a user specified unique id (MUTEXID) from [0 .. 31], not necessarily a constant. There is no range checking on the required id.

To coordinate threads on multiple microengines, there are microengine global mutexes.

The MUTEXG uses the construct:

```
__declspec(import | export)
```

which by default are shared volatile variables across microengines.

MUTEXG_IMPORT and MUTEXG_EXPORT are implemented as macros. They are functionally similar to the MUTEXLV macros, except they are not vector valued (that is, there is only one available per declaration).

> **Note**
>
> There is no corresponding functions for the semaphore library.

9.2 MUTEXLV Usage

```
MUTEXLV lock= 0;
    f(MUTEXID handle)
    {
        ...
```

```
        MUTEXLV_lock(lock, handle)

            // lock data/code access to only one ME local thread
        MUTEXLV_unlock(lock, handle)
        ...
    }
```

If the handle value is reused, you must use the correct barrier synchronization (semaphore).

9.3 MUTEXG Usage

```
MUTEXG lock= 0;
    f(MUTEXID handle) {
    ...
    MUTEXG_lock(lock, handle)

    // lock data/code access to one ME thread
        MUTEXG_unlock(lock, handle)
        ...
    }
```

9.4 Functions

9.4.1 MUTEXLV_init (MUTEXLV)

Parameters

 MUTEXLV mutex object to be initialized.

Description

Initialize all ids to zero.

This is very hard to use, as you must insure the mutex is initialized only once (either globally, or with some semaphore).

Errcode

No error code is returned.

9.4.2 MUTEXLV_destroy(MUTEXLV,MUTEXID)

Parameters

MUTEXLV mutex object.

MUTEXID handle to specific mutex id.

Description

This clears the specific [mutex, id] in mutex.

Errcode

No error code is returned.

9.4.3 MUTEXLV_lock(MUTEXLV, MUTEXID)

Parameters

MUTEXLV mutex object.

MUTEXID handle to specific mutex id.

Description

This tests and blocks a specific [mutex, id] for a lock. If busy, the current context is swapped out. If free, the [mutex,id] is set and ERRCODE_EOK is returned in supplied argument. There is no sense of ctx() ownership or recursion. If the same thread tries to reacquire the lock (that it already owns), it too will block.

Errcode

No error code is returned.

9.4.4 MUTEXLV_unlock(MUTEXLV, MUTEXID)

Parameters

MUTEXLV mutex object.

MUTEXID handle to specific mutex id.

Description

Unlock the [mutex, id]. There is no sense of ctx() ownership (i.e. a different thread may unlock the object).

Errcode

No error code is returned.

9.4.5 MUTEXLV_trylock(MUTEXLV, MUTEXID, ERRCODE)

Parameters

MUTEXLV:	mutex object.
MUTEXID:	handle to specific mutex id.
ERRCODE:	specific error code returned.

Description

This tries to acquire the [mutex, id] (if free) but does not block.

Errcode

ERRCODE_EBUSY	lock is busy.
ERRCODE_EOK	lock has been acquired.

9.4.6 MUTEXLV_testlock(MUTEXLV, MUTEXID, ERRCODE)

Parameters

MUTEXLV	mutex object.
MUTEXID	handle to specific mutex id.
ERRCODE	specific error code returned.

Description

This tests but does not acquire the [mutex, id].

Errcode

ERRCODE_EBUSY	lock is currently spinning.
ERRCODE_EOK	lock is free (i.e. may be acquired).

9.4.7 MUTEXG_init (MUTEXG)

Parameters

MUTEXG mutex object to be initialized.

Description

Initialize to zero. You must insure the mutex is initialized only once, either globally or through a semaphore.

Errcode

No error code is returned.

9.4.8 MUTEXG_destroy (MUTEXG)

Parameters

MUTEXG mutex object.

Description

Clears the specified mutex object.

Errcode

No error code is returned.

9.4.9 MUTEXG_lock (MUTEXG)

Parameters

MUTEXG mutex object.

Description

This tests and blocks a specific mutex for a lock. If busy, the current context is swapped out. If free, the mutex is set and ERRCODE_EOK is returned in the supplied argument. There is no concept of ctx() ownership or recursion. If the same thread tries to re-acquire the lock that it already owns, it too will block.

Errcode

No error code is returned.

9.4.10 MUTEXG_unlock (MUTEXG)

Parameters

MUTEXG mutex object.

Description

Unlock the mutex. There is no concept of ctx() ownership (that is, a different thread may unlock the object).

Errcode

No error code is returned.

9.4.11 MUTEXG_trylock (MUTEXG, ERRCODE)

Parameters

MUTEXG   mutex object.
ERRCODE specific error code returned.

Description

This attempts to acquire the [mutex, id] if it is free, but does not block.

Errcode

ERRCODE_EBUSY Lock is busy.
ERRCODE_EOK   The lock has been acquired.

9.4.12 MUTEXG_testlock (MUTEXG, ERRCODE)

Parameters

MUTEXG   mutex object.
ERRCODE specific error code returned.

Description

This tests but does not acquire the [mutex, id].

Errcode

ERRCODE_EBUSY Lock is currently spinning.
ERRCODE_EOK   The lock is free (that is, it may be acquired).

---

10. Semaphore Library

10.1 Semaphore Data Types

```
typedef unsigned int SEMVALUE;


typedef volatile __declspec(shared gp_reg) struct SEML
{
    unsigned barrier:1;
    unsigned init:1;
    unsigned reserved:14;
    unsigned initval:8;
    unsigned val :8;
} SEML
```

10.2 Semaphore Functions

10.2.1 SEML_init(SEML, SEMVALUE)

Parameters

SEML	Semaphore object.
SEMVALUE	Initial value of the semaphore counter (max 0xff).

Description

This function initializes an unnamed semaphore. The initial value of the semaphore is set to 'value'. The semaphore may also be initialized globally by:

```
SEML sem= SEML_init_list(value);
```

Errcode

No error code is returned.

10.2.2 SEML_destroy(SEML)

Parameters

SEML Semaphore object.

Description

This function destroys an unnamed semaphore.

Errcode

No error code is returned.

10.2.3 SEML_post(SEML) SEML_dec(SEML)

Parameters

SEML Semaphore object.

Description

This function will post (or dec) a wakeup to a semaphore. If there are waiting threads, one is unblocked; otherwise, the semaphore value is incremented (decremented) by one.

Errcode

No error code is returned.

10.2.4 SEML_wait(SEML)

Parameters

SEML Semaphore object.

Description

This function waits on a semaphore. If the semaphore value is greater than zero, it decreases its value by one. If the semaphore value is less than or equal zero, then the calling thread is blocked until it can successfully decrease the value.

Errcode

No error code returned.

10.2.5 SEML_trywait(SEML, ERRCODE)

Parameters

SEML   Semaphore object.

ERRCODE

Description

Similar to SEML_wait except that if the semaphore value is zero, then this function returns immediately with the error EAGAIN.

Errcode

ERRCODE_EAGAIN The semaphore was already locked.

ERRCODE_OK

10.2.6 SEML_barrier(SEML,n)

Parameters

SEML Semaphore object.

Description

This function performs the dual of SEML_wait. It is used to provide a synchronization point for threads to join by waiting on a semaphore. If the semaphore value is greater than zero, then the calling thread is blocked until it can successfully increase the value. If the semaphore value is zero, it increases its value by one.

Errcode

No error code returned.

10.2.7 SEML_trybarrier(SEML, ERRCODE)

Parameters

SEML   Semaphore object.

ERRCODE

Description

Similar to SEML_barrier except that if the semaphore value is less than or equal zero, then this function increments the value.

Errcode

ERRCODE_EAGAIN The semaphore was already locked.

ERRCODE_OK

10.2.8 SEML_getvalue(SEML)

Parameters

SEML Semaphore object.

Description

Return the value associated with the semaphore.

Errcode

No error code returned.

10.2.9 SEML_set_barrier_at(SEML,n) SEML_clr_barrier_at(SEML,n)

Auxiliary macros used by semaphore macros.

11. Technical Support

To obtain additional information, or to provide feedback, please email `<support@netronome.com>` or contact the nearest **Netronome** technical support representative.

Appendix A. Qperfinfo Output Information

This appendix provides additional information for the -Qperinfo command-line option.

A.1 -Qperfinfo=1

Function: Variable spill

Description

Provides information for all GPR variables spilled into local memory, cluster local scratch or IMEM during register allocation.

> **Note**
>
> This does not include variables spilled into NN (next neighbor) registers.

Format

```
foo.c(217): warning: V1 (class cls_gpr) spilled to localmem because of register pressure

                                                    Spilled memory region
                                                    Register class
                                                    Variable name



                        Source line number
                        Source file name
```
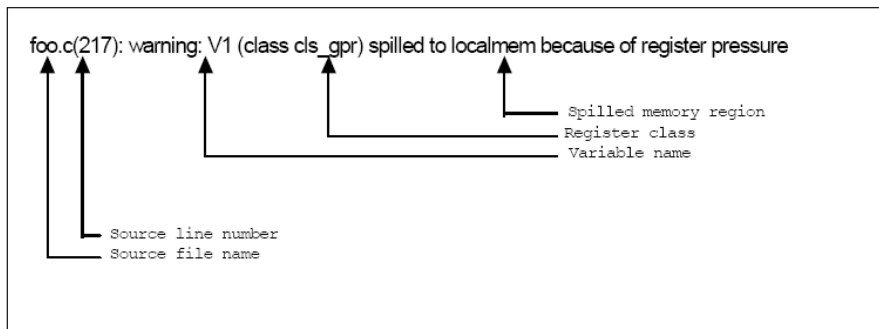
Example

```
foo.c(217): warning: V1 (class gp_reg) spilled to localmem because of register pressure.
foo.c(218): warning: V2 (class gp_reg) spilled to localmem because of register pressure.
foo.c(219): warning: V3 (class gp_reg) spilled to localmem because of register pressure.
foo.c(220): warning: V4 (class gp_reg) spilled to imem because of register pressure.
foo.c(221): warning: V5 (class gp_reg) spilled to imem because of register pressure.
foo.c(222): warning: V6 (class gp_reg) spilled to imem because of register pressure.
```

A.2 -Qperfinfo=2

Function: Live range and allocation

Description

Provides instruction-level symbol liveliness and register allocation.

Format

```
Format:

Source file name          Function begin and          Register holding
                          end source line             return address
            Function name  number

FILE: "c:/MEv2-2/uEngineC/src/testutil.c"
{
  function put line#99..#120 PC=2318..2325, rtn=A1          Machine instruction
  {                                                          range of this
    c -> +0 (not enregistered)                               function in .list file
    lockp -> +0 A2 2319:2324
    locked -> +0 srw$0 2320:2322 2324:2324
    ch -> +0 (not enregistered)
    s1 -> +0 (not enregistered)                             Register live range in
    s2 -> +0 sig2 2321:2321                                 terms of instruction
         +4 sig3 2321:2321                                  counts.
    L74 -> spilled to lmem$tls+88, lmem$tls-si:
  }
                                                   Physical register
  ... .. ...      Variable offset                  allocated to, or why it is
}                                                  not allocated.

         Variable name
```
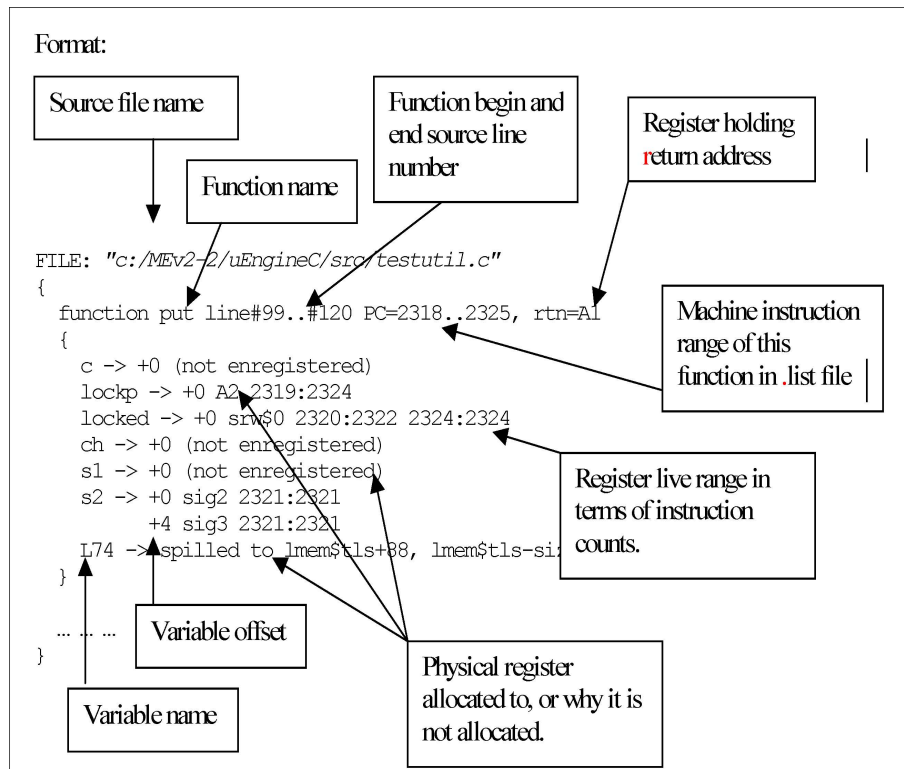
Notes

- Live range is in the format of beginning_instruction_#:end_instruction_#.

- Some variables are not allocated to registers because compiler removed this variable, or it is merged into another variable.

- Some variables do not have a recognizable name. They are compiler generated.

- Physical registers format:

  - An: A bank register

  - Bn: B bank register

  - sign: signal register

  - xr$n: sram/xfer read register

- dr$n: dram read register

- xw$n: sram/xfer write register

- dw$n: dram write register

- xrw#n: sram/xfer read/write register

Example

```
//----------------------------------------------------------------------
{
   FILE: "largeRegSpill_0_1.c"
   {
   function main line#28..#1006 PC=0..2315, rtn=A0
   {
   L0 -> spilled to lmem$tls+0, lmem$tls-size=256
   NOTE: lmem$tls start = 0
       L1 -> spilled to lmem$tls+4, lmem$tls-size=256
       L2 -> spilled to lmem$tls+8, lmem$tls-size=256
       L3 -> spilled to lmem$tls+12, lmem$tls-size=256
       L4 -> spilled to lmem$tls+16, lmem$tls-size=256
       L84 -> +0 A24 157:1543
       L85 -> +0 B23 160:1552
       L86 -> +0 A23 163:1561
       L128 -> +0 A1 244:1939
       L129 -> +0 A0 245:1948
       L130 -> spilled to lmem$tls+92, lmem$tls-size=256
       L131 -> spilled to lmem$tls+96, lmem$tls-size=256
       L133 -> spilled to lmem$tls+104, lmem$tls-size=256
       }
   }
FILE: "c:\MEv2-2\uEngineC\src\rtl.c"
   {
   function exit line#50..#65 PC=2316..2317, rtn=A0
   {
   status -> +0 (not enregistered)
   }
   }
   FILE: "c:\MEv2-2\uEngineC\src\testutil.c"
   {
   function put line#99..#120 PC=2318..2325, rtn=A1
   {
     c -> +0 (not enregistered)
   lockp -> +0 A2 2319:2324
   locked -> +0 srw$0 2320:2322 2324:2324
   ch -> +0 (not enregistered)
   s1 -> +0 (not enregistered)
   s2 -> +0 sig2 2321:2321
   +4 sig3 2321:2321
   }

   function putui line#179..#199 PC=2326..2335, rtn=A1
   {
     val -> +0 (not enregistered)
   lockp -> +0 A2 2327:2334
   locked -> +0 srw$0 2329:2331 2334:2334
   h -> +0 sw$1 2328:2332
   s1 -> +0 (not enregistered)
```

```
    s2 -> +0 sig2 2330:2330
    +4 sig3 2330:2330
    }
    }
    }
```

A.3 -Qperfinfo=4

Function: (None)

Description

This option has been deprecated and replaced with the -Qliveinfo option. Please see Section 4.10 for more information:

A.4 -Qperfinfo=8

Function: Function size

Description:

Provides the number of instructions in each function.

Example:

| function:	size:
_exit:	2
_put:	8
_putui:	10
_mput$5:	16
_putns:	38
_putsi:	13
_main:	2316
Total size:	2403

A.5 -Qperfinfo=16

Function: Local memory allocation

Description

Provides information about how the compiler allocates local memory. Each local memory variable (either user-defined or a compiler-generated spill variable) belongs to a local memory group. Each local memory group contains one or several local memory variables allocated within a 64-byte range. Access to two local variables in the same group can share the same local memory base pointer if its value is still valid.

Eventually, all the groups are divided into two categories: thread-local group and shared group. All threads share the shared groups, and each thread has its own copy of the thread-local groups.

Layout (TLS for thread-local-storage):
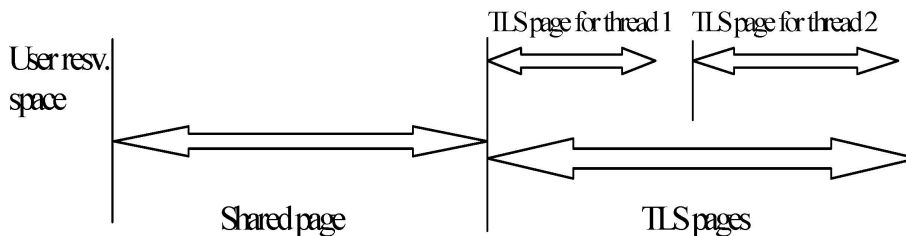
Notes

- You can reserve an area of local memory from compiler allocation through a command-line option.

- The offset field in the printout is the offset from the beginning of the thread-local page for thread-local data, or from the beginning of the shared page for shared data. They are not offsets within the group.

Example

=> User reserved: 0 bytes, Shared segment: 64 bytes, Local page (including gap): 320 bytes

=> Gap between context pages is 40 bytes The data on the page is 280 bytes

Direct access local mem group 0x14ea3f8

Maximum offset used: 60 Alignment: 64

Num members: 1 Total size: 128

[This group contains thread local symbols]

  copylmemsram.c(126): tmp allocated at offset 0

Direct access local mem group 0x1516c04

Maximum offset used : 12 Alignment: 4

Num members: 4 Total size: 16

[This group contains thread local symbols]

  copylmemsram.c(240): _m1 allocated at offset 128
  copylmemsram.c(241): _m2 allocated at offset 132

copylmemsram.c(242): _m3 allocated at offset 136

copylmemsram.c(243): _m4 allocated at offset 140

Direct access local mem group 0x1516dfc

Maximum offset used: 12 Alignment: 4

Num members: 4 Total size: 16

[This group contains shared symbols]

copylmemsram.c(244): _p1 allocated at offset 0

copylmemsram.c(244): _p2 allocated at offset 4

copylmemsram.c(244): _p3 allocated at offset 8

copylmemsram.c(244): _p4 allocated at offset 12

A.6 -Qperfinfo=32

Function: Interference information for variables spilled into IMEM

Description

When a GPR variable is spilled into IMEM, this information prints out all the other variables that interfere with this one. Two variables interfere with each other when they can not be allocated to the same register.

Example

```
foo.c(221):L24 conflicts with:
foo.c(197):L0 foo.c(198):L1 foo.c(199):L2 foo.c(200):L3
foo.c(201):L4 foo.c(202):L5
```

> **Note**
>
> Variables are in the format: <source file>(<source line>):<name>

A.7 -Qperfinfo=64

Function: Scheduler statistics

Description:

The instruction scheduler moves instructions to fill delay slots. The -Qperfinfo printout for the scheduler looks like the following example.

Example

```
/*
* Scheduler Summary
*/

Nop(s) removed: 53 (8.5% of total 624)
```

In this example, there were a total 624 nops in the whole program, and the compiler was able to remove 53 of them.

A.8 -Qperfinfo=128

Function: Warn if the compiler cannot detect the I/O buffer size

Description

The I/O intrinsic functions such as cls_write() allow you to pass a non-constant count argument. In this case, the compiler will generate the "indirect" form of the I/O instruction. However, the compiler will need to know how many transfer registers to reserve for the I/O transfer. Since the compiler does not know the exact reference count, it will assume that the whole variable is used in the intrinsic.

Example

```
int __declspec(write_reg) rr[8];
..
cls_write(&rr, addr, size, ctx_swap, &s1);
// size is unknown at compile time
```

The compiler will reserve 8 transfer registers for the CLS transfer in this example, although at runtime only a smaller number may be needed. -Qperfinfo=128 will print warnings when the compiler detects a situation similar to the above.

Example

`C:\CVS\include\align.h`(124): warning: cls_write(): Size of data access cannot be determined at compile-time. __implicit_read/write may be needed to protect xfer buffer. Use of cls_write_ind() is recommended instead.

A.9 -Qperfinfo=256

Function: Display information on restrict pointer optimization

Description

The compiler can optimize dereferences of pointer parameters declared with the "restrict" keyword (see Section 4.8.4.1 for details). This allows objects passed to such functions to be allocated to registers. Not all "restrict"

pointer parameters can be optimized. The -Qperfinfo=256 option shows which parameters are optimized, and provides information about the parameters which are not optimized.

The argument name may be printed in its internal modified form. It has the format:

<user_name>_<compiler_mangling_string>.

Example

Pointer argument xyz_379_V$200$1$1 in function _foo was optimized and dereferences were eliminated.

A.10 -Qperfinfo=512

Function: Compiler printout for jump target offsets.

Description

The compiler supports the jump[] instruction in inline assembly code. You have to make sure that the offsets passed to the jump[] instruction match valid target labels in the inline assembly. - Qperfinfo=512 provides the offset of each symbolic label listed in a jump target list.

Example

For code like:

```
__asm __attribute(ASM_HAS_JUMP)
{
    br[jmp]
base0:
    immed[result, 1]
    br[last]
base1:
    immed[result, 2]
    br[last]
jmp:
    jump[offset, base0], targets [base0, base1, base2, base3]
base2:
    immed[result, 3]
    call[foo2]
    br[last]
base3:
    immed[result, 4]
    call[foo2]
    br[last]
last:
}
```

Compiler -Qperfinfo=512 printout has the following appearance:

test_jump.c(55): inline-asm jump may have potential offset(s) = 0, 2, 5, 9

A.11 -Qperfinfo=1024

Function: Boolean propagation optimization

Description

The compiler performs an optimization that determines the value of a constant conditional based on the result of other conditionals.

Example

```
if (1) {
    x = 5;
}
else {
    x = 6;
}
if (x == 5) {
...
}
```

When this optimization occurs, this -Qperfinfo option displays results as the following example shows:

```
[1] Bool_prop transformation performed
[2] Bool_prop transformation performed
```

A.12 -Qperfinfo=2048

Function: Register requirements report

Description

This -Qperfinfo option displays a report on areas of your program that require a large number of "live" registers, indicating possible spill areas. For a more detailed description of the concepts of liveness and spilling, please refer to Section 4.9.

Example

```
-------------------- GPR Requirements Report ----------------------
0 relative A bank GPRs were used for shared variables
0 relative B bank GPRs were used for shared variables
3 thread local variables can be colored using abs registers
1 relative A bank GPRs used as absolute regs
1 relative B bank GPRs used as absolute regs
15 relative A registers available for coloring
15 relative B registers available for coloring
409 variables colored with relative GPRs
```

```
59 A bank and 57 B bank GPRs were needed for coloring
118 total GPRs needed to color without any spilling

Start of high GPR usage region in function _mput$5
Starting at c:\\clin1\\ixp1200/uEngineC/src/../samples/util/util.c line 414
   End of high GPR usage region at

c:\\clin1\\ixp1200\uEngineC\src\..\samples\util\util.c 415
Max usage is (115) at c:\\clin1\\ixp1200\uEngineC\src\..\samples\util\util.c
415
```

> **Note**
>
> Formatting can vary, depending upon text content and line breaks.

(Note that formatting can vary, depending upon text content and line breaks.)

"Coloring" is the process of assigning registers to variables. In the above example, all program variables must be assigned to one of 30 registers (15 in each bank). The region of the program between lines 414 and 415 in the file util.c would require 115 free registers to avoid spilling. Since the number of registers is fixed by the architecture, if you wish to avoid spilling, the program code or data must be restructured so that 115-30 = 85 fewer words of data are live in the indicated region.

A.13 -Qperfinfo=4096

Function: Switch optimization report

Description: Appendix , "User-Guided switch() Statement Optimization"

The compiler can perform an optimization, described in Section 7.3.2, which creates faster code for switch() statements This - Qperfinfo option generates a report on which switch() statements in your program were optimized, and how the jump[] calculation was performed.

Example

```
/*
* Switch Pack Report
*/

Function: _main,
      .switch[$0, SW, l_11#, swpack_auto, nlive=1, a1, b0, a0]
    Switch has 3 target(s); Min/Max-distance = 2/2
    Estimated PC(s): 31 33 35
    0 nop(s) needed
    Sequence to compute new jump index y(b0) = x($0) * 2: (t = a1)
      y = x<<1
    >>> Pack it
```