Intel® Internet Exchange Architecture Optimized Data Plane Libraries

Reference Manual

November 2003

Document Number: 278744-004

Contents

Figures

Tables

Revision History

Date	Revision	Description
February 2003	001	First release of this information as a stand-alone document included with SDK 3.0 Release 6. (Previously released as part of the Intel ® Internet Exchange Architecture (IXA) Portability Framework Reference Manual).
June 2003	002	Updated to add Crypto APIs and enumerations.
September 2003	003	IXA SDK 3.5 Tools Pre-Release 1
November 2003	004	IXA SDK 3.5 Tools Pre-Release 2. Removed Crypto Library. Added Relocatable Data Structure API and CAM Sharing API

Introduction 1

1.1 About this Document

This Reference Manual introduces you to the Intel® Internet Exchange Architecture (IXA) Optimized Data Plane Libraries, which are a part of the Intel IXA® Software Development Kit (SDK) 3.5 Tools. This software enables you to develop and deliver network applications that utilize the Intel® IXP2400, IXP2800 and IXP2850 Network Processors.

The IXA Optimized Data Plane Libraries are a network application framework and infrastructure for writing modular and portable microengine code, which:

- Saves time by providing robust microengine infrastructure software and APIs
- Permits portability across IXA network processors

1.2 Audience

This guide is intended for software developers who will design, develop, and deliver network applications that must process packets at high speed. It assumes that you are familiar with the following:

- C or assembler programming
- Realtime network applications

1.3 In This Manual

This manual includes the following chapters:

- Chapter 1, "Introduction," this chapter provides an overview of this manual.
- Chapter 2, "Optimized Data Plane Libraries Overview,"provides an overview of the optimized data plane libraries, the topic of Chapter 3 through Chapter 5.
- Chapter 3, "Hardware Abstraction Layer for the Microengine," documents the microengine hardware abstraction interfaces, "Instruction Simplification Interface" and "OS Emulation Interface."
- Chapter 4, "Utilities for the Microengine," describes various utility APIs supporting data plane processing on microengines.
- Chapter 5, "Protocol Support for the Microengine," describes protocol-level support for data plane processing on microengines.

1.4 Other Sources of Information

This manual is part of the Intel® Internet Exchange Architecture Portability Framework documentation set, which also includes the following documents:

- *Intel® Internet Exchange Architecture Portability Framework Developer's Manual*
- *Intel® Internet Exchange Architecture Portability Framework Reference Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Developer's Manual*
- *Intel® Internet Exchange Architecture Software Building Blocks Reference Manual*
- *Intel® Internet Exchange Architecture Software Reference Manual*
- *Intel® Internet Exchange Architecture Software Development Kit Tools Getting Started Guide*
- *Intel® Internet Exchange Architecture Software Development Kit Software Framework Getting Started Guide*
- *Development Tools User's Guide*
- *Help Topics: Developer Workbench*
- *Intel ® IXP2400/IXP2800 Network Processors Microengine C Compiler Language Support Reference Manual*
- *Intel ® IXP2400/IXP2800 Network Processors Microengine C Compiler LibC Library Reference Manual*
- *Intel® IXP2400/IXP2800 and Network Processor Programmer's Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

The following documents are available from the Internet.

- *Intel® IXP2400 Network Processor Datasheet*
- *Intel® IXP2800 Network Processor Datasheet*
  They can be retrieved at: http://fdbl.intel.com.

Optimized Data Plane Libraries Overview 2

2.1 Purpose

This part of the reference describes the libraries of microengine macros and functions supporting data plane program development. The purpose of these libraries is to relieve the application programmer of low-level programming tasks while providing portability across the following processors:

- Intel® IXP2400 Network Processor (see *Intel® IXP2400 Network Processor Hardware Reference Manual* for details)
- Intel® IXP2800 Network Processor (see *Intel® IXP2800 Network Processor Hardware Reference Manual* for details)
- Intel® IXP2850 Network Processor (see *Intel® IXP2850 Network Processor Hardware Reference Manual* for details)

2.2 Scope

The applications that run on Intel® Network Processor microengines are typically used for packet classification and forwarding in network devices such as routers, switches, and gateways.

2.3 Chapter Organization

This chapter provides an overview of the Optimized Data Plane libraries. This chapter also describes naming and formatting conventions that assist the reader in interpreting the API documentation in this reference manual.

2.4 Overview

The optimized data plane libraries consist of generic microengine software building blocks used to construct an application's microengine modules—called microblocks. Microblocks are reusable microengine code modules built using the portability framework. For more information on microblocks, see *Intel® Internet Exchange Architecture (IXA) Portability Framework Developer's Manual*.

The optimized data plane macro libraries are reusable software functions optimized for high performance and minimal executable code size. Programs written with macro libraries are structured assembly directives that facilitate creation of software programs that are easy to read, understand and maintain without sacrificing real-time performance.

Figure 2-1 graphically lists the Optimized Data Plane libraries. Table 2-1 describes the content of each of these libraries.

Figure 2-1. Graphical Overview of the Optimized Data Plane Libraries

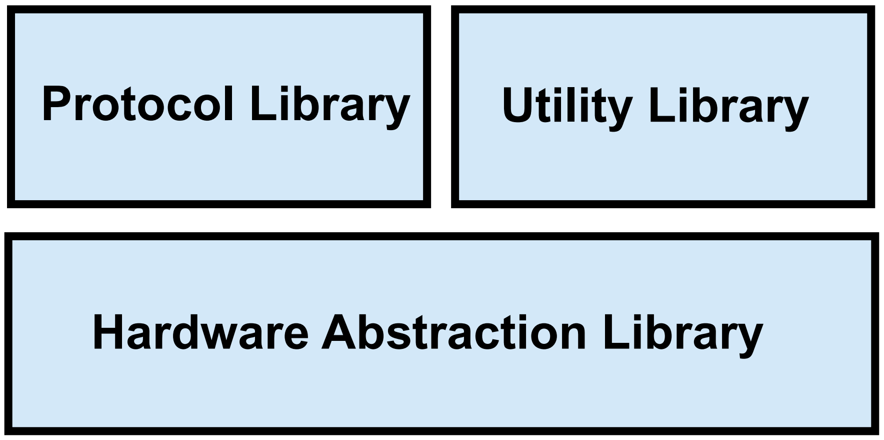


Table 2-1. Summary of the Optimized Data Plane Libraries shown in Figure 2-1.

Hardware Abstraction Library (HAL)	Provides operating system type abstraction of hardware-assisted functions such as: • Memory and Buffer Management • Critical Section Management • Inter-Process Communication • Bus I/O • Control and Status Registers
Utility Library	Provides a range of data structures and algorithm support including generic table lookups, endian swaps, and other useful functions.
Protocol Library	Provides an interface supporting link layer protocols (L2) as well as network layer protocol (L3) support through a combination of structures and functions.

2.5 Methodologies and Notations

This section specifies the interface and operation names for both Microengine Assembler macros and Microengine C language functions or macros. The interfaces in each category are listed in alphabetic order.

Operation names are the same for similar functions, except as described below.

The following notation is used in the API, to avoid redundant lists for C functions and assembler macros

```
[ixp_]interfacename_operationname
```

where C functions use the `ixp_` prefix and assembler macros do not.

For example, the `buf_alloc` API entry appears as the macro `buf_alloc` in Microengine Assembler and as the function `ixp_buf_alloc` in Microengine C.

Microengine Assembler macro arguments and Microengine C function arguments also differ in that C functions declare types, may return a value, and may take a pointer argument. For those API entries whose Microengine C function returns a value which is equivalent to a Microengine Assembler macro argument, the C return value is documented under the assembler argument name with a notation indicating that it is also a return value. For example:

Output/Returns

`out_argument` ***or*** Return Value	A description of the C return value and assembler output argument.

2.5.1 Common Style for Macros and Functions

2.5.1.1 Arguments

Output registers or memory variables are lower case words separated by underscore (_), starting with `out_`. Output arguments are the left-most arguments to the function. These must be passed by reference.

Input/output registers or memory variables are lower case words separated by underscore (_), starting with `io_`. These follow output arguments, if any. These must be passed by reference.

Input registers or memory variables are lower case words separated by underscore (_), starting with `in_`. These follow output arguments and input/output arguments, if any.

2.5.1.2 Estimated Size

Each operation is defined with an estimated size expressed as the number of optimized microengine instructions to execute the operation. This size assumes compiler optimization is turned on.

2.5.2 Languages

This API is specified for two libraries, one for Microengine Assembler macros and the other for Microengine C functions and macros. Source code modules must be written in either one language or the other, but not both. Compiled or assembled results from different language sources can, however, be linked to form a microengine application. Refer to *IXP2400/IXP2800 Development Tools User's Guide* for further information.

2.5.2.1 Microengine Assembler Language

The term Microengine Assembler is used to refer to Intel Network Processor Structured Microengine Assembly Language, also known as microcode written as macros. All Microengine Assembler operations are specified as macros.

2.5.2.1.1 Constants as Arguments

The file `constants.h` defines the maximum size of an immediate value in an instruction as:

For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors:

```
#define MAX_IMMEDIATE 0xFF
```

For Intel® IXP1200 Network Processor the equivalent definition is:

```
#define MAX_IMMEDIATE 0x1F
```

This is used by macros to accept registers or constants, while inserting the appropriate `immed*` instructions to use constants. In general for all macros that accept constants, the following holds true when estimating size:

- One additional instruction per constant when it is less than or equal to 16 bits in length
- Two additional instructions per constant when it is greater than 17 bits in length

2.5.2.1.2 File Names

Each interface is packaged in a separate file. For example,

- `buf.uc`
- `dram.uc`
- `ipv4.uc`

2.5.2.1.3 Macro Names

Macro names are lowercase words separated by underscore (_).

The first word is the interface name.

The second and subsequent words comprise the operation name.

2.5.2.1.4 Microengine Assembler Macros

Sometimes it takes several microengine instructions to perform a task due to the specific restrictions of certain instructions—for example, which types of registers can be used, or how many immediate data bits can be coded into the instruction. Microengine Assembler macros simplify the coding of basic microengine operations, such as loading registers with immediate values, using constants of different sizes in ALU operations, and performing indirect shifts. For example, constants of up to 32 bits or general-purpose registers may be used as arguments in move and arithmetic operations. All operations specified in Chapter 3, “Hardware Abstraction Layer for the Microengine,” Chapter 4, “Utilities for the Microengine,” and Chapter 5, “Protocol Support for the Microengine” are implemented as Microengine Assembler macros. Refer to the Section 2.5, “Methodologies and Notations” for a discussion of naming conventions.

2.5.2.2 Microengine C

2.5.2.2.1 Simplified Function Syntax

For the purpose of simplified function syntax, the following definitions are contained in `ixp_lib.h`.

Microengine C Syntax

```
// since this is not C++, the compiler uses __inline
// to define an inline function.
// extern is used because inline functions of this library
// are external to(in separate file) the user application
#define INLINE static __forceinline
// shorthand for more compact function prototypes
#define UINT uint32_t
// define those types that must be immediate arguments, that is,
// when not using an immediate will result in a compilation error.
#define IMMED const UINT
// define return values
#define TRUE 1
#define FALSE 0
```

2.5.2.2.2 File Names

Each interface is packaged as a separate source file, for example,
`ixp_buf.c`

2.5.2.2.3 Function Declarations

Since a call to a function translates to a branch instruction, most of the functions are declared as `INLINE`.

2.5.2.2.4 Function Names

Function names are lowercase words separated by underscore (_).

The first word is the module name `ixp`.

The second word is the interface name.

The third and subsequent words comprise the operation name.

2.5.2.2.5 Function Arguments

The C language function arguments follow the naming conventions outlined in Section 2.5.1.1, “Arguments.”

2.5.2.2.6 Enumerations

If a constant argument has a limited number of values, an enumerated type is used. The argument name is specified in upper case.

2.5.2.2.7 Immediates

Arguments that must be immediate values to satisfy compiler intrinsic restrictions must be declared as `IMMED`—that is, if the argument is not an immediate, the compilation fails with an error message to this effect. The argument name is specified in lower case.

2.5.2.2.8 Constant `uint32_t`

The argument name for `const uint32_t` is specified in lower case.

2.5.2.2.9 Function Return Values

A function may return a value if there is only one output argument of type `int`, and the function can be used in an expression. For example, return values can include error status, results of calculations, and so on.

2.5.2.3 Microengine C Functions and Assembler Macros

Many of the Microengine C language functions provide a layer just above intrinsics which gives the user control over features unique to the Intel® IXP2400, IXP2800 and IXP2850 Network Processors while still maintaining independence from chip specifics such as the format of control and status registers (CSRs), indirect commands, and so on. For example, some of the Microengine C Optimized Data Plane library functions enable the programmer to control concurrent memory access and the synchronization of multiple signal events.

Note: Not all operations specified in this reference are implemented as Intel® Network Processor Microengine C Language functions. For example, the C compiler correctly handles use of constants, size of immediate values, and indirect shifts eliminating the need for explicit Microengine C language functions to provide this capability. Interfaces or operations provided in *Microengine Assembler only* are noted.

- This library is dependent only on the C language syntax and intrinsics as specified in the *IXP2400/IXP2800 Development Tools User's Guide* and *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler Language Support Reference Manual.* The mechanism for selecting chip-specific intrinsics is described in the next section.

2.5.3 Portability Issues

This section describes portability issues relevant to Intel® Network Processor chip application programmers.

2.5.3.1 Chip Version

The Microengine C compiler and Microengine Assembler symbolic constant, `CHIP_VERSION`, identifies the chip and version directing the compiler or assembler to emit code appropriate to that chip.

The `CHIP_VERSION` compiler define determines the instruction code syntax an operation generates. This symbolic constant is defined in the file:

```
constants.uc
```

There may be significant variations of instruction code and features between the Intel® IXP2400 Network Processor, Intel® IXP2800 Network Processor, Intel® IXP2850 Network Processor, and future Intel® Network Processor chip generations. The value of `CHIP_VERSION` directs the compiler to emit code utilizing chip-specific hardware features. Compile-time error messages are generated for unsupported interface usage.

To run on the Intel® IXP2400, IXP2800 and IXP2850 Network Processors, insert the following into application code or build switches:

```
#define CHIP_VERSION IXP2000
```

2.5.3.2 Addressing

The Intel® IXP1200 Network Processor and the Intel® IXP2400, IXP2800 and IXP2850 Network Processors handle addresses differently. Table 2-2 illustrates these differences.

Table 2-2. Processor Addressing Differences

Memory Type	Intel® IXP1200 Network Processor	Intel® IXP2400, IXP2800 and IXP2850 Network Processors
DRAM Memory, RBUFs, and TBUFs	Use 64-bit quadword addresses. Data is read or written in 64-bit quadwords.	Use byte addresses. The memory unit ignores the low three address bits. Data is read or written in 64-bit quadwords.
SRAM and Scratch Memory	Use 32-bit longword addresses. Data is read or written in 32-bit longwords.	Use byte addresses. The memory unit ignores the low two address bits. Data is read or written in 32-bit longwords.
Local Memory	N/A	Use byte addresses. The memory unit ignores the low two address bits. Data is read or written in 32-bit longwords.

2.5.3.3 Signals

For each thread, the Intel® IXP1200 Network Processor supports one signal outstanding per functional unit—DRAM, SDRAM, FBI, or PCI. The Intel® IXP2400, IXP2800 and IXP2850 Network Processors support up to 15 signals outstanding per thread across all functional or coprocessor units. Each memory request must specify a signal such that multiple outstanding memory requests all have different signals. The macros and functions specify signal arguments that run on both the Intel® IXP1200 Network Processor and Intel® IXP2400, IXP2800 and IXP2850 Network Processors.

The use of signal arguments in Microengine C is specified in Section 2.6.13, “Signal Arguments and Usage.”

2.6 Defined Types, Enumerations, and Data Structures

This section describes common types, enumerations, and data structures used throughout the microengine Optimized Data Plane libraries.

2.6.1 cam_entry_t

The cam_entry_t data structure defines the data and state resulting from reading a content addressable memory (CAM) entry.

Microengine C Syntax

```
typedef struct cam_entry_t{
    uint32_t data: 32;
    uint32_t state: 4; };
```

Data Members

data	The data for a CAM entry.
state	The associated state for a CAM entry.

2.6.2 cam_lookup_t

The cam_lookup_t data structure defines the state, status, and index values resulting from a CAM match lookup.

Microengine C Syntax

```
typedef struct {
    union {
        struct {
            unsigned int zeros1         :20;
            unsigned int state          : 4;
            unsigned int hit            : 1;
            unsigned int entry_num      : 4;
            unsigned int zeros2         : 3;
        };
        unsigned int value;
    };
} cam_lookup_t;
```

Data Members

`zeros1`	All zeros.
`state`	The CAM entry state.
`hit`	The lookup result—a hit indicated by a value of one or miss indicated by a value of zero.
`entry_num`	The CAM entry number.
`zeros2`	All zeros.
`value`	The CAM value.

2.6.3 `cycle_t`

The `cycle_t` data structure is used to hold the value of the 64-bit chip cycle count register.

Note: The Intel® IXP1200 Network Processor hardware represents cycle count as little-endian 32-bit words—the least-significant 32 bits are in the left side of the register. The function `cycle64_read()` swaps the two 32-bit longwords and returns a long in correct order.

Microengine C Syntax

```
typedef unsigned long long cycle_t;
```

2.6.4 `hw_hash_t`

The `hw_hash_t` enumeration lists the sizes of indexes that can be translated in the hardware hash unit. See the description in the following manuals:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Also, see Section 4.3, “Hash Utilities.”

Microengine C Syntax

```
typedef enum{
HW_HASH_48 = 1,
HW_HASH_64 = 2,
HW_HASH_128 = 3,
}hw_hash_t;
```

Values

`HW_HASH_48`	A 48-bit hardware hash index.
`HW_HASH_64`	A 64-bit hardware hash index.
`HW_HASH_128`	A 128-bit hardware hash index.

2.6.5 `lm_handle_t`

The `lm_handle_t` enumeration lists the local memory handles used to address local memory. Currently there are two local memory handles, `LM_HANDLE_0` and `LM_HANDLE_1`.

Microengine C Syntax

```
typedef enum{
   LM_HANDLE_0,
   LM_HANDLE_1,
}lm_handle_t;
```

2.6.6 `trie_t`

The `trie_t` enumeration lists the types of trie lookups used in the longest prefix match algorithm—see the description in, Section 5.1.2.2.8, "`ipv4_route_lookup()`." The numeric suffix indicates the depth of the lookup tree, that is, the maximum number of SRAM latencies that are required in order to get the index of a route table.

Microengine C Syntax

```
typedef enum{
   TRIE5,
   TRIE15
}trie_t;
```

Values

`TRIE5`	Uses a dual lookup of a high-64K entry table and trie block lookups consuming a maximum of five SRAM reference signals.
`TRIE15`	Uses a lookup of the high-64K entry table and all 256 entry trie lookups consuming a maximum of fifteen SRAM latencies.

2.6.7 `mem_t`

The `mem_t` enumeration specifies the available memory types for operations that include a selection of the type of memory where data can be stored.

For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors, the choices are `SRAM`, `SCRATCH`, `RBUF`, `TBUF`, `DRAM`, `XFER`, `LOCALMEM`, `SRAM_RD_REG`, `SRAM_WR_REG`, `DRAM_RD_REG`, `DRAM_WR_REG`, and `GP_REG`.

For the Intel® IXP1200 Network Processor, the choices are `SCRATCH`, `SRAM`, `RBUF`, `TBUF`, and `DRAM`.

Microengine C Syntax

```
typedef enum{
    SRAM  = 1,
    SCRATCH  = 2,
    RBUF = 3,
    TBUF = 4,
    DRAM = 5,
    XFER = 6,
    LOCALMEM = 7,
    SRAM_RD_REG = 8,
    SRAM_WR_REG = 9,
    DRAM_RD_REG = 10,
    DRAM_WR_REG = 11,
    GP_REG = 12
} mem_t;
```

Values

`SRAM`	Static RAM memory.
`SCRATCH`	Scratch memory.
`RBUF`	A receive buffer.
`TBUF`	A transmit buffer.
`DRAM`	Dynamic RAM memory.
`XFER`	A transfer register pair.
`LOCALMEM`	Local memory (on the network processor chip).
`SRAM_RD_REG`	A static RAM read register.
`SRAM_WR_REG`	A static RAM write register.
`DRAM_RD_REG`	A dynamic RAM read register.
`DRAM_WR_REG`	A dynamic RAM write register.
`GP_REG`	A general-purpose register.

2.6.8 netif_elem_size_t

The `netif_elem_size_t` enumeration lists the allowable sizes of RBUF and TBUF elements. For further information, see the following manuals:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

The RBUF and TBUF sizes currently supported are 64-, 128-, and 256-bits.

Microengine C Syntax

```
typedef enum {
    NETIF_ELEM_64 = 0,
    NETIF_ELEM_128 = 1,
    NETIF_ELEM_256 = 2,
}netif_elem_size_t;
```

2.6.9 pool_t

The `pool_t` enumeration lists the allowable set of memory buffer pools supported by the `buf` interface. For the Intel® IXP1200 Network Processor, there can be up to eight pools. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors, there can be up to 64 pools.

Each buffer pool represents buffers of the same size. For example, one buffer pool holds 2-KB DRAM and 8-byte SRAM buffers while another buffer pool holds 256-byte DRAM and 4-byte SRAM buffers. The pool identifier and associated parameters can be specified in a `FREELIST_HANDLE`, as described in Section 3.2.1, “buf.”

Microengine C Syntax

```
typedef enum {
    POOL0 = 0,
    POOL1 = 1,
    POOL2 = 2,
    POOL3 = 3,
    POOL4 = 4,
    POOL5 = 5,
    POOL6 = 6,
    POOL7 = 7,
to
    POOL63 = 63
} pool_t;
```

2.6.10 queue_t

- The `queue_t` directive indicates an application’s choice of queue type when issuing memory references. See the definition of `queue_t` in the *Intel® IXP2400/2800 Network Processors Microengine C Compiler Language Support Reference Manual.*

The options used by this library include `queue_default`, `ordered`, `order_queue`, `any_queue`, `priority`, and `optimize_mem`. In addition, `no_option`, or three consecutive underscores, (`___`), may be used in place of `default_queue`.

If the chip version supports it, the queue option is honored, otherwise the `default_queue` option is substituted.

Each chip's hardware reference manual documents its supported options.

Note: `queue_t` is also used for the Intel® IXP1200 Network Processor intrinsics with the names matching those of queue options used in corresponding Intel® IXP1200 microengine instructions. See the Intel® IXP1200 Network Processor documentation for details.

For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors `queue_t` is defined as follows.

Microengine C Syntax

```
#define queue_t unsigned int
```

2.6.11 thread_model_t

The `thread_model_t` directive determines whether threads executing the function are local to a microengine or global to the chip.

If `LOCAL_THREADS` is specified, the code emitted by the compiler or assembler has the option to use registers local to the microengine. If `GLOBAL_THREADS` is specified, the emitted code may need to store information in SRAM or scratch memory. Following this directive, `msgq` operations keep the *msgq* index in a general-purpose register if `LOCAL_THREADS` is specified or in globally accessed memory such as scratch memory if `GLOBAL_THREADS` is specified.

Microengine C Syntax

```
typedef enum {
    LOCAL_THREADS,
    GLOBAL_THREADS,
}thread_model_t;
```

Values

`LOCAL_THREADS`	Indicates that the emitted code is used by a single microengine producer and consumer.
`GLOBAL_THREADS`	Indicates that the emitted code is used by multiple microengine producers and consumers.

2.6.12 mbox_t

The data structure `mbox_t` stores an `SRAM` or `SCRATCH` memory address or a thread number and address of an `XFER` remote transfer register.

Microengine C Syntax

```
typedef struct {
    union{
        __declspec(sram) void *sram_addr;
        __declspec(scratch) void *scratch_addr;
        struct
        {
            uint32_t THD_ID;
            volatile __declspec(remote sram_read_reg) uint32_t *rem_reg;
        }mbox_xfer;
    };
}mbox_t;
```

Data Members

`sram_addr`	A static RAM address.
`scratch_addr`	A scratch memory address.
`THD_ID`	A thread ID number.
`rem_reg`	A remote transfer register address.

2.6.13 Signal Arguments and Usage

In order to develop code that runs on the Intel® IXP2400, IXP2800 and IXP2850 Network Processors as well as future chip generations, the Microengine C library memory functions provide two signal-related arguments. The first argument defines the requested signal. The second argument defines the wakeup signals—those signals that must be received before the thread resumes execution.

Note: For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the Microengine Assembler provides a `.sig` directive and the compiler provides the `SIGNAL/SIGNAL_PAIR` types.

The following arguments are given for memory operations where signals are involved.

2.6.13.1 `REQ_SIG`

The requested signal.

Microengine Assembler Syntax

```
.sig req_sig
```

Microengine C

```
SIGNAL req_sig
```

2.6.13.2 IN_WAKEUP_SIGS

The wakeup signals.

The in_wakeup_sigs mask describes the signals that cause a thread to wakeup. If signals are specified the thread swaps out. The signals must all be received before the thread continues execution.

Microengine Assembler Example Usage

```
.sig my_sig1 my_sig2
//
//The first operation
sram_read($out_data, base_addr, offset, cnt, my_sig1, sig_none, ___);
//
//
//The second operation
sram_read($out_data, base_addr, offset, cnt, my_sig2, signals(my_sig1,\
   my_sig2), sig_none, ___);
```

Microengine C Example Usage

```
SIGNAL my_sig1, my_sig2;
uint32_t my_sig_mask;
my_sig_mask = __signals(&my_sig1, &my_sig2);
//
//The first operation
ixp_sram_read(&data0, addr0, ref_count, my_sig1, SIG_NONE, priority);
//
//
//The second operation
ixp_sram_read(&data1, addr1, ref_count, my_sig2, my_sig_mask, ___);
__implicit_read(&my_sig1);
__implicit_read(&my_sig2);
```

The REQ_SIG and in_wakeup_sigs parameters for the first operation are my_sig1 and SIG_NONE respectively.

The REQ_SIG and in_wakeup_sigs for the second operation is my_sig2 and signals(my_sig1, my_sig2).

The second operation will execute, the thread will swap out, and the thread will wakeup after my_sig1 and my_sig2 have been received.

Because in_wakeup_sigs are specified, the thread will swap out, allowing other threads on the microengine to run. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the thread is enabled allowing execution to resume when both signals are received.

Sequential reads of this kind are referred to as pipelined reads.

Note: For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors each reference requests a separate signal, the thread swaps out following the second reference and wakes up after both signals are received.

2.6.13.3 `SIGNAL`

The `SIGNAL` data type is used to specify the requested signal.

Microengine C Syntax

```
typedef __declspec(signal) int SIGNAL
```

2.6.13.4 `SIGNAL_PAIR`

The `SIGNAL_PAIR` data type is used for declaring hardware signal pair registers. This type of variable can be used with I/O instructions that generate dual signals upon completion.

Microengine C Syntax

```
Typedef __declspec(signal_pair) struct {int even; int odd;} SIGNAL_PAIR;
```

2.6.13.5 `SIGNAL_MASK`

The `SIGNAL_MASK` data type is used for masks specifying one or more signal registers.

Microengine C Syntax

```
typedef int SIGNAL_MASK;
```

2.6.13.6 `signal_op_t`

This enumeration defines the types of calculation that can be performed on multiple signals. The selection `ALL` means the operation requires all signals of a set of signals to be present. The selection `ANY` means the operation is performed if any one signal from a set of signals is present. See Section 3.2.11.1.3, "sig_wait()."

Microengine C Syntax

```
typedef enum{
    ANY,
    ALL,
}signal_op_t;
```

2.6.14 `buf_handle_t`

This data structure stores a longword SRAM address used by the `buf` functions.

Microengine C Syntax

```
typedef __declspec(packed) struct {
```

```
union {
        struct {
                unsigned int eop          : 1;
                unsigned int sop          : 1;
                unsigned int seg_count    : 6;
                unsigned int lw_offset    :24;
            };
        unsigned int value;
        };
} buf_handle_t;
```

Data Members

`eop`	The end-of-packet flag.
`sop`	The start-of-packet flag.
`seg_count`	The segment count.
`lw_offset`	The longword offset.
`value`	The aggregate value.

2.6.15 `atm_header_t`

This enumeration defines the ATM header type.

Microengine C Syntax

```
typedef enum {
    ATM_UNI = 0,
    ATM_NNI = 1
} atm_header_t;
```

2.6.16 `atm_uni_header_le_t`

This data structure defines the little endian UNI ATM cell header fields.

Microengine C Syntax

```
typedef __declspec(packed) struct {
    uint32_t clp :  1;
    uint32_t pti :  3;
    uint32_t vci : 16;
    uint32_t vpi :  8;
    uint32_t gfc :  4;
} atm_uni_header_le_t;
```

Data Members

`clp`	The cell loss priority.
`pti`	The payload type.
`vci`	The virtual channel identifier.
`vpi`	The virtual path identifier.
`gfc`	The generic flow control.

2.6.17 atm_nni_header_le_t

This data structure defines the little endian NNI ATM cell header fields.

Microengine C Syntax

```
typedef __declspec(packed)  struct {
    uint32_t clp : 1;
    uint32_t pti : 3;
    uint32_t vci : 16;
    uint32_t vpi : 12;
} atm_nni_header_le_t;
```

Data Members

`clp`	The cell loss priority.
`pti`	The payload type.
`vci`	The virtual channel identifier.
`vpi`	The virtual path identifier.

2.6.18 atm_uni_header_be_t

This data structure defines the big endian user-to-network (UNI) ATM cell header fields.

Microengine C Syntax

```
typedef __declspec(packed) struct {
    uint32_t gfc :  4;
    uint32_t vpi :  8;
    uint32_t vci : 16;
    uint32_t pti :  3;
    uint32_t clp :  1;
} atm_uni_header_be_t;
```

Data Members

gfc	The generic flow control.
vpi	The virtual path identifier.
vci	The virtual channel identifier.
pti	The payload type.
clp	The cell loss priority.

2.6.19 atm_nni_header_be_t

This data structure defines the big endian network-to-network (NNI) ATM cell header fields.

Microengine C Syntax

```
typedef __declspec(packed) struct {
    uint32_t vpi : 12;
    uint32_t vci : 16;
    uint32_t pti : 3;
    uint32_t clp : 1;
} atm_nni_header_be_t;
```

Data Members

vpi	The virtual path identifier.
vci	The virtual channel identifier.
pti	The payload type.
clp	The cell loss priority.

2.6.20 atm_cell_hdr_t

This data structure defines a union of all the types of ATM cell header.

Microengine C Syntax

```
typedef __declspec (packed) union {
    atm_uni_header_le_t atmUniHdrLe;
    atm_nni_header_le_t atmNniHdrLe;
    atm_uni_header_be_t atmUniHdrBe;
    atm_nni_hader_be_t atmNniHdrBe;
    uint32_t atmHeader;
} atm_cell_hdr_t;
```

2.6.21 `oam_cell_le_t`

This data structure defines the little endian Operation, Administration, and Maintenance (OAM) cell fields.

Microengine C Syntax

```
typedef __declspec (packed) struct {
    uint32_t  function_specific_first : 24;
    uint32_t  oam_function_type : 4;
    uint32_t  oam_cell_type : 4;
    uint32_t  function_specific[10];
    uint32_t  crc10 : 10;
    uint32_t  reserved : 6;
    uint32_t  function_specific_last : 16;
} oam_cell_le_t;
```

Data Members

`FunctionSpecificFirst`	The first three octets of the function specific field to use when aligning to a word boundary.
`OamFunctionType`	The OAM function type field.
`OamCellType`	The OAM cell type.
`FunctionSpecific`	The function specific field first 40 octets.
`Crc10`	The CRC-10 value for the OAM cell.
`Reserved`	Reserved bits—6 bits in extent.
`FunctionSpecificLast`	The last two octets of the function specific field to use when aligning to a word boundary.

2.6.22 `oam_cell_be_t`

This data structure defines the big endian OAM cell fields.

Microengine C Syntax

```
typedef __declspec (packed) struct {
    uint32_t  oam_cell_type : 4;
    uint32_t  oam_function_type : 4;
    uint32_t  function_specific_first : 24;
    uint32_t  function_specific[10];
    uint32_t  function_specific_last : 16;
    uint32_t  reserved : 6;
    uint32_t  crc10 : 10;
} oam_cell_be_t;
```

Data Members

`OamCellType`	The OAM cell type.
`OamFunctionType`	The OAM function type field.
`FunctionSpecificFirst`	The first three octet of function specific field to align to word boundary.
`FunctionSpecific[10]`	The function specific field first 40 octets.
`FunctionSpecificLast`	The last two octets of the function specific field to use when aligning to a word boundary.
`Reserved`	Reserved bits—6 bits in extent.
`Crc10`	The CRC-10 value for the OAM cell.

2.6.23 `oam_cell_t`

This data structure defines OAM cell type in all forms.

Microengine C Syntax

```
typedef __declspec(packed) union {
   oam_cell_le_t oam_cell_le;
   oam_cell_be_t oam_cell_be;
   xfers_t oam_cell;
   char oam_byte_cell[48];
}oam_cell_t;
```

Data Members

`OamCellLe`	OAM cell definition in little-endian mode.
`OamCellBe`	OAM cell definition in big-endian mode.
`OamCell`	An OAM cell—48 bytes in length—provided as a sequence of longwords.
`OamByteCell`	An OAM cell—48 bytes in length—provided as a byte array.

2.6.24 `five_tuple_t`

This data structure defines the five-tuple structure type.

Microengine C Syntax

```
typedef __declspec(packed) struct {
       uint32_t src_ip: 32;
```

```
        uint32_t int dest_ip: 32;
        uint32_t src_port: 16;
        uint32_t dest_port: 16;
        uint32_t protocol: 8;
    } five_tuple_t;
```

Data Members

`src_ip`	The source IP.
`dest_ip`	The destination IP.
`src_port`	The source port.
`dest_port`	The destination port.
`protocol`	The protocol.

2.6.25 `six_tuple_t`

This data structure defines the five tuple structure type.

Microengine C Syntax

```
typedef __declspec(packed) struct {
        uint32_t src_ip: 32;
        uint32_t dest_ip: 32;
        uint32_t src_port: 16;
        uint32_t dest_port: 16;
        uint32_t protocol: 8;
        uint32_t dscp: 6;
    } six_tuple_t;
```

Data Members

`src_ip`	The source IP.
`dest_ip`	The destination IP.
`src_port`	The source port.
`dest_port`	The destination port.
`protocol`	The protocol.
`dscp`	The DiffServ control point.

2.6.26 `n_tuple_t`

This data structure defines the n-tuple structure type.

Microengine C Syntax

```
typedef __declspec(packed) union {
    five_tuple_t    five_tuple;
    six_tuple_t     six_tuple;
}n_tuple_t
```

2.6.27 `tuple_t`

This enumeration defines the valid tuple structure types.

Microengine C Syntax

```
typedef enum {
    FIVE_TUPLE = 1,
    SIX_TUPLE = 2
}tuple_t;
```

Defined Values

`FIVE_TUPLE`	The tuple structure is a five-tuple.
`SIX_TUPLE`	The tuple structure is a six-tuple.

Hardware Abstraction Layer for the Microengine 3

The Hardware Abstraction Library (HAL) provides operating system like abstraction of hardware-assisted functions. HAL improves code portability by providing interfaces for common arithmetic and logic unit (ALU) operations (see Section 3.1, "Instruction Simplification Interface") and memory and thread management (see Section 3.2, "OS Emulation Interface").

3.1 Instruction Simplification Interface

Instruction simplification macros simplify common programming tasks and provide portability across instruction sets for the following network processors:

- Intel® IXP2400 Network Processor
- Intel® IXP2800 Network Processor
- Intel® IXP2850 Network Processor

3.1.1 `stdmac`

The `stdmac` macros are a set of frequently used standard macros and are microengine centric. These are only used in Microengine Assembler code blocks. In Microengine C code blocks, the standard C language expressions, arithmetic operators, conditionals, and so on are used instead.

The `stdmac` macros are the only category of macros that do not obey the general naming conventions described in Section 2.5.2.1.3, "Macro Names." Whereas the other interfaces all have operations that begin with the interface name (that is, DRAM macro names all start with `dram_`), the `stdmac` macros have a variety of names that are descriptive of their individual functions.

Note: The size estimates for `stdmac` macros reflect the extra instruction cycles needed to load constants.

Table 3-1. `stdmac` API

Name	Description
`add()`	Performs a 32-bit add where `in_a` is added to `in_b`.
`add_c()`	Performs a 32-bit add with carry where `in_a` is added to `in_b` which is then added to the previous carry.
`add_shf_left()`	Shifts `in_b` left then adds the result to `in_a`.
`add_shf_right()`	Shifts `in_b` right then adds the result to `in_a`.
`alu_op()`	Performs the ALU operation `op_spec` using `in_a` and `in_b`.
`alu_rot_left()`	Rotates `in_b` left by `in_shift_amt` bit positions, then performs operation `in_op_spec` using `in_a` and the shifted `in_b` value.

Table 3-1. `stdmac` API (Continued)

Name	Description
`alu_rot_right()`	Rotates `in_b` right by `in_shift_amt` bit positions, then performs operation `in_op_spec` using `in_a` and the shifted `in_b` value.
`alu_shf_left()`	Shifts `in_b` left by `in_shift_amt` bit positions, then performs operation `in_op_spec` using `in_a` and the shifted `in_b` value.
`alu_shf_right()`	Shifts `in_b` right by `in_shift_amt` bit positions, then performs the operation `in_op_spec` using `in_a` and the shifted `in_b` value.
`and_shf_left()`	Shifts `in_b` left then performs a logical and using `in_a` and the shifted `in_b` value.
`and_shf_right()`	Shifts `in_b` right then performs a logical and using `in_a` and the shifted `in_b` value.
`arith_shf_right()`	Arithmetic shifts `in_src` right by `shift_amt`, filling with sign bit at `sign_bit_pos`.
`array_index_from_ele_addr()` alias: `index_from_addr`	Converts the array element address `in_address` to the array index `out_index`.
`balr()`	A subroutine call that loads a thread's *next pc* to register `in_link_pc`, then branches to execute code at `target_label`.
`bitfield_extract()`	Extracts a bit field from a 32-bit register.
`bitfield_insert()`	Inserts a bit field into a 32-bit register.
`bits_clr()`	Clears the bits indicated by the mask `in_mask` at starting position `in_start_bit_pos`.
`bits_set()`	Sets the bits indicated by the mask `in_mask` at starting position `in_start_bit_pos`.
`divide()`	Divides `in_divisor` by `DIVIDEND` returning the result in `out_result`. The `DIVIDEND` must be a power of two.
`elem_addr_from_array_index()` alias: `addr_from_index`	Converts the array index `in_index` to the array element address `out_addr`.
`immed32()`	Loads `out_result` with a 32-bit constant.
`move()`	Copies `in_src` to `out_result`.
`multiply()`	Multiplies `in_a` by `in_b`. The result is 32 bits. The operands are 16 bits.
`multiply32()`	Multiplies `in_a` and `in_b` returning the result in `out_result`. The argument `out_result` is a 32-bit value. This instruction is optimized for `OPERAND_SIZE` specifications of `8x24` and `16x16`.
`multiply64()`	Multiplies `in_a` and `in_b` returning the result in `out_result`. The argument `ouput` is a 64-bit value. This instruction is optimized for `OPERAND_SIZE` specifications of `8x32`, `16x32`, and `32x32`.
`or_shf_left()`	Shifts `in_b` left then performs a logical `OR` using `in_a` and the shifted `in_b` value.
`or_shf_right()`	Shifts `in_b` right then performs a logical `OR` using `in_a` and the shifted `in_b` value.
`rand()`	Returns a pseudo-random number.
`rot_left()`	Rotates `in_src` left by `in_shift_amt` bit positions.

Table 3-1. `stdmac` API (Continued)

Name	Description
`rot_right()`	Rotates `in_src` right by `in_shift_amt` bit positions.
`shf_left()`	Shifts `in_src` left by `in_shift_amt` bit positions.
`shf_right()`	Shifts `in_src` right by `in_shift_amt` bit positions.
`srand()`	Sets the seed value for the pseudo-random number generator.
`sub()`	Subtracts `in_b` from `in_a`.
`sub_shf_left()`	Shifts `in_b` left then subtracts the result from `in_a`.
`sub_shf_right()`	Shifts `in_b` right then subtracts the result from `in_a`.
`arith_shf_right()`	Shifts `in_b` right by `in_shift_amt`, filling from the left with the value of the bit at `sign_bit_pos`.

3.1.1.1 Condition Codes

This section describes the status codes returned as the result of arithmetic and logic operations.

Condition Codes

`N`	Negative—the value of result bit 31 is one.
`Z`	Zero—the value of the result equals zero.
`V`	Sign Overflow—the value of the carry from bit 31 exclusive ored with the carry from bit 30.
`C`	Carry—the value of bit 31.

3.1.1.2 Common Arguments

This section describes arguments common to many `stdmac` operations.

Output

`out_result`	The result of an arithmetic and logic operation stored in a general-purpose register, a write transfer register, or a next neighbor register. **NOTE:** In the Intel® IXP1200 Network Processor a next neighbor register is *not* supported.

Input

`in_a`	The left source operand of an arithmetic and logic operation. May be a constant of up to 32 bits, a general-purpose register, a read transfer register, or a next neighbor register. **NOTE:** In the Intel® IXP1200 Network Processor a next neighbor register is *not* supported.
`in_b`	The right source operand of an arithmetic and logic operation. May be a constant of up to 32 bits, a general-purpose register, a read transfer register, or a next neighbor register. **NOTE:** In the Intel® IXP1200 Network Processor a next neighbor register is *not* supported.
`in_shift_amt`	The number of bit positions to shift `in_b` before performing the operation. Or, the number of bit positions to shift `in_src`. May be a constant where the low 5 bits are used, a general-purpose register, a read transfer register, or a next neighbor register. **NOTE:** In the Intel® IXP1200 Network Processor a next neighbor register is *not* supported.
`in_src`	The source operand of a unary arithmetic and logic operation. May be a constant of up to 32 bits, a general-purpose register, a read transfer register, or a next neighbor register. **NOTE:** In the Intel® IXP1200 Network Processor a next neighbor register is *not* supported.

3.1.1.3 API Functions

3.1.1.3.1 add()

Adds in_a to in_b.

Sets the condition code N if out_result bit 31 is one. Sets the condition code Z if out_result equals zero. Sets the condition code V if sign overflow occurs—that is, if a carry from bit 31 exclusive ored with the carry from bit 30 is true. Sets the condition code C if carry out from bit 31.

Microengine Assembler Syntax

```
add (out_result, in_a, in_b);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Syntax

```
.reg output
add (output, 0x1234, 0x12345678); // output = 0x123468AC
```

3.1.1.3.2 `add_c()`

Adds `in_a` to `in_b` then adds the result to `C`, the carry value.

Sets the condition code `N` if bit 31 of `out_result` equals one. Sets the condition code `Z` if `out_result` equals zero. Sets the condition code `V` if sign overflow occurs—that is, if a carry from bit 31 exclusive ored with the carry from bit 30 is true. Sets the condition code C if a carry out from bit 31 occurs.

Microengine Assembler Syntax

```
add _c(out_result, in_a, in_b);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg output source_a
move (source_a, 0x1);
add (output, 0xFFFFFFFF, 0x1); // output = 0
add_c (output, source_a, 0x1); // output = 3
```

3.1.1.3.3 `add_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with zeros. Adds the shifted result to `in_a`. Sets the condition code `N` if `out_result` bit 31 equals one. Sets the condition code `Z` if `out_result` equals zero. Sets the condition code `V` if sign overflow occurs—that is, if a carry from bit 31 exclusive ored with the carry from bit 30 is true. Sets the condition code `C` if a carry out from bit 31 occurs.

Microengine Assembler Syntax

```
add_shf_right (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg output
add_shf_right (output, 0xABCD1234, 0x12345678, 3); // output = 0xAE139D03
```

3.1.1.3.4 `add_shf_left()`

Shifts `in_b` left by `in_shift_amt`, filling from the right with zeros. Adds the shifted result to `in_a`. Sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Sets the condition code `V` if sign overflow occurs—that is, if a carry from bit 31 exclusive ored with the carry from bit 30 is true. Set condition code `C` if carry out from bit 31.

Microengine Assembler Syntax

```
add_shf_left (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg output
add_shf_left (output, 0xABCD1234, 0x12345678, 3); // 0x3D6FC5F4 with a carry.
```

3.1.1.3.5 alu_op()

Performs the operation specified by op_spec using in_a and in_b.

If op_spec is any of {B, ~B, AND, ~AND, AND~, OR}, sets condition code N if bit 31 of out_result equals one. Sets condition code Z if out_result equals zero. Clears condition code V and condition code C.

If op_spec is any of {+, +16, +4, +8, +carry, -, B-A, XOR, -carry}, sets condition code N if bit 31 of out_result equals one. Sets condition code Z if out_result equals zero. Sets condition code V if a sign overflow occurs—that is, bit 31 is exclusive ored with the carry from bit 30. Sets condition code C if a carry out from bit 31 occurs.

op_spec Definitions

+	Adds in_a and in_b.
+16	Masks in_b with 0xFFFF and adds the result to in_a.
+4	Masks in_b with 0xF and adds the result to in_a.
+8	Masks in_b with 0xFF and adds the result to in_a.
+carry	Adds in_a and in_b and adds the result to C, the condition code from the previous carry.
-	Subtracts in_b from in_a.
B-A	Subtracts in_a from in_b.
B	Returns in_b through out_result.
~B	Returns the logical inverse of in_b through out_result.
AND	Performs a logical bit-wise and of in_a with in_b.
~AND	Logically inverts in_a then performs a logical bit-wise and using the result and in_b.
AND~	Logically inverts in_b then performs a logical bit-wise and using the result and in_a.
OR	Returns the logical bit-wise or of in_a with in_b.
XOR	Returns the logical bit-wise exclusive or of in_a with in_b.
-carry	Adds in_a and in_b and subtracts C, the condition code for the previous carry. **NOTE:** This operation is supported by the Intel® IXP2400, IXP2800 and IXP2850 Network Processors only.

Microengine Assembler Syntax

```
alu_op (out_result, in_a, op_spec, in_b);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions—six if `op_spec` is `+4`.

See section Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg out_result in_a in_b
move(in_a, 0x100);
move(in_b, 0x200);
alu_op (out_result, in_a, +, in_b);
    // out_result is 0x300
alu_op (out_result, 0x22, +4, 0xf3);
    // out_result = 0x25
```

3.1.1.3.6 `alu_rot_right()`

Rotates `in_b` right by `in_shift_amt`, filling from the left with `in_b`. Performs `op_spec` using `in_a` and the shifted result.

If `op_spec` is any of {`B`, `~B`, `AND`, `~AND`, `AND~`, `OR`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

If `op_spec` is any of {`+`, `+16`, `+4`, `+8`, `+carry`, `-`, `B-A`, `XOR`, `-carry`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if sign overflow occurs—that is, if a carry from bit 31 exclusive ored with the carry from bit 30 is true. Sets condition code `C` if carry out from bit 31. See Section 3.1.1.3.5, "alu_op()" for description of `op_specs`.

Microengine Assembler Syntax

```
alu_rot_right (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to three instructions—four if `op_spec` is `+4`.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 2);
move(input, 0x12345678);
alu_rot_right (output, input2, +, input, 4);
    // output = 0x81234569
```

3.1.1.3.7 `alu_rot_left()`

Rotates `in_b` left by `in_shift_amt`, filling from the right with `in_b`. Performs `op_spec` using `in_a` and the shifted result.

If `op_spec` is any of {`B`, `~B`, `AND`, `~AND`, `AND~`, `OR`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

If `op_spec` is any of {`+`, `+16`, `+4`, `+8`, `+carry`, `-`, `B-A`, `XOR`, `-carry`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if sign overflow occurs—overflow is the carry from bit 31 exclusive ored with carry from bit 30. Sets condition code `C` if carry out from bit 31.

Microengine Assembler Syntax

```
alu_rot_left (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions. See Section 3.1.1.3.5, "alu_op()" for a description of `op_spec` values.

Estimated Size

One to three instructions—four if `op_spec` is `+4`.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 5);
move(input, 0xFF);
alu_rot_left (output, input2, +, input, 4);
    // output = 0xFF5
```

3.1.1.3.8 `alu_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with zeros then performs `op_spec` using `in_a` and the shifted result.

If `op_spec` is one of {`B`, `~B`, `AND`, `~AND`, `AND~`, `OR`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

If `op_spec` is one of {`+`, `+16`, `+4`, `+8`, `+carry`, `-`, `B-A`, `XOR`, `-carry`}, sets condition code `N` if bit 31 of `out_result` equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if a sign overflow occurs—carry from bit 31 exclusive ored with carry from bit 30. Sets condition code `C` if carry out from bit 31.

Microengine Assembler Syntax

```
alu_shf_right (out_result, in_a, op_spec, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions. See Section 3.1.1.3.5, "alu_op()" for a description of `op_spec` values.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 5);
move(input, 0x12345678);
alu_shf_right (output, input2, +, input, 3);
    // output = 0x91a2b3c5
```

3.1.1.3.9 alu_shf_left()

Shifts in_b left by in_shift_amt, filling from the right with zeros then performs op_spec using in_a and the shifted result.

If op_spec is one of {B, ~B, AND, ~AND, AND~, OR}, sets condition code N if bit 31 of out_result equals one. Sets condition code Z if out_result equals zero. Clears condition code V and condition code C.

If op_spec is one of {+, +16, +4, +8, +carry, -, B-A, XOR, -carry}, sets condition code N if bit 31 of out_result equals one. Sets condition code Z if out_result equals zero. Sets condition code V if sign overflow occurs—carry from bit 31 exclusive ored with carry from bit 30. Sets condition code C if carry out from bit 31.

Microengine Assembler Syntax

```
alu_shf_left (out_result, in_a, op_spec, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions. See Section 3.1.1.3.5, "alu_op()" for a description of op_spec values.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 5);
move(input, 0x12345678);
alu_shf_left (output, input2, +, input, 3);
    // output = 0x91a2b3c5
```

3.1.1.3.10 `and_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with zeros then performs a logical and of the shifted result and `in_a` placing the result in `out_result`. Sets condition code `N` if `out_result` bit 31 is one. Sets condition code `Z` if `out_result` is zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
and_shf_right (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 5);
move(input, 0x12345678);
and_shf_right (output, input2, input, 3);
    // output = 5
```

3.1.1.3.11 `and_shf_left()`

Shifts `in_b` left by `in_shift_amt`, filling from the right with zeros then performs a logical and using the shifted result and `in_a`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
and_shf_left (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg output input2 input
move(input2, 5);
move(input, 0x12345678);
and_shf_left (output, input2, input, 3);
    // output = 0
```

3.1.1.3.12 `array_index_from_ele_addr()`

Convert array element address `in_addr` to array index `out_index` using the equation:

```
out_index = (in_addr - BASE_ADDRESS) / ELEMENT_SIZE
```

Microengine Assembler Syntax

```
array_index_from_ele_addr(out_index, in_addr, BASE_ADDRESS,\
    ELEMENT_SIZE);

index_from_addr(out_index, in_addr, BASE_ADDRESS, ELEMENT_SIZE);
```

Output

`out_index`	The index of the array element.

Input

`in_addr`	The address of the array element.
`BASE_ADDRESS`	The memory address of the first element in the array.
`ELEMENT_SIZE`	The size of the array element in address units.

Estimated Size

Typically three instructions—two immediate instructions plus one ALU instruction.

Microengine Assembler Example Usage

```
.reg test_result, in_addr
immed32(in_addr, 0x24294)
array_index_from_elem_addr(test_result, in_addr, 0xb1e4, 0x4)
    // test_result = 0x642c
```

3.1.1.3.13 `balr()`

A subroutine call to load the thread's *next pc* to register `in_link_pc`, then branch to execute code at `target_label`. At the end of the subroutine code segment, an `RTN` instruction is used to resume execution at the instruction indicated by `in_link_pc`.

Note: `link_pc` must be a register of global scope in order for the subroutine to correctly return to it.

Microengine Assembler Syntax

```
balr (in_link_pc, target_label);
```

Input

`in_link_pc`	The register to hold the program counter of the calling routine.
`target_label`	The label where the target code resides.

Estimated Size

Two instructions.

Microengine Assembler Example Usage

```
balr(link_addr, subroutine_a);
// continue from here after RTN
// other code
subroutine_a#:
// subroutine code
RTN[link_addr]
```

3.1.1.3.14 `bitfield_extract()`

Extracts a bit field from a 32-bit register. It is useful when packing fields into a limited set of registers. The bits are numbered 31 to zero, left to right.

Microengine Assembler Syntax

```
bitfield_extract (out_result, in_src, MSB, LSB);
```

Output

`out_result`	The extracted field, right justified.

Input

`in_src`	The register containing the field.
`MSB`	A constant first bit specifying a range of bits.
`LSB`	A constant last bit specifying a range of bits.

Estimated Size

One or two instructions.

Microengine Assembler Example Usage

```
#define FIELD1 31, 16
#define FIELD2 15, 12
#define FIELD3 11, 0
immed32(src_reg, 0x01234567);
bitfield_extract(result, src_reg, FIELD3); // result = 0x567
```

3.1.1.3.15 `bitfield_insert()`

Inserts a specified bit field into a 32-bit register. It is useful when packing fields into a limited set of registers. The bits are numbered 31 to zero, left to right.

Microengine Assembler Syntax

```
bitfield_insert (out_result, in_a, in_b, MSB, LSB);
```

:

Input

`in_a`	The register where the field is to be inserted—must be a general-purpose register.
`in_b`	The constant or register with the field to be inserted.
`MSB`	The constant first bit in a range of bits.
`LSB`	The constant last bit in a range of bits.

Output

`out_result`	The extracted field, right justified.

Estimated Size

Two or three instructions.

Microengine Assembler Example Usage

```
#define FIELD1 31, 16
#define FIELD2 15, 12
#define FIELD3 11, 0
immed32(src_reg, 0x01234567);
immed32(field, 0xf);
bitfield_insert(result, src_reg, field, FIELD2); // result = 0x0123f567
```

3.1.1.3.16 `bits_clr()`

Clears the bits indicated by the mask `in_mask` at starting position `in_start_bit_pos`. Logically shifts the mask left filling from the right with zeros, logically inverts the shifted mask, then logically ands the results with `io_data`.

Microengine Assembler Syntax

```
bits_clr (io_data, in_start_bit_pos, in_mask);
```

Input

`in_start_bit_pos`	The right-most bit of a field of bits to clear.
`in_mask`	The mask of bits to clear where a one in any bit position specifies clear.

Input/Output

`io_data`	The index of the array element.

Estimated Size

Three to five instructions.

Microengine Assembler Example Usage

```
.reg test_result, x, y
immed32(test_result,0xffffffff);
bits_clr(test_result, 31, 1);
    // now test_result = 0x7fffffff
```

3.1.1.3.17 `bits_set()`

Bits are set as indicated by mask `in_mask` and starting position `in_start_bit_pos`. The instruction logically shifts the mask left, filling from the right with zeros, then logically ors the result with `io_data`.

Microengine Assembler Syntax

```
bits_clr (io_data, in_start_bit_pos, in_mask);
```

Input

`in_start_bit_pos`	The right-most bit of the field of bits to set.
`in_mask`	The mask indicating the bits to clear. A one in any bit position is a request to set that bit.

Input/Output

`io_data`	The index of the array element.

Estimated Size

Three to five instructions.

Microengine Assembler Example Usage

```
.reg test_result
immed32(test_result,0x12345678);
bits_set(test_result, 31, 1);
    //test_result = 0x92345678
```

3.1.1.3.18 `divide()`

Returns the value of one parameter divided by the second as in the following assignment statement:

```
out_result =  dividend/DIVISOR
```

The value of `DIVISOR` must be a power of two.

Microengine Assembler Syntax

```
divide(out_result, in_dividend, DIVISOR);
```

Input

`in_dividend`	A register or constant containing the numerator.
`DIVISOR`	A constant. For a register based `in_divisor`, the `DIVIDEND` must be a power of two. For a constant `in_divisor`, `DIVIDEND` can be any constant.

Output

`out_result`	The result of the divide operation.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
.reg  test_result, divisor

immed32(divisor, 0xabdecdec);
divide(test_result, divisor, 128);
// test_result = 0x157bd9b
```

3.1.1.3.19 `elem_addr_from_array_index()`

Converts array index `in_index` to array element address `out_addr`.

Microengine Assembler Syntax

```
elem_addr_from_array_index (out_addr, in_index, BASE_ADDRESS, ELEMENT_SIZE);

addr_from_index (out_addr, in_index, BASE_ADDRESS, ELEMENT_SIZE);
```

Input

`in_index`	Index of the array element.
`BASE_ADDRESS`	Memory address of the first element in the array.
`ELEMENT_SIZE`	Size of the array element in address units.

Output

`out_addr`	The array element's address.

Estimated Size

Typically three instructions—two immediate instructions plus one ALU instruction.

The worst case is 21 instructions—four immediate instructions plus 17 ALU instructions.

Microengine Assembler Example Usage

```
.reg test_result
//
elem_addr_from_array_index(test_result, 0x85, 0xcafeefac, 0x40)
    // test_result = 0xcaff10ec
```

3.1.1.3.20 `immed32()`

Loads a constant into a general-purpose register.

Microengine Assembler Syntax

```
immed32 (out_result, VAL);
```

Input

`VAL`	A constant whose width is up to 32 bits.

Output

`out_result`	The register into which the constant is loaded.

Estimated Size

One or two instructions—three instructions if output is to a transfer register.

Microengine Assembler Example Usage

```
.reg result1
immed32(result1, 0x12345678)
    // result1 = 0x12345678
```

3.1.1.3.21 `move()`

Copies the contents of one register to another register.

Microengine Assembler Syntax

```
move (out_result, in_src);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One or two instructions—three instructions if output is to a transfer register.

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x45645677)
move(reg2, reg1)         // reg2 = 0x45645677
```

3.1.1.3.22 `multiply()`

Multiplies `in_a` by `in_b`. The result is 32 bits.

For Intel® IXP2400, IXP2800 and IXP2850 Network Processors the operation assumes 16-bit operands.

Note: See `multiply32()` and `multiply64()` for other operand sizes.

Microengine Assembler Syntax

```
multiply (out_result, in_a, in_b);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

Four instructions.

Microengine Assembler Example Usage

```
.reg test_result, in_b
//
immed32(in_b, 0xad);
multiply(test_result, in_b, 0xef);     // test_result = 0xa183;
```

3.1.1.3.23 `or_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with zeros. The result is or-shifted with `in_a`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
or_shf_right (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
immed32(reg2, 0x11000000)
```

```
or_shf_right(reg1, reg2, 8)     // reg1 = 0x12355678
```

3.1.1.3.24 or_shf_left()

Shifts `in_b` left by `in_shift_amt`, filling from the right with zeros. The result is or-shifted with `in_a`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
or_shf_left (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
immed32(reg2, 0x00110000)
or_shf_left(reg1, reg2, 8)      // reg1 = 0x13345678
```

3.1.1.3.25 rot_right()

Rotates `in_src` right by `in_shift_amt`, filling from the left with `in_b`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
rot_right (out_result, in_src, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
rot_right(reg2, reg1, 8)     // reg2 = 0x78123456
```

3.1.1.3.26 `rot_left()`

Rotates `in_src` left by `in_shift_amt`, filling from the right with `in_b`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
rot_left (out_result, in_src, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One instruction.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
rot_left(reg2, reg1, 8)         // reg2 = 0x34567812
```

3.1.1.3.27 `shf_right()`

Shifts `in_src` right by `in_shift_amt`, filling from the left with zeros. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
shf_right (out_result, in_src, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One or two instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
shf_right(reg2, reg1, 8)        // reg2 = 0x123456
```

3.1.1.3.28 `shf_left()`

Shifts `in_src` left by `in_shift_amt`, filling from the right with zeros. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Clears condition code `V` and condition code `C`.

Microengine Assembler Syntax

```
shf_left (out_result, in_src, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg reg1 reg2
immed32(reg1, 0x12345678)
shf_left(reg2, reg1, 8)         // reg2 = 0x34567800
```

3.1.1.3.29 `sub()`

Subtracts `in_b` from `in_a`. This is equivalent to the expression:

```
out_result = in_b - in_a
```

Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if sign overflow occurs—that is, if the carry from bit 31 exclusive ored with the carry from bit 30 is true. Sets condition code `C` if carry out from bit 31.

Microengine Assembler Syntax

```
sub (out_result, in_a, in_b);
```

Inputs and Outputs

See Section 3.1.1.2, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg reg1 reg2 out_result
immed32(reg1, 0x12345678)
immed32(reg2, 0x12340000)
sub (out_result, reg1, reg2)        // out_result = 0x5678
```

3.1.1.3.30 `sub_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with zeros then subtracts the shifted result from `in_a`. Sets condition code `N` if `out_result` bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if a sign overflow occurs—carry from bit 31 exclusive ored with carry from bit 30. Sets condition code `C` if a carry out from bit 31 occurs.

Microengine Assembler Syntax

```
sub_shf_right (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage

```
.reg reg1 reg2 out_result
immed32(reg1, 0x12345678)
immed32(reg2, 0x12340000)
sub_shf_right(out_result, reg1, reg2, 4)      // out_result = 11111678
```

3.1.1.3.31 `sub_shf_left()`

Shifts `in_b` left by `in_shift_amt`, filling from the right with zeros then subtracts the shifted result from `in_a`. Sets condition code `N` if out_result bit 31 equals one. Sets condition code `Z` if `out_result` equals zero. Sets condition code `V` if a sign overflow occurs—carry from bit 31 exclusive ored with carry from bit 30. Sets condition code `C` if a carry out from bit 31 occurs.

Microengine Assembler Syntax

```
sub_shf_left (out_result, in_a, in_b, in_shift_amt);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

See Section 2.5.2.1.1, "Constants as Arguments."

Microengine Assembler Example Usage

```
.reg reg1 reg2 out_result
immed32(reg1, 0x12345678)
immed32(reg2, 0x01234000)
sub_shf_left(out_result, reg1, reg2, 4)          // out_result = 5678
```

3.1.1.3.32 `arith_shf_right()`

Shifts `in_b` right by `in_shift_amt`, filling from the left with the value of the bit at `sign_bit_pos`. Condition codes are not affected.

This operation can also be used to `sign_extend` a value. There are three cases:

1. If `SIGN_BIT_POS` is set to 31, no sign extend occurs, only the arithmetic shift right.
2. If `in_shift_amt` is a constant whose value is zero, only sign extend occurs.
3. If both `in_shift_amt` is non-zero and `SIGN_BIT_POS` is not 31, the sign is extended from `SIGN_BIT_POS` left to bit 31. Then the result is shifted right, filling with the sign bit.

Microengine Assembler Syntax

```
arith_shf_right (out_result, in_src, shift_amt, sign_bit_pos);
```

Input

`in_src`	The source operand to be shifted.
`in_shift_amt`	The number of bit positions to shift `in_src`.
`SIGN_BIT_POS`	A constant representing the location of the sign bit within `in_src`.

Output

`out_result`	The result of the arithmetic shift.

Estimated Size

Two to eight instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg reg1 out_result
immed32(reg1, 0xFF345678)
arith_shf_right(out_result, reg1, 4, 31)
    // out_result = 0xFFF34567
```

3.1.1.3.33 `multiply32()`

Multiply `in_a` with `in_b` producing a 32 bit result.

Microengine Assembler Syntax

```
multiply32(out_result, in_a, in_b, OPERAND_SIZE);
```

Input

`in_a`	The left source operand for the arithmetic and logic operation.
`in_b`	The right source operand for the arithmetic and logic operation.
`OPERAND_SIZE`	Indicates the size of the operands: • `8x24`—`in_a` is 8 bits, `in_b` is 24 bits • `16x16`—`in_a` is 16 bits, `in_b` is 16 bits

Output

`out_result`	The result of the arithmetic and logic operation.

Estimated Size

Three instructions if `OPERAND_SIZE` is `8x24` or four instructions if `OPERAND_SIZE` is `16x16`.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg test_result, in_a, in_b
//
immed32(in_a, 0xd7f7);
immed32(in_b, 0x8e8f);
multiply32(test_result, in_a, in_b, OP_SIZE_16X16);
    // test_result = 0x7843a4f9
```

3.1.1.3.34 `multiply64()`

Multiply `in_a` and `in_b` producing a 64 bit result.

Microengine Assembler Syntax

```
multiply64(out_result_hi, out_result_lo, in_a, in_b, OPERAND_SIZE);
```

Input

`in_a`	The left source operand of a multiplication.
`in_b`	The right source operand of a multiplication.
`OPERAND_SIZE`	Indicates the size of operands: • `16x32`—`in_a` is 16 bits, `in_b` is 32 bits • `32x32`—`in_a` is 32 bits, `in_b` is 32 bits

Output

`out_result_hi`	The most significant 32 bits of the result.
`out_result_lo`	The least significant 32 bits of the result.

Estimated Size

Seven instructions.

See Section 2.5.2.1.1, “Constants as Arguments.”

Microengine Assembler Example Usage

```
.reg expected, test_result, in_a, in_b, out_result_hi, out_result_lo
immed32(in_a, 0xffffffff);
immed32(in_b, 0xffffffff);
//
multiply64(out_result_hi, out_result_lo, in_a, in_b, OP_SIZE_32X32);
    // out_result_hi = 0xfffffffe, out_result_lo = 0x00000001
```

3.1.1.3.35 `rand()`

Returns a pseudo-random number.

Microengine Assembler Syntax

```
rand (out_result);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

Two instructions.

Microengine Assembler Example Usage

```
.reg      out_result
rand(out_result)      // out_result will contain the pseudo-random number
```

3.1.1.3.36 `srand()`

Sets the seed for the pseudo-random number generator.

Microengine Assembler Syntax

```
srand(in_src);
```

Inputs and Outputs

See Section 3.1.1.2, "Common Arguments" for argument descriptions.

Estimated Size

Three to five instructions.

Microengine Assembler Example Usage

```
.reg      in_seed
move(in_seed, 0x1234)
srand(in_seed)
```

3.2 OS Emulation Interface

These operations emulate features that are often provided to applications from an operating system. There is no operating system for the microengine threads. However, the operations in this section offer a simple interface to hardware features similar to that provided by an operating system. These operations provide centralized functionality such as buffer management (Section 3.2.1, “buf”), common code critical section management (Section 3.2.3, “critsect”), cycle count (Section 3.2.4, “cycle”), concurrent memory management (Section 3.2.5, “dram” , Section 3.2.9, “rbuf” , Section 3.2.10, “scratch” , Section 3.2.12, “sram” , and Section 3.2.13, “tbuf”), interthread signaling (Section 3.2.11, “sig”), and messaging (Section 3.2.7, “mailbox” and Section 3.2.8, “msgq”).

3.2.1 `buf`

The `buf` interface is used to allocate and free memory buffers from defined pools. A pool of memory buffers is a set of same-size memory buffers.

Note: In the Intel® IXP2400, IXP2800 and IXP2850 Network Processors hardware-supported SRAM queues are used to access pools of memory buffers—refer to one of the following manuals:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

There are two ways to use these operations:

1. As SRAM buffer descriptors for DRAM data buffers

   A buffer descriptor is used to hold condensed information about a buffer of DRAM data, for example a packet buffer. Buffer descriptors are stored in an array in SRAM. Data buffers live in a parallel array in DRAM. An SRAM descriptor at a given index position refers to a DRAM data buffer at the same index. A free DRAM data buffer is allocated through the allocation of the associated SRAM buffer descriptor. Access to the SRAM buffer descriptor free pool is through `buf_alloc()` and `buf_free()`.
2. As SRAM data buffers

   The data is held directly in the SRAM buffers, and the relationship with corresponding DRAM buffers is ignored. Here just the SRAM addresses are used—`d_base`, `d_size`—while the `buf_dram` operations are not used. SRAM and DRAM addresses may be stored as indexes to save space. Operations to convert between indexes and addresses are provided to facilitate this practice.

A freelist handle is defined that can be used in these operations as follows:

```
#define FREELIST_HANDLE POOL_ID, d_base, d_size, s_base, s_size
```

Where:

- `POOL_ID`—the identifier of a buffer pool
- `d_base`—the base address of a region of DRAM buffers
- `d_size`—the size of a DRAM buffer
- `s_base`—the base address of a region of SRAM buffers
- `s_size`—the size of an SRAM buffer

Not all parameters of the `FREELIST_HANDLE` are necessarily used by all of the `buf` operations. However, for consistency the same `FREELIST_HANDLE` definition is used for all.

Table 3-2. `buf` API

Name	Description
`buf_alloc()`	Returns a buffer address from the freelist.
`buf_dram_addr_from_index()`	Calculates a DRAM address from an index.
`buf_dram_addr_from_sram_addr()`	Calculates the address of a DRAM buffer from an SRAM address.
`buf_free()`	Frees a memory buffer and pushes it back to the freelist.
`buf_freelist_create()`	Allocates a DRAM memory buffer.
`buf_index_from_dram_addr()`	Calculates an index from a DRAM address.
`buf_index_from_sram_addr()`	Calculates an index from an SRAM address.
`buf_sram_addr_from_dram_addr()`	Calculates an SRAM address from the address of a DRAM buffer.
`buf_sram_addr_from_index()`	Calculates an SRAM address from an index.

3.2.1.1 Common Arguments

The following arguments are used throughout `buf` operations:

Input

`FREELIST_HANDLE`	The handle to a freelist which includes the parameters `POOL_ID`, `d_base`, `d_size`, `s_base`, and `s_size`.
`POOL_ID`	The ID of a particular pool, such as `POOL0`, `POOL1`, `POOL2`, and so on. See Section 2.6.9, “pool_t.”
`d_base`	Base address of the DRAM buffers. The buffer must be aligned on a 32-byte boundary.(1) If `ADDRESS_MODE` is `IXP2XXX`, then `d_base` is a byte address.
`d_size`	The size of DRAM buffers must be a power of two.(2) The buffer must be aligned on a 32-byte boundary.(1) If `ADDRESS_MODE` is `IXP2XXX`, then `d_size` is the number of bytes.
`s_base`	Base address of SRAM buffers. The buffer must be aligned on a 4-byte boundary. If `ADDRESS_MODE` is `IXP2XXX`, then `s_base` is a byte address.
`s_size`	The size of SRAM buffers must be a power of two.(2) The buffer must be aligned on a 4-byte boundary. If `ADDRESS_MODE` is `IXP2XXX`, then `s_size` is the number of bytes.

1. For efficient operation of DRAM buffer access for all possible DRAM chip types, `d_base` and `d_size` should be aligned to a 32-byte boundary. This is to prevent the buffer from crossing channels to more two separate DRAM chips, resulting in additional memory cycle overhead. Refer one of the following manuals for more information:
   -IXP 2400 Hardware Reference Manual
   -IXP 2800 Hardware Reference Manual
   -
2. To simplify address conversions, `d_size` and `s_size` must be a power of two. This is to avoid lengthy division, where a simple shift and add suffices.

3.2.1.2 API Functions

3.2.1.2.1 `buf_alloc()`

Allocates SRAM or DRAM buffers from a buffer freelist identified by `FREELIST_HANDLE`.

Microengine Assembler Syntax

```
buf_alloc(out_sram_addr, POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE, \
    REQ_SIG, in_wakeup_sigs, Q_OPTIONS);
```

Microengine C Syntax

```
INLINE void ixp_buf_alloc(
    __declspec(sram) buf_handle_t *out_sram_addr,
    pool_t POOL_ID,
    uint32_t d_base,
    uint32_t d_size,
    uint32_t s_base,
    uint32_t s_size,
    Signal * REQ_SIG,
    uint32_t in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, "Common Arguments" for `FREELIST_HANDLE` argument descriptions.

`REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
`Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Output

`out_sram_addr`	The allocated buffer handle that contains an SRAM address. Note that for Microengine C this argument is typed by a data structure, for Microengine Assembler the data returned is in the same format as defined in `buf_handle_t`. The return address is in longword units. If `ADDRESS_MODE` is `IXP2XXX`, the result is a byte address.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
#define FREELIST_HANDLE 0, 0, 64, 512, 64
alu[i, --, B, 3]
buf_freelist_create(i, FREELIST_HANDLE);
//
alu[i, --, B, 0]
// get the next free address from the free list
buf_alloc($out_addr, FREELIST_HANDLE, sig1, sig1, ___);
// $out_addr contains the free list address
// ...
// put this address back in to freelist
buf_free($out_addr, FREELIST_HANDLE);
```

Microengine C Example Usage One

```
// define pool id, sram and dram buffer bases and sizes
// buffer pool id is POOL0
// dram buffer base address is 100 and dram buffer size is
// 256 quadword(2KB)
// sram buffer base address is 500 and sram buffer is 4 32 bit words(16B).
#define FREELIST_HANDLE POOL0,128,256,512,4
//
__declspec(sram_read_reg) uint32_t descriptor_address;
// allocate a buffer, request a signal, wakeup on the same signal
SIGNAL sig1;
buf_handle_t descriptor_address;
Mask = __signals(&sig1)
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, sig1, mask, ___);
__implicit_read(&sig1); // required to mark the sig1 as read
// descriptor_address.lw_offset contains the longword sram address
// some code, e.g., use the buffer
// decide buffer is no longer needed
ixp_buf_free((void *)descriptor_address, FREELIST_HANDLE);
```

Microengine C Example Usage Two

```
// use byte addressing for ADDRESS_MODE IXP2XXX
#define FREELIST_HANDLE POOL0,512,2048,2048,16
__declspec(sram_read_reg) uint32_t descriptor_address;
SIGNAL sig1;
buf_handle_t descriptor_address;
// asynchronous allocation
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, &sig1, SIG_NONE, ___);
// some other code
wait_for_all (&sig1);// swap out until the requested signal sig1 signal is
received
// descriptor_address.lw_offset contains the longword sram address
```

3.2.1.2.2 buf_dram_addr_from_index()

Calculates the DRAM buffer address from an index.

For example:

```
out_dram_addr = (in_index * d_size) + d_base
```

The index is more compact than the address itself—the maximum size of an index equals the total number of buffers available. Therefore a 16-bit index supports 64-KB buffers.

Microengine Assembler Syntax

```
buf_dram_addr_from_index (out_dram_addr, in_index, \
    POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(dram) void* ixp_buf_dram_addr_from_index(
    uint32_t in_index, pool_t POOL_ID, uint32_t d_base, uint32_t d_size,
    uint32_t s_base, uint32_t s_size);
```

Input

`in_index`	A relative index that identifies DRAM buffer and SRAM buffer descriptor addresses.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1 for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_dram_addr`	A DRAM buffer address. If `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

Estimated Size

Three to four instructions.

Microengine Assembler Example Usage

```
.reg dram_index out_address
#define FREELIST_HANDLE1, 1024, 128, 512, 32
immed32(dram_index, 4);
buf_dram_addr_from_index(out_address, dram_index, FREELIST_HANDLE);
   // out_address = 8192, contains the address for the specified index
```

Microengine C Example Usage

```
#define ADDRESS_MODE IXP2XXX
// use byte addressing for ADDRESS_MODE IXP2XXX
#define FREELIST_HANDLE POOL0,512,2048,2048,16
uint32_t index;
__declspec(dram) uint32_t *packet_address;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
index = ixp_buf_index_from_sram_addr(descriptor_address, FREELIST_HANDLE);
packet_address = ixp_buf_dram_addr_from_index(index, FREELIST_HANDLE);
```

3.2.1.2.3 buf_dram_addr_from_sram_addr()

Given an SRAM buffer descriptor address and a freelist handle, this operation calculates the DRAM buffer address.

Microengine Assembler Syntax

```
buf_dram_addr_from_sram_addr (out_dram_addr, in_sram_addr,
    POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(dram) void* ixp_buf_dram_addr_from_sram_addr(
    __declspec(sram) void* in_sram_addr,
    pool_t POOL_ID,
    uint32_t d_base,
    uint32_t d_size,
    uint32_t s_base,
    uint32_t s_size);
```

Input

`in_sram_addr`	An SRAM buffer descriptor address. If `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, "Common Arguments" for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_dram_addr` *or* Return Value	A DRAM buffer address. If `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

Estimated Size

Five to six instructions.

Microengine Assembler Example Usage

```
#define ADDRESS_MODE IXP2XXX
.reg sram_addr out_address
#define FREELIST_HANDLE1, 1024, 128, 512, 32
immed32(sram_addr, 0x100);
buf_dram_addr_from_sram_addr(out_address, sram_addr, FREELIST_HANDLE);
```

Microengine C Example Usage

```
#define ADDRESS_MODE IXP2XXX
// use byte addressing for ADDRESS_MODE IXP2XXX
#define FREELIST_HANDLE POOL0,512,2048,2048,16
__declspec(dram) uint32_t *packet_address;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
packet_address = ixp_buf_dram_addr_from_sram_addr(descriptor_address,
FREELIST_HANDLE);
```

3.2.1.2.4 `buf_free()`

Returns an SRAM address to the buffer pool identified by `FREELIST_HANDLE`.

Microengine Assembler Syntax

```
buf_free (in_sram_addr, POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE void ixp_buf_free(__declspec(sram) buf_handle_t in_sram_addr,
   pool_t POOL_ID, uint32_t d_base, uint32_t d_size, uint32_t s_base,
   uint32_t s_size);
```

Input

`in_sram_addr`	Buffer handle containing the SRAM address in longword form. If `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, “Common Arguments” for `FREELIST_HANDLE` argument descriptions.

Estimated Size

Three to six instructions.

Microengine Assembler Example Usage

See example usage in `buf_alloc()`.

Microengine C Example Usage

See example usage in `buf_alloc()`.

3.2.1.2.5 `buf_freelist_create()`

Creates an SRAM buffer (descriptor) freelist using SRAM queues. The freelist termination is indicated by a return value of zero from the call to `buf_alloc()`. The freelist is permanent. There is no operation to destroy a freelist.

Note: To avoid interfering with runtime performance, freelist creation should be performed only when the microengines are initialized.

Microengine Assembler Syntax

```
buf_freelist_create (in_num_buffers,
    POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE void ixp_buf_freelist_create(uint32_t in_num_buffers,
    pool_t POOL_ID, uint32_t d_base, uint32_t d_size, uint32_t s_base,
    uint32_t s_size);
```

Input

`in_num_buffers`	The number of buffers put on the freelist. The minimum value is one. The maximum value is arbitrary and is chosen based on the amount of memory available for buffer space. `maximum = (NUM_BUFFERS x d_size) AND (NUM_BUFFERS x s_size)`

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, “Common Arguments” for `FREELIST_HANDLE` argument descriptions.

Estimated Size

Eleven to fourteen instructions plus seven times `in_num_buffers` instructions.

Microengine Assembler Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500, 4
move(my_sbase, 128);
move(my_dbase, 0);
//
// dbase, dsize in quadwords, sbase ssize in longwords
#define FREELIST_HANDLE 1, my_dbase, 8, my_sbase, 164
// create freelist
buf_freelist_create(100, FREELIST_HANDLE);
```

Microengine C Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500,4
ixp_buf_freelist_create(300, FREELIST_HANDLE);
```

3.2.1.2.6 `buf_index_from_dram_addr()`

Calculates the array index from a DRAM buffer address according to the following formula:

```
out_index = (in_dram_addr - d_base)/ d_size
```

The index is more compact than the address itself, since its maximum size is the total number of buffers available. Therefore a 16-bit index supports 64-KB buffers.

Microengine Assembler Syntax

```
buf_index_from_dram_addr (out_index, in_dram_addr,
    POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_buf_index_from_dram_addr(
    __declspec(dram) void* in_dram_addr,
    pool_t POOL_ID,
    uint32_t d_base,
    uint32_t d_size,
    uint32_t s_base,
    uint32_t s_size);
```

Input

`in_dram_addr`	A DRAM buffer address. When the `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, "Common Arguments" for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_index` *or* Return Value	A relative index that identifies DRAM buffer and SRAM buffer descriptor addresses.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
move(my_sbase, 512);
move(my_dbase, 0);
//
move(my_ssize, 64);
move(my_dsize, 64);
#define FREELIST_HANDLE 0, my_dbase, my_dsize, my_sbase, my_ssize
immed32(dram_addr, 0x100);
    // get index from dram address
buf_index_from_dram_addr(out_index, dram_addr, FREELIST_HANDLE);
```

Microengine C Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500,4
uint32_t index;
__declspec(dram) uint32_t *packet_address;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
packet_address = ixp_buf_dram_addr_from_sram_addr(descriptor_address,
FREELIST_HANDLE);
index = ixp_buf_index_from_dram_addr(packet_address, FREELIST_HANDLE);
```

3.2.1.2.7 `buf_index_from_sram_addr()`

Calculates the array index from an SRAM buffer address according to the following formula:

```
out_index = (in_sram_addr - s_base) / s_size
```

The index is more compact than the address itself. An index has as its maximum size the total number of buffers available. Therefore a 16-bit index supports 64K buffers.

Microengine Assembler Syntax

```
buf_index_from_sram_addr (out_index, in_sram_addr, POOL_ID, \
    D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_buf_ index_from_sram_addr(
    __declspec(sram) void* in_sram_addr,
    pool_t POOL_ID,
    uint32_t d_base,
    uint32_t d_size,
    uint32_t s_base,
    uint32_t s_size);
```

Input

`in_sram_addr`	An SRAM buffer descriptor address. When the ADDRESS_MODE is IXP2XXX the result is a byte address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, "Common Arguments" for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_index` *or* Return Value	A relative index that identifies DRAM buffer-descriptor and SRAM buffer-descriptor addresses.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
move(my_sbase, 512);
move(my_dbase, 0);
//
move(my_ssize, 64);
move(my_dsize, 64);
#define FREELIST_HANDLE 0, my_dbase, my_dsize, my_sbase, my_ssize
immed32(sram_addr, 0x100);
    // get index from sram address
buf_index_from_sram_addr(out_index, sram_addr, FREELIST_HANDLE);
```

Microengine C Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500,4
uint32_t index;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
index = ixp_buf_index_from_sram_addr(descriptor_address, FREELIST_HANDLE);
```

3.2.1.2.8 `buf_sram_addr_from_dram_addr()`

Given a DRAM buffer address and a freelist handle this operation calculates the corresponding SRAM buffer address.

Microengine Assembler Syntax

```
buf_sram_addr_from_dram_addr (out_sram_addr, in_dram_addr,
    POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(sram) void* ixp_buf_sram_addr_from_dram_addr(
    __declspec(dram) void* in_dram_addr,
    pool_t POOL_ID,
    uint32_t d_base,
    uint32_t d_size,
    uint32_t s_base,
    uint32_t s_size);
```

Input

`in_dram_addr`	A DRAM buffer descriptor address. When the `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, "Common Arguments" for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_sram_addr` *or* Return Value	The SRAM buffer address. If `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

Estimated Size

Five to six instructions.

Microengine Assembler Example Usage

```
move(my_sbase, 512);
move(my_dbase, 0);
//
move(my_ssize, 64);
move(my_dsize, 64);
#define FREELIST_HANDLE 0, my_dbase, my_dsize, my_sbase, my_ssize
immed32(dram_addr, 0x100);
// get index from sram address
buf_sram_addr_from_dram_addr(out_address, dram_addr, FREELIST_HANDLE);
```

Microengine C Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500,4
uint32_t index;
__declspec(dram) uint32_t *packet_address;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
packet_address = ixp_buf_dram_addr_from_sram_addr(descriptor_address,
FREELIST_HANDLE);
descriptor_address =
    ixp_buf_sram_addr_from_dram_addr(packet_address, FREELIST_HANDLE);
```

3.2.1.2.9 `buf_sram_addr_from_index()`

Calculates the SRAM buffer address from a corresponding buffer index.

```
out_sram_addr = (in_index * s_size) + s_base
```

The index is more compact than the address itself—its maximum size is the total number of buffers available. A 16-bit index supports 64-KB buffers.

Microengine Assembler Syntax

```
buf_sram_addr_from_index (out_sram_addr, in_index, \
   POOL_ID, D_BASE, D_SIZE, S_BASE, S_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(sram) void* ixp_buf_sram_addr_from_index(
   uint32_t in_index,
   pool_t POOL_ID,
   uint32_t d_base,
   uint32_t d_size,
   uint32_t s_base,
   uint32_t s_size);
```

Input

`in_index`	A relative index that identifies a DRAM buffer and an SRAM buffer descriptor address.

`FREELIST_HANDLE`
`POOL_ID`
`D_BASE`
`D_SIZE`
`S_BASE`
`S_SIZE`
	See Section 3.2.1.1, “Common Arguments” for `FREELIST_HANDLE` argument descriptions.

Output/Returns

`out_sram_addr` ***or*** Return Value	The SRAM buffer descriptor address. When the `ADDRESS_MODE` is `IXP2XXX` the result is a byte address.

Estimated Size

Three to four instructions.

Microengine Assembler Example Usage

```
move(my_sbase, 512);
move(my_dbase, 0);
//
move(my_ssize, 64);
move(my_dsize, 64);
#define FREELIST 0, my_dbase, my_dsize, my_sbase, my_ssize
    // get index from sram address
immed32(sram_index, 20);
buf_sram_addr_from_index(out_address, sram_index, FREELIST);
```

Microengine C Example Usage

```
#define FREELIST_HANDLE POOL0,100,2048,500,4
uint32_t index;
__declspec(sram) uint32_t *descriptor_address;
ixp_buf_alloc(&descriptor_address, FREELIST_HANDLE, SIG_SRAM, SIG_NONE, ___);
index = ixp_buf_index_from_sram_addr(descriptor_address, FREELIST_HANDLE);
descriptor_address = ixp_buf_sram_addr_from_index(index, FREELIST_HANDLE);
```

3.2.2 cam

The `cam` interface performs microengine-local content addressable memory (CAM) access. This is an Intel® IXP2400 and IXP2800 Network Processor interface and is not supported by the Intel® IXP1200 Network Processor. Please refer to one of the following manuals for a description of this hardware feature:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Refer to Section 3.2.14, “cam Sharing” for information about the CAM sharing API.

Table 3-3. `cam` API

Name	Description
`cam_clear_all()`	Clears all CAM entries and resets all CAM locks. Puts a CAM in initial state.
`cam_match()`	Performs a content match to get a CAM state, hit/miss status and data.
`cam_read_data()`	Reads the data portion of a CAM entry.
`cam_read_entry()`	Reads the data and state of a CAM entry.
`cam_write_entry()`	Writes data and state to a CAM entry.

3.2.2.1 Common Arguments

The following arguments are used throughout `cam` operations.

Input

`in_data`	The 32-bit data to be written to a CAM entry.
`in_index`	The index of the CAM entry to be written.
`in_state`	The 4 bits of associated state of a CAM entry. For example, this could be used to hold a lock bit.

Output

`out_data`	The 32-bit value of a CAM entry.

Output

`out_index`	The index of a CAM entry.
`out_state`	The 4 bits of associated state of a CAM entry. For example, this could be used, in part, to hold a lock bit.
`out_status`	The resulting status of lookup. • `TRUE`—hit, the entry is valid • `FALSE`—miss, the entry is not valid See Section 2.6, “Defined Types, Enumerations, and Data Structures.”

3.2.2.2 API Functions

3.2.2.2.1 `cam_clear_all()`

Clears all CAM entries of a microengine and resets all CAM locks. This puts the CAM in an initial state.

Microengine Assembler Syntax

```
cam_clear_all();
```

Microengine C Syntax

```
INLINE void ixp_cam_clear_all();
```

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
// Clear all the cam entries
cam_clear_all();
```

Microengine C Example Usage

```
// Clear all the cam entries
ixp_cam_clear_all();
```

3.2.2.2.2 `cam_match()`

Performs a content match to return a CAM entry's state, hit-and-miss status, and data.

Microengine Assembler Syntax

```
cam_match(out_state, out_status, out_index, in_data);
```

Microengine C Syntax

See Section 2.6.2, "cam_lookup_t."

```
INLINE cam_lookup_t ixp_cam_match(uint32_t in_data);
```

Inputs and Outputs

See Section 3.2.2.1, "Common Arguments" for argument descriptions.

Estimated Size

Four to six instructions.

Microengine Assembler Example Usage

```
cam_clear();
// assign the value to match into "data"
immed32[in_data, 600]
cam_write_entry(1, in_data, 8);
cam_write_entry(2, 0xffffffff, 9);
   //...
result = cam_match(in_data);
```

Microengine C Example Usage

```
cam_lookup_t result;
uint32_t data;
// assign the value to match into "data"
result = ixp_cam_match(data);
```

3.2.2.2.3 `cam_read_data()`

Reads a CAM entry.

Microengine Assembler Syntax

```
cam_read_data(out_data, in_index);
```

Microengine C Syntax

```
INLINE uint32_t ixp_cam_read_data (uint32_t in_index);
```

Inputs and Outputs

See Section 3.2.2.1, "Common Arguments" for argument descriptions.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg out_data out_state
cam_clear();
    // assign the value to match into "data"
immed32[in_data, 600]
cam_write_entry(1, in_data, 8);
cam_write_entry(2, 0xffffffff, 9);
    //...
cam_read_data(out_data, 1);
```

Microengine C Example Usage

```
uint32_t cam_data, index;
// Assign value to "index"
cam_data = ixp_cam_read_data(index);
```

3.2.2.2.4 `cam_read_entry()`

Reads the data and state of a CAM entry.

See Section 2.6.1, "cam_entry_t."

Microengine Assembler Syntax

```
cam_read_entry(out_data, out_state, in_index);
```

Microengine C Syntax

```
INLINE cam_entry_t ixp_cam_read_entry(uint32_t in_index);
```

Inputs and Outputs

See Section 3.2.2.1, "Common Arguments" for argument descriptions.

Estimated Size

Three instructions.

Microengine Assembler Example Usage

```
.reg out_data out_state
cam_clear();
    // assign the value to match into "data"
immed32[in_data, 600]
cam_write_entry(1, in_data, 8);
cam_write_entry(2, 0xffffffff, 9);
    //...
cam_read_entry(out_data, out_state, 1);
```

Microengine C Example Usage

```
cam_entry_t cam_data;
uint32_t index;
cam_data = ixp_cam_read_entry(index);
```

3.2.2.2.5 `cam_write_entry()`

Write data and state to a CAM entry.

Microengine Assembler Syntax

```
cam_write_entry(in_index, in_data, in_state);
```

Microengine C Syntax

```
INLINE void ixp_cam_write_entry (uint32_t in_index, uint32_t in_data, uint32_t
in_state);
```

Inputs and Outputs

See Section 3.2.2.1, "Common Arguments" for argument descriptions.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
.reg out_data out_state
cam_clear();
   // assign the value to match into "data"
immed32[in_data, 600]
cam_write_entry(1, in_data, 8);
cam_write_entry(2, 0xffffffff, 9);
   //...
```

Microengine C Example Usage

```
uint32_t index, data, state;
// populate "index", "data", "state"
ixp_cam_write_entry(index, data, state);
```

3.2.3 critsect

The `critsect` interface is used to ensure that only one thread at a time is in a critical code section. If `THD_MODEL` is `LOCAL_THREADS`, one thread of a microengine can exclude other threads of the same microengine from entering the critical section. If `THD_MODEL` is `GLOBAL_THREADS`, any thread can exclude all other threads from entering the critical section. `critsect` is defined as a macro interface—in a function it is not possible to modify an argument that must be instantiated as a shared absolute register.

Currently critical sections zero through seven are supported, since the code is best optimized for this number of critical sections.

Note: For Microengine C critical section functionality refer to the *Mutual Exclusion Library* chapter of the *Intel® IXP2400/IXP2800 Network Processor Microengine C Compiler Language Support Reference Manual.*

Table 3-4. `critsect` API

Name	Description
`critsect_enter()`	Marks the entrance to a critical section.
`critsect_exit()`	Marks the exit from a critical section.
`critsect_init()`	Initializes a critical section

3.2.3.1 Common Arguments

3.2.3.1.1 Microengine C Data Types

The data types described in this section are provided for Microengine C programming.

Local Threads

```
typedef unsigned int                                   MUTEXID;
typedef volatile __declspec(shared gp_reg) unsigned int MUTEXLV;
```

Global Threads

```
typedef volatile SCRATCH_U32       MUTEXG;
typedef MUTEXG __declspec(import)   MUTEXG_IMPORT;
typedef MUTEXG __declspec(export)   MUTEXG_EXPORT;
```

Note: Refer to `MUTEXLV.h` and `MUTEXG.h` for details.

3.2.3.1.2 Arguments

The arguments in this section are used in all `critsect` operations.

Input

`THD_MODEL`	The thread model which can have a value of: • `LOCAL_THREADS`—all threads involved are on the same microengine • `GLOBAL_THREADS`—all threads involved are on multiple microengines

.

Input/Output

`io_critsect`	The variable containing the semaphore. The set or clear value of the semaphore is implementation-specific.

3.2.3.2 `critsect` Usage Examples

The following examples illustrate the use of the `critsect` interface.

Microengine Assembler Example Usage

```
critsect_init(@y01, LOCAL_THREADS);
critsect_enter(@y01, LOCAL_THREADS);
//
// critical section code goes here
//
critsect_exit(y01, LOCAL_THREADS);
```

Microengine C Example Usage One

```
// For Local threads.
#include <mutexlv.h>
MUTEXLV mutex;
// Make sure that init is called only once before calling any
// other critical section macros.

if (__ctx() == 0) {
    MUTEXLV_init(mutex);
    //Post signal or semaphore to other threads
    // waiting on init to continue.
    }

// Wait on MUTEXLV_init from thread 0.
MUTEXLV_lock(mutex, 0);

// Critical Section Code Goes Here

MUTEXLV_unlock(mutex,0);
```

Microengine C Example Usage Two

```
// For Global threads.
#include <mutexg.h>
MUTEXG_EXPORT mutex;
// Make sure that init is called only once before calling any
// other critical section macros.

if (__ctx() == 0) {
    MUTEXG_init(mutex);
    //Post signal or semaphore to other threads
    // waiting on init to continue.
    }
//
// Wait on MUTEXG_init from thread 0.
MUTEXG_lock(mutex);
//
// Critical Section Code Goes Here
//
MUTEXG_unlock(mutex);
```

3.2.3.3 API Functions

3.2.3.3.1 `critsect_enter()`

If a semaphore representing a critical section of microengine code is set, this macro swaps and retests until that semaphore is clear. If the semaphore is clear, the macro sets it.

Microengine Assembler Syntax

```
critsect_enter(io_critsect, THD_MODEL);
```

Inputs and Outputs

See Section 3.2.3.1, "Common Arguments" for argument descriptions.

Estimated Size

Three to five instructions when `THD_MODEL` is `LOCAL_THREADS`—that is, when there is no memory access.

Six to thirteen instructions when `THD_MODEL` is `GLOBAL_THREADS`—there is a cost of one atomic scratch access each time there is a check for critical section semaphore availability.

3.2.3.3.2 `critsect_exit()`

Clears a semaphore representing use of a critical section of microengine code.

Microengine Assembler Syntax

```
critsect_exit(io_critsect, THD_MODEL);
```

Inputs and Outputs

See Section 3.2.3.1, "Common Arguments" for argument descriptions.

Estimated Size

One to two instructions when `THD_MODEL` is `LOCAL_THREADS`—that is, when there is no memory access.

Two to five instructions when `THD_MODEL` is `GLOBAL_THREADS`—there is one scratch write to clear the semaphore.

3.2.3.3.3 `critsect_init()`

Initializes and clears a semaphore representing a critical section of microengine code.

Microengine Assembler Syntax

```
critsect_init(io_critsect, THD_MODEL);
```

Input

See Section 3.2.3.1, “Common Arguments” for argument descriptions.

Input/Output

`io_critsect`	On input, if `THD_MODEL` is: • `LOCAL_THREADS`—this is an absolute register that is used as a critical section parameter **NOTE:** An index is not allowed since eight bits (bits zero through seven) of this parameter are initialized to support up to eight critical sections. • `GLOBAL_THREADS`—this is a register or constant longword address. Each parameter supports one critical section only

See Section 3.2.3.1, “Common Arguments” for argument descriptions.

Estimated Size

One instruction when `THD_MODEL` is `LOCAL_THREADS`—that is, when there is no memory access.

Two to five instructions when `THD_MODEL` is `GLOBAL_THREADS`—there is one scratch write to initialize the semaphore.

3.2.4 `cycle`

The `cycle` interface performs timer-related functionality. The interface uses the chip cycle count hardware resource. The cycle count updates every core machine cycle. For example, at 200MHz, the cycle count updates every 5ns. Timers are typically used to shape the transmission of packet data at specified rates. Transmit may poll the cycle count using `cycle64_read()` and check whether packet data should be sent by comparing the elapsed interval to `cycle32_diff()`.

Table 3-5. `cycle` API

Name	Description
`cycle32_delay()`	Delays `n` cycles without a context swap.
`cycle32_diff()`	Calculates the difference of two 32-bit cycle counts.
`cycle64_diff()`	Calculates the difference of two 64-bit cycle counts.
`cycle32_read()`	Returns the 32-bit cycle count.
`cycle64_read()`	Returns the 64-bit cycle count.
`cycle32_sleep()`	Sleeps until `n` cycles have elapsed, allowing other contexts to run.

3.2.4.1 API Functions

3.2.4.1.1 `cycle32_delay()`

Delays `n` cycles without a context swap.

Microengine Assembler Syntax

```
cycle32_delay (in_cycle_count);
```

Microengine C Syntax

```
INLINE void ixp_cycle32_delay (uint32_t in_cycle_count);
```

Input

`in_cycle_count`	The approximate number of cycles to delay. This is a general-purpose register or a constant.

Estimated Size

- For Microengine Assembler—seven to ten instructions
- For Microengine C—fifteen instructions

Microengine Assembler Example Usage

```
cycle32_delay(50);
```

Microengine C Example Usage

```
ixp_cycle32_delay(50); //  spin delay for 50 cycles
```

3.2.4.1.2 `cycle32_diff()`

Returns the elapsed cycle count. The return value is restricted to a 32-bit difference.

Microengine Assembler Syntax

```
cycle32_diff (out_diff, in_later_timer, in_prev_timer);
```

Microengine C Syntax

```
INLINE uint32_t ixp_cycle32_diff(uint32_t in_later_timer, uint32_t
in_prev_timer);
```

Input

`in_later_timer`	The new, most recent, timer value.
`in_prev_timer`	The previous timer value.

Output/Returns

`out_diff`	The 32-bit difference of `in_later_timer` and `in_prev_timer`. This is equivalent to the expression: `in_later_timer - in_prev_timer` This value accounts for the 32-bit cycle count wrap.

Estimated Size

- For Microengine Assembler—five to ten instructions
- For Microengine C—five to nine instructions

Microengine Assembler Example Usage

```
cycle32_read(a);
cycle32_read(b);
cycle32_diff (diff, a, b);
```

Microengine C Example Usage

```
uint32_t a, b, diff;
a = (uint32_t) ixp_cycle_read(); // read and cast cycle to 32-bit unsigned int
b = (uint32_t) ixp_cycle_read(); // read and cast cycle to 32-bit unsigned int
diff = ixp_cycle32_diff(b, a);
```

3.2.4.1.3 cycle64_diff()

Returns the elapsed cycle count. Restricted to a 64-bit difference.

Microengine Assembler Syntax

```
cycle64_diff (out_diff_hi, out_diff_lo, in_later_timer_hi, \
        in_later_timer_lo, in_prev_timer_hi, in_prev_timer_lo);
```

Microengine C Syntax

```
INLINE U64 ixp_cycle64_diff(U64 in_later_timer, U64 in_prev_timer);
```

Input

`in_later_timer_hi`	The high 32 bits of the new—that is, most recent—timer value.
`In_later_timer_lo`	The low 32 bits of the new—that is, most recent—timer value.
`in_prev_timer_hi`	The high 32 bits of the previous timer value.
`in_prev_timer_lo`	The low 32 bits of the previous timer value.
`in_later_timer`	The new—that is, most recent—64-bit timer value.
`in_prev_timer`	The previous 64-bit timer value.

Output/Returns

`out_diff_hi`	The high 32-bit difference of `in_later_timer` and `in_prev_timer` accounting for 64-bit cycle count wrap.
`out_diff_lo`	The low 32-bit difference of `in_later_timer` and `in_prev_timer` accounting for 64-bit cycle count wrap.

Estimated Size

- For Microengine Assembler—two to four instructions
- For Microengine C—five to twelve instructions

Microengine Assembler Example Usage

```
cycle64_read(prev_hi, prev_lo);
// other code
cycle64_read(later_hi, later_lo);
cycle64_diff (diff_hi, dif_lo, later_hi, later_lo, prev_hi, prev_lo);
```

Microengine C Example Usage

```
U64 a, b, diff;
a = ixp_cycle_read();           // read 64-bit timer value
b = ixp_cycle_read();           // read 64-bit timer value
diff = ixp_cycle64_diff(b, a);
```

3.2.4.1.4 `cycle32_read()`

Returns the 32-bit current cycle count.

Microengine Assembler Syntax

```
cycle32_read (out_cycle_lo);
```

Microengine C Syntax

```
typedef unsigned long cycle_t;
```

Output

`out_cycle_lo`	The low 32 bits of the 64-bit chip cycle count.

Estimated Size

- Microengine Assembler—two instructions
- Microengine C—eight instructions

Microengine Assembler Example Usage

```
cycle32_read(a);     // get low 32 bits of cycle count
```

Microengine C Example Usage

```
uint32_t a;
a = (uint32_t) ixp_cycle_read(); // read and cast cycle to 32-bit unsigned int
```

3.2.4.1.5 `cycle64_read()`

Returns the current 64-bit cycle count.

Microengine Assembler Syntax

```
cycle64_read (out_cycle_hi, out_cycle_lo);
```

Microengine C Syntax

```
INLINE cycle_t ixp_cycle_read ();
```

Output

`out_cycle_hi`	The high 32 bits of the 64-bit chip cycle count.
`out_cycle_lo`	The low 32 bits of the 64-bit chip cycle count.

Estimated Size

- For Microengine Assembler—four instructions
- For Microengine C—eight instructions

Microengine Assembler Example Usage

```
cycle64_read(b, a);      // get 64 bits of cycle count
```

Microengine C Example Usage

```
cycle_t a;
a = ixp_cycle_read();    // get the full 64-bit cycle count
```

3.2.4.1.6 `cycle32_sleep()`

Sleeps `n` cycles after a context swap that allows other threads on the microengine to run.

Microengine Assembler Syntax

```
cycle32_sleep (in_cycle_count);
```

Microengine C Syntax

```
INLINE void ixp_cycle32_sleep (uint32_t in_cycle_count);
```

Input

`in_cycle_count`	The approximate number of cycles to delay.

Estimated Size

- For Microengine Assembler—seven to eight instructions
- For Microengine C—fifteen instructions

Microengine Assembler Example Usage

```
cycle32_sleep(600);
```

Microengine C Example Usage

```
ixp_cycle32_sleep(600);     // sleep for 600 cycles
```

3.2.5 dram

The dram interface can be used for concurrent DRAM access, specifying hardware queue types, multiword reads and writes, and DRAM operations that perform more than just simple read and write access.

Table 3-6. dram API

Name	Description
dram_mask_write()	Writes bytes selected by in_byte_mask to a DRAM quadword.
dram_read()	Reads in_qw_count quadwords from DRAM.
dram_rbuf_read()	Copies in_qw_count quadwods from RBUF to DRAM.
dram_tbuf_write()	Copies in_qw_count quadwords from DRAM to TBUF.
dram_write()	Writes in_qw_count quadwords to DRAM.

3.2.5.1 Common Arguments

The following arguments are used throughout the DRAM access operations.

Input

in_data	The value to be written.
in_dram_addr	The DRAM address to which the data is to be written. If ADDRESS_MODE is IXP2XXX, then it is a byte address.
in_addr_offset	For *Microengine Assembler only*, this offset is added to in_dram_addr to form the address to be accessed.
REQ_SIG	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
in_wakeup_sigs	The list of signals causing the thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
Q_OPTION	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

3.2.5.2 API Functions

3.2.5.2.1 `dram_mask_write()`

Writes the bytes selected by `in_byte_mask` to a DRAM quadword.

Microengine Assembler Syntax

```
dram_mask_write (in_data, in_dram_addr, in_addr_offset,
    in_byte_mask,
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_dram_mask_write(
    __declspec(dram_write_reg) void *in_data,
    volatile __declspec(dram) void * in_dram_addr,
    uint32_t in_byte_mask,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_data`	See Section 3.2.5.1, “Common Arguments” for details.
`in_dram_addr`	See Section 3.2.5.1, “Common Arguments” for details.
`in_addr_offset`	See Section 3.2.5.1, “Common Arguments” for details.
`in_byte_mask`	The register or constant 8-bit mask indicating which bytes to write. The bits in the mask correspond to bytes, left to right. For example, `0x80` specifies the left-most byte while `0x1` specifies the right-most byte.
`REQ_SIG`	See Section 3.2.5.1, “Common Arguments” for details.
`in_wakeup_sigs`	See Section 3.2.5.1, “Common Arguments” for details.
`Q_OPTION`	See Section 3.2.5.1, “Common Arguments” for details.

Estimated Size

Three to eight instructions.

Microengine Assembler Example Usage

```
.reg dram_addr
.reg $$buf0
.sig tmp_sig2
immed32($$buf0, 0xa73265da);
immed32(dram_addr, 0x1000);
dram_mask_write($$buf0, dram_addr, 0, 0x55, tmp_sig2, tmp_sig2, ___);
```

Microengine C Example Usage

```
__declspec(dram_write_reg) uint32_t data[2];
__declspec(dram) void *addr;
SIGNAL_PAIR sig_dram;
SIGNAL_MASK wake_up_sigs;
data[1] = 0;
data[0] = 0x00ff00ff
addr = (__declspec(dram) void *)0x100;
wake_up_sigs = __signals ( &sig_dram );
 // write big-endian bytes 5, 7
ixp_dram_mask_write(data, addr, 0x5, &sig_dram, wake_up_sigs, ___);
```

3.2.5.2.2 `dram_read()`

Reads `in_qw_count` quadwords from DRAM.

Microengine Assembler Syntax

```
dram_read (out_data, in_dram_addr, in_addr_offset, in_qw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_dram_read(__declspec(
    dram_read_reg) void *out_data,
    volatile __declspec(dram) void* in_dram_addr,
    uint32_t in_qw_count,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_data`	See Section 3.2.5.1, “Common Arguments” for details.
`in_dram_addr`	See Section 3.2.5.1, “Common Arguments” for details.
`in_addr_offset`	See Section 3.2.5.1, “Common Arguments” for details.
`in_qw_count`	The number of quadwords to read. The minimum quadword count is one. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors, the maximum quadword count is 16.
`REQ_SIG`	See Section 3.2.5.1, “Common Arguments” for details.
`in_wakeup_sigs`	See Section 3.2.5.1, “Common Arguments” for details.
`Q_OPTION`	See Section 3.2.5.1, “Common Arguments” for details.

Output

`out_data`	The data read from the location `in_dram_addr`.

Estimated Size

One to seven instructions.

Microengine Assembler Example Usage

```
.reg dram_addr
.reg $$rreg0 $$rreg1
.xfer_order $$rreg0 $$rreg1
.sig tmp_sig2
immed32(dram_addr, 0x1000);
dram_read($$rreg0, dram_addr, 0, 1, tmp_sig2, tmp_sig2, ___);
```

Microengine C Example Usage One

```
__declspec(dram_read_reg) uint32_t data[4];
__declspec(dram) UINTvoid *addr;
SIGNAL_PAIR sig_dram1;
SIGNAL_MASK wake_up_sigs;
addr = (__declspec(dram) void *)0x100;
wake_up_sigs = __signals( &sig_dram1 );
ixp_dram_read(data, addr, 2, SIG_DRAM&sig_dram1, SIG_DRAMwake_up_sigs, ___);
// read 2 quadwords
```

Microengine C Example Usage Two

```
__declspec(dram_read_reg) uint32_t data[16];
__declspec(dram) void *addr;
SIGNAL_PAIR sig_dram1, sig_dram2;
SIGNAL_MASK wake_up_sigs;
addr =(__declspec(dram) void *) 800;
// read 4 quad words
ixp_dram_read(data, addr, 4, &sig_dram1, 0, ___);
wake_up_sigs = __signals( &sig_dram1, &sig_dram2 );
//
addr =(__declspec(dram) void *) 832;
// read 4 quadwords and return after sig_dram1 & sig_dram2
// signal pairs consumed
ixp_dram_read(&data[8], addr, 4, &sig_dram2, wake_up_sigs, ___);
//
// Call implicit read to tell compiler that signal scope
__implicit_read(&sig_dram1);
__implicit_read(&sig)dram2);
```

3.2.5.2.3 `dram_rbuf_read()`

Copies `in_qw_count` quadwords from RBUF to DRAM. An RBUF is the interface buffer for data received from the network. For details on a network processor's RBUF implementation, refer to one of the following documents:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
dram_rbuf_read (in_dram_addr, in_dram_addr_offset, \
    in_rbuf_addr, in_rbuf_addr_offset, in_qw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_dram_rbuf_read(
volatile __declspec(dram) void* in_dram_addr,
volatile __declspec(rbuf) void* in_rbuf_addr, uint32_t in_qw_count,
SIGNAL_PAIR *REQ_SIG, SIGNAL_MASK in_wakeup_sigs, queue_t Q_OPTION);
```

Input

`in_dram_addr`	See Section 3.2.5.1, "Common Arguments" for details.
`in_addr_offset`	See Section 3.2.5.1, "Common Arguments" for details.
`in_rbuf_addr`	The address of RBUF. If `ADDRESS_MODE` is `IXP2XXX`, then this is a byte address.
`in_rbuf_addr_offset`	For *Microengine Assembler only*, the offset to be added to `in_rbuf_addr` to form the address to be accessed.
`in_qw_count`	A register or constant indicating the number of quadwords to write. The minimum quadword count is one. The maximum quadword count is 16.
`REQ_SIG`	See Section 3.2.5.1, "Common Arguments" for details.
`in_wakeup_sigs`	See Section 3.2.5.1, "Common Arguments" for details.
`Q_OPTION`	See Section 3.2.5.1, "Common Arguments" for details.

Estimated Size

Two to five instructions.

Microengine Assembler Example Usage

```
#define_eval RBUF_BASE 0x2000
.reg dram_addr
.reg $$rreg0 $$rreg1 $$rreg2 $$rreg3
.xfer_order $$rreg0 $$rreg1 $$rreg2 $$rred3
.sig tmp_sig2

immed32(dram_addr, 0x1000);
dram_rbuf_read(dram_addr, 0, RBUF_BASE, 0, 2, tmp_sig2, tmp_sig2, ___);
```

Microengine C Example Usage

```
__declspec(dram) void *dram_addr;
__declspec(rbuf) void *rbuf_addr;
SIGNAL_PAIR sig_dram1;
SIGNAL_MASK wake_up_sigs;
//
dram_addr = (__declspec(dram) void *)0x100;
rbuf_addr =(__declspec(rbuf) void *) 0x2000;
wake_up_sigs = __signals( &sig_dram1 );
// copy 64 bytes rbuf element 0 to dram address 0x100
ixp_dram_rbuf_read(dram_addr, rbuf_addr, 8, &sig_dram1, wake_up_sigs, ___);
```

3.2.5.2.4 `dram_tbuf_write()`

Copies `in_qw_count` quadwords from DRAM to TBUF. TBUF is the interface buffer for data to be transmitted to the network. For details on a network processors TBUF implementation, refer to one of the following manuals:

- *Intel® IXP 2400 Hardware Reference Manual*
- *Intel® IXP 2800 Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
dram_tbuf_write (in_dram_addr, in_dram_addr_offset,
in_tbuf_addr, in_tbuf_addr_offset,
in_qw_count, REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_dram_tbuf_write(
    volatile __declspec(dram) void* in_dram_addr,
    volatile void __declspec(tbuf) *in_tbuf_addr,
    uint32_t in_qw_count,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_dram_addr`	See Section 3.2.5.1, “Common Arguments” for details.
`in_addr_offset`	See Section 3.2.5.1, “Common Arguments” for details.
`in_tbuf_addr`	The address of TBUF. If `ADDRESS_MODE` is `IXP2XXX`, then this is a byte address.
`in_tbuf_addr_offset`	For *Microengine Assembler only*, this is an offset to be added to `in_tbuf_addr` to form the address to be accessed.
`in_qw_count`	A general-purpose register or constant that specifies the number of quadwords to write. The minimum quadword count is one. The maximum quadword count is 16.
`REQ_SIG`	See Section 3.2.5.1, “Common Arguments” for details.
`in_wakeup_sigs`	See Section 3.2.5.1, “Common Arguments” for details.
`Q_OPTION`	See Section 3.2.5.1, “Common Arguments” for details.

Estimated Size

Two to ten instructions.

Microengine Assembler Example Usage

```
#define_eval TBUF_BASE 0x2000
.reg dram_addr
.sig tmp_sig2
//
immed32(dram_addr, 0x1000);
dram_tbuf_write(dram_addr, 0, TBUF_BASE, 0, 16, tmp_sig2, tmp_sig2, ___);
```

Microengine C Example Usage

```
__declspec(dram) void *dram_addr;
__declspec(tbuf) void *tbuf_addr;
SIGNAL_PAIR sig_dram1;
//
dram_addr = (__declspec(dram) void *)0x100;
tbuf_addr = (__declspec(tbuf) void *)0x2000;
// copy 16 bytes to tbuf element 1 (if element size is 64 Bytes),
// from dram address 0x100
ixp_dram_tbuf_write(dram_addr, tbuf_addr, 2, &sig_dram1,
    __signals( &sig_dram1 ), ___);
```

3.2.5.2.5 `dram_write()`

Writes `in_qw_count` quadwords to DRAM memory.

Microengine Assembler Syntax

```
dram_write (in_data, in_dram_addr, in_addr_offset, in_qw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_dram_write(__declspec(dram_write_reg) void *in_data,
    volatile __declspec(dram) void* in_dram_addr,
    uint32_t in_qw_count,
    SIGNAL_PAIR* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_data`	See Section 3.2.5.1, "Common Arguments" for details.
`in_dram_addr`	See Section 3.2.5.1, "Common Arguments" for details.
`in_addr_offset`	See Section 3.2.5.1, "Common Arguments" for details.
`in_qw_count`	The number of quadwords to write. The minimum quadword count is one. The maximum quadword count is 16.
`REQ_SIG`	See Section 3.2.5.1, "Common Arguments" for details.
`in_wakeup_sigs`	See Section 3.2.5.1, "Common Arguments" for details.
`Q_OPTION`	See Section 3.2.5.1, "Common Arguments" for details.

Estimated Size

One to seven instructions.

Microengine Assembler Example Usage

```
.reg dram_addr
.reg $$wreg0 $$wreg1 $$wreg2 $$wreg3
.xfer_order $$wreg0 $$wreg1 $$wreg2 $$wreg3
.sig tmp_sig2
//
immed32($$wreg0, 0xabcdef01);
immed32($$wreg1, 0x12345678);
immed32($$wreg2, 0xebcdef01);
immed32($$wreg3, 0xc2345678);
immed32(dram_addr, 0x1000);
dram_write($$wreg0, dram_addr, 0, 2, tmp_sig2, tmp_sig2, ___);
```

Microengine C Example Usage

```
__declspec(dram_write_reg) uint32_t data[4];
SIGNAL_PAIR sig_dram1;
data[0] = 0x01234567;
data[1] = 0x89ABCDEF
data[2] = 0x12345678;
data[3] = 0x87654321;
//
__declspec(dram) void *addr;
addr = (__declspec(dram) void *)0x100;
ixp_dram_write(data, addr, 2, &sig_dram1, __signals(&sig_dram1), ___);
       //write 2 quadwords
```

3.2.6 `localmem`

The `localmem` interface enables reading from and writing to local memory. Performs one to eight word reads to and writes from registers using separate assembly language macros. This is an interface for Intel® IXP2400, IXP2800 and IXP2850 Network Processors and is not supported for the Intel® IXP1200 Network Processor. Local memory is on-IXP-chip memory local to a microengine. Multi-word reads to or writes from transfer registers are possible. Refer to one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

When using the Microengine C language functions in this API, the memory type of the source and destination is specified by the calling application.

Table 3-7. `localmem` API

Name	Description
`localmem_set_address()`	Sets the local memory address for a local memory handle.
`localmem_set_active_handle()`	Sets the current active local memory access handle. *Microengine Assembler only.*
`localmem_read_next()`	Reads the location and increments the local memory address for a local memory handle.
`localmem_read()` [`localmem_read1()` through `localmem_read8()`]	Reads `in_lw_count` 32-bit longwords from local memory.
`localmem_write()` [`localmem_write1()` through `localmem_write8()`]	Writes `in_lw_count` 32-bit longwords to local memory.
`localmem_write_next()`	Writes the location and increments the local memory address for a local memory handle.

3.2.6.1 API Functions

3.2.6.1.1 `localmem_set_address()`

Sets the local memory address for a local memory handle and makes the handle active.

Microengine Assembler Syntax

```
localmem_set_address(in_lm_addr, in_addr_offset,  LM_HANDLE);
```

Microengine C Syntax

```
INLINE uint32_t  ixp_localmem_set_address(lm_handle_t LM_HANDLE,
    volatile void __declspec(local_mem) *in_lmaddr);
```

Input

`in_lm_addr`	A local memory address.
`in_addr_offset`	For *Microengine Assembler only*, the offset to be added to `in_lm_addr` to form the address to be accessed.
`LM_HANDLE`	The identification of a local memory handle. Either `LM_HANDLE_0` or `LM_HANDLE_1`.

See Section 2.6.5, "lm_handle_t."

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
localmem_set_address(500, 0, LM_HANDLE_0);
nop    //3 cycle latency
nop
nop
```

Microengine C Example Usage

```
uint32_t gpr_readData;
gpr_writeData = 0x234f56a1;
ixp_localmem_set_address(0, (volatile void __declspec(local_mem)*)0x100);
```

3.2.6.1.2 `localmem_set_active_handle()`

Sets the current local memory access handle.

This function is available in *Microengine Assembler only.*

Microengine Assembler Syntax

```
localmem_set_active_handle(LM_HANDLE);
```

Microengine C Syntax

Input

`LM_HANDLE`	The identification of the local memory handle. Either `LM_HANDLE_0` or `LM_HANDLE_1`.

See Section 2.6.5, "lm_handle_t."

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
localmem_set_active_handle(LM_HANDLE_1);
```

3.2.6.1.3 `localmem_read_next()`

Reads a next 32-bit word from the local memory and increments the index of the local memory.

Note: This function should be called only after a call to `localmem_write()` or `localmem_set_address()`.

Microengine Assembler Syntax

```
localmem_read_next (out_data);
```

Microengine C Syntax

```
INLINE uint32_t ixp_localmem_read_next(lm_handle_t LM_HANDLE)
```

Output/Returns

`out_data`	The data read from local memory.

Input

`LM_HANDLE`	The identification of a local memory handle. This value is one of: • `LM_HANDLE_0` • `LM_HANDLE_1` See Section 2.6.5, “lm_handle_t.”

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg x y z out_data1 out_data2 out_data3 out_data4
immed32[x, 0xffffffff]
immed32[y,0xffff]
immed32[z, 0xfffff]
localmem_write4(0xfffffff, x, y, z, 100, 0);
localmem_set_address(100, 0, LM_HANDLE_0);
nop
nop
nop
localmem_read_next(out_data1);
```

Microengine C Example Usage

```
uint32_t writeData0, writeData1, readData0, readData1;
myBase0 = 0x01;
writeData0 = 0x87654321;
ixp_localmem_write(&writeData0, 0, (volatile void
__declspec(local_mem)*)myBase0, 1);
ixp_localmem_set_address(
       0, (volatile void __declspec(local_mem)*)myBase0);
readData0 = ixp_localmem_read_next(0);
```

3.2.6.1.4 localmem_read()

Reads in_lw_count words at local memory address in_lm_addr to out_data.

Microengine Assembler Syntax

```
localmem_read1 (out_data, in_lm_addr, in_addr_offset);
localmem_read2 (out_data0, out_data1, in_lm_addr, in_addr_offset);
localmem_read3 (out_data0, out_data1, out_data2, \
    in_lm_addr, in_addr_offset);
```

to

```
localmem_read8 (out_data0, out_data1, out_data2, out_data3, \
    out_data4, out_data4, out_data6, out_data7,
    in_lm_addr, in_addr_offset);
```

Note: Separate out_data args are used in Microengine Assembler for counts of one to four because there is no concept of array or structure as there is in the C language. As a result, in Microengine Assembler the argument in_lw_count is not needed.

Microengine C Syntax

```
INLINE void  ixp_localmem_read(
    void *out_data,
    lm_handle_t LM_HANDLE,
    volatile void __declspec(local_mem) *in_lmaddr,
    uint32_t in_lw_count)
```

Input

in_lm_addr	A local-memory address.
in_addr_offset	For *Microengine Assembler only*, the offset to be added to in_lm_addr to form the address to be accessed.
in_lw_count	The number of words to read. For Microengine Assembler this argument is not used when its value is one to four, inclusive.
LM_HANDLE	The identification of a local memory handle. This value is one of: • LM_HANDLE_0 • LM_HANDLE_1 See Section 2.6.5, “lm_handle_t.”

Output

out_data *to* out_dataN	The resulting 32-bit words.

Estimated Size

Five to eight instructions plus one additional instruction for each word read.

Microengine Assembler Example Usage

```
immed32[my_lmdata1, 0x00ffffff]
immed32[my_lmdata2, 0x00]
alu[my_lmbase, --, B, 100]
localmem_write2(my_lmdata1, my_lmdata2, my_lmbase, 0);
localmem_read2(my_read_data1, my_read_data2, my_lmbase, 0);
```

Microengine C Example Usage

```
uint32_t myBase;
uint32_t gpr_writeData;
uint32_t gpr_readData;
myBase= 0x100;
gpr_writeData = 0x234f56a1;
ixp_localmem_set_address(0, (volatile void __declspec(local_mem)*)0x100);
ixp_localmem_write(
        &gpr_writeData,
        0,
        (volatile void __declspec(local_mem)*)myBase,
        1);
ixp_localmem_read(
        &gpr_readData,
        0,
        (volatile void __declspec(local_mem)*)myBase,
        1);
```

3.2.6.1.5 `localmem_write()`

Writes to the local memory address `in_lm_addr` a count of `in_lw_count` 32-bit longwords from `in_data`.

Microengine Assembler Syntax

```
localmem_write1 (in_data, in_lm_addr, in_addr_offset);
localmem_write2 (in_data0, in_data1, in_lm_addr, in_addr_offset);
localmem_write3 (in_data0, in_data1, in_data3, \
in_lm_addr, in_addr_offset);
```

to

```
localmem_write8 (in_data0, in_data1, in_data2, in_data3, \
in_data4, in_data5, in_data6, in_data7, in_lm_addr, in_addr_offset);
```

Note: Separate `in_data` args are used in Microengine Assembler for counts of one to four because there is no concept of array or structure as there is in the C language. Therefore, the argument `in_lw_count` is not needed.

Microengine C Syntax

```
INLINE void  ixp_localmem_write(
    void* in_data,
    lm_handle_t LM_HANDLE,
    volatile void __declspec(local_mem) *in_lmaddr,
    uint32_t in_lw_count)
```

Input

`in_data` ***to*** `in_dataN`	One or more 32-bit words.
`in_lm_addr`	A local-memory byte address.
`in_addr_offset`	For *Microengine Assembler only*, the offset to be added to `in_lm_addr` to form the address to be accessed.
`in_lw_count`	The number of words to read.
`LM_HANDLE`	The identification of a local memory handle. This value is one of: • `LM_HANDLE_0` • `LM_HANDLE_1` See Section 2.6.5, “lm_handle_t.”

Estimated Size

Five to eight instructions plus one additional instruction for each word written.

Microengine Assembler Example Usage

```
immed32(lm_loc, 4);
immed32(data0, 0x01234567);
immed32(data1, 0x89abcdef);
local_mem_write2(data0, data1, , lm_loc, 0)
```

Microengine C Example Usage

```
uint32_t myBase;
uint32_t gpr_writeData;
uint32_t gpr_readData;
myBase= 0x100;
gpr_writeData = 0x234f56a1;
ixp_localmem_set_address(0, (volatile void __declspec(local_mem)*)0x100);
ixp_localmem_write(
    &gpr_writeData,
    0,
    (volatile void __declspec(local_mem)*)myBase,
    1);
```

3.2.6.1.6 `localmem_write_next()`

Auto-increments a local memory address and writes the next 32-bit longword to local memory.

Microengine Assembler Syntax

```
localmem_write_next (in_data);
```

Microengine C Syntax

```
INLINE void  ixp_localmem_write_next(uint32_t in_data, lm_handle_t LM_HANDLE)
```

Input

`in_data`	The data to write to local memory.
`LM_HANDLE`	The identification of a local memory handle. This value is one of: • `LM_HANDLE_0` • `LM_HANDLE_1` See Section 2.6.5, “lm_handle_t.”

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
localmem_write1(in_data1, lm_loc, 0)
localmem_write_next(in_data2);
```

Microengine C Example Usage

```
uint32_t myBase0;
uint32_t writeData0;
myBase0 = 0x01;
writeData0 = 0x87654321;
ixp_localmem_set_address(0, (volatile void __declspec(local_mem)*)myBase0);
ixp_localmem_write_next(writeData0, 0);
```

3.2.7 mailbox

The `mailbox` interface implements mailbox operations for the purpose of interthread messaging. When the mailbox is written using `mailbox_send()`, bit 31 of the first 32 bit word is set as a valid bit along with the rest of the data. The operation `mailbox_receive()` can optionally retry reading the mailbox until the valid bit is set. This interface is typically used in single producer situations—that is, when one thread sends a message to the mailbox. However, mailbox use can be made thread-safe for multi-producer situations through calling-application invocation of `critsect_enter()` and `critsect_exit()` around the mailbox operations `mailbox_is_full()` and `mailbox_send()`.

Note: The operation `msgq_receive()` is multi-consumer.

An example use of a mailbox is to pass information between software stages where one thread (stage) hands off to another thread (stage).

Table 3-8. mailbox API

Name	Description
`mailbox_is_full()`	Tests whether a mailbox is full.
`mailbox_init()`	Initializes a mailbox.
`mailbox_receive()`	Receives a message.
`mailbox_send()`	Sends a message.

3.2.7.1 API Functions

3.2.7.1.1 mailbox_is_full()

Tests whether a mailbox is full. If full, there is no room for a message and the application should retest before issuing `mailbox_send()`.

Microengine Assembler Syntax

```
mailbox_is_full(in_addr, in_addr_offset, IN_MEM_TYPE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_mailbox_is_full( mbox_t in_addr,  mem_t IN_MEM_TYPE);
```

Input

`in_addr`	The mailbox address in bytes.
`in_addr_offset`	If SRAM or scratch memory is used for the mailbox, this offset is added to `in_addr` to form the mailbox address.
`IN_MEM_TYPE`	The memory source used for the mailbox. This value is one of {`SRAM`, `SCRATCH`, or `XFER`}.

See Section 2.6.7, "mem_t."

Output/Returns

Return Value ***or*** Condition Code	The return value. • Zero—the mailbox is full • Non-zero—the mailbox is *not* full

Estimated Size

Two instructions. Add one memory read instruction if the mailbox is in SRAM or scratch memory.

Microengine Assembler Example Usage

```
.sig SIG_MB
// Initialize a scratch type mailbox of 2 longwords in size
mailbox_init(base_addr, offset, scratch, 2, SIG_MB, SIG_MB, ___)
// Check if the scratch type mailbox is full
mailbox_is_full(base_addr, offset, scratch)
// 1 for not full (condition code)
bne[passed#]
```

Microengine C Example Usage

```
// see also ixp_mailbox_send
uint32_t my_base = 0x200;
uint32_t result;
mem_t mem_type = SRAM;
mbox_t x;
x.sram_addr = (__declspec(sram) void*)my_base;
result = ixp_mailbox_is_full (x, mem_type);
//Check result: 1 - FULL; 0 - Empty
```

3.2.7.1.2 `mailbox_init()`

Initializes a mailbox. The mailbox may be scratch, SRAM, another thread's transfer registers. It is only necessary to make sure bit 31 of `mailbox[0]` is set. Initializes the memory locations to zero—except for bit 31 of `mailbox[0]` which is one. `mailbox_init()` must be called by either sender or receiver before `mailbox_send()` or `msgq_receive()` is called.

Microengine Assembler Syntax

```
mailbox_init (in_addr_or_thd,  in_offset_or_xfer, IN_MEM_TYPE, \
    in_mb_size, IN_REQ_SIG, in_wakeup_sigs, IN_Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_mailbox_init(
    mbox_t in_addr_or_thd,
    mem_t IN_MEM_TYPE,
    uint32_t  in_mb_size,
    SIGNAL *IN_REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t IN_Q_OPTION);
```

Input

`in_addr_or_thd`	The mailbox address in bytes for `IN_MEM_TYPE` of `SRAM` and `SCRATCH` or the thread number for `IN_MEM_TYPE` of `XFER` registers A constant value should be provided when specifying a thread number. • For an IXP2400—zero to 63 • For an IXP2800—zero to 128
`In_offset_or_xfer`	For a value of `MEM_TYPE` of either `SRAM` or `SCRATCH`, this is the offset to be added to the mailbox base address. For a value of `MEM_TYPE` of `XFER`, this is the read transfer register for the mailbox of the remote destination thread.
`IN_MEM_TYPE`	The type of memory for mailbox storage. This value is one of {`SRAM`, `SCRATCH`, `XFER`}. See Section 2.6.7, "mem_t."
`In_mb_size`	The mailbox size in 32-bit words. Maximum mailbox size is eight.
`In_REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage." **NOTE:** For a value of `MEM_TYPE` of `XFER` the value of `IN_REQ_SIG` and of `in_wakeup_sigs` should be the same.
`IN_Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Estimated Size

Two to 26 instructions if `MEM_TYPE` is `XFER`—that is, if data is transferred across microengines.

Two to 25 instructions if `MEM_TYPE` is `SRAM` or `SCRATCH`—that is, one instruction for each memory write to SRAM or scratch memory.

Microengine Assembler Example Usage

```
.sig SIG_MB
// Initialize a mailbox of SRAM type, 8 longwords in size
mailbox_init(base_addr, offset, sram, 8, SIG_MB, SIG_MB, ___)
```

Microengine C Example Usage

```
volatile __declspec(scratch) uint32_t *addr;
addr = 10;
ixp_mailbox_init(addr, SCRATCH, 1, SIG1, SIG1, ___);
// then you can call ixp_mailbox_send() or ixp_mailbox_receive()
uint32_t my_base = 0x200;
uint32_t in_wakeup_sigs;
SIGNAL sig1;
__declspec(sram_write_reg) xfer_t sram_wr_reg;
mem_t mem_type = SRAM;
mbox_t x;
//
x.sram_addr = (__declspec(sram) void*)my_base;
sram_wr_reg.x0 = 0x11;
sram_wr_reg.x1 = 0x12;
sram_wr_reg.x2 = 0x13;
sram_wr_reg.x3 = 0x14;
sram_wr_reg.x4 = 0x15;
sram_wr_reg.x5 = 0x16;
sram_wr_reg.x6 = 0x17;
sram_wr_reg.x7 = 0x18;
in_wakeup_sigs = __signals(&sig1);
ixp_sram_write (
       &sram_wr_reg,
       (volatile __declspec(sram) void*)my_base, 8,
       &sig1,
       in_wakeup_sigs,
       0);
   ixp_mailbox_init (
       x,
       mem_type,
       8,
       &sig1,
       in_wakeup_sigs,
       0);
__implicit_read(&sig1);
```

3.2.7.1.3 `mailbox_receive()`

Returns a message from a mailbox. The mailbox may be stored in scratch, SRAM, or another thread's transfer registers. The mailbox must first be initialized using `mailbox_init()` before calling this function. If the first 32-bit word's bit 31 is set to zero this indicates the message is valid.

Microengine Assembler Syntax

```
mailbox_receive (out_data, in_addr_or_thd, in_offset_or_xfer, \
   IN_MEM_TYPE, in_mb_size, IN_REQ_SIG, in_wakeup_sigs, IN_Q_OPTION);
```

Microengine C Syntax

```
INLINE uint32_t ixp_mailbox_receive(
   __declspec(sram_read_reg) uint32_t *out_data,
   mbox_t in_addr_or_thd,
   mem_t IN_MEM_TYPE,
   uint32_t in_mb_size,
   SIGNAL *IN_REQ_SIG,
   SIGNAL_MASK in_wakeup_sigs, queue_t IN_Q_OPTION);
```

Input

`in_addr_or_thd`	The mailbox address in bytes when `MEM_TYPE` is `SRAM` and `SCRATCH`. If `MEM_TYPE` is `XFER`, this parameter is not used.
`in_offset_or_xfer`	If `MEM_TYPE` is `SRAM` or `SCRATCH`, this is the offset to be added to `in_addr_or_thd`, the base mailbox address. If `MEM_TYPE` is `XFER`, this parameter is not used.
`IN_MEM_TYPE`	The type of memory for mailbox storage. This value is one of {`SRAM`, `SCRATCH`, `XFER`}. See Section 2.6.7, "mem_t."
`in_mb_size`	The mailbox size in 32-bit words. The minimum size is one. The maximum size is eight.
`IN_REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
`IN_Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Output/Returns

Return Value	If the mailbox is in `SRAM` or `SCRATCH` memory the return value indicates whether the mailbox has valid data—that is, the data are fresh—or if it does not—that is, the data are stale. • If this value is zero the mailbox is valid • If this value is one the mailbox is *not* valid If the mailbox is in `XFER` memory it always returns zero.
`out_data`	The mailbox contents.

Estimated Size

One instruction if `MEM_TYPE` is `XFER`.

Two to eight instructions if `MEM_TYPE` is `SRAM` or `SCRATCH`—one atomic SRAM or scratch memory access in the best case and one SRAM or scratch memory read and one atomic SRAM or scratch memory access in the worst case.

Microengine Assembler Example Usage

See example usage in `mailbox_send()` section.

Microengine C Example Usage

```
uint32_t my_base = 0x200;
uint32_t in_wakeup_sigs;
SIGNAL sig2;
mem_t mem_type = SRAM;
__declspec(sram_read_reg) uint32_t sram_rd_reg[8];
mbox_t x;
x.sram_addr = (__declspec(sram) void*)my_base;
in_wakeup_sigs = __signals(&sig2);
ixp_mailbox_receive (sram_rd_reg, x, mem_type, 8, &sig2, in_wakeup_sigs, 0);
__implicit_read (&sig2);
```

3.2.7.1.4 mailbox_send()

Sends a message to a mailbox. The mailbox may be scratch, SRAM, another thread's transfer registers, or the adjacent microengine's—that is, the current microengine plus one—next-neighbor registers. Bit 31 of the first 32-bit word is reserved for use as a valid bit. This bit is cleared when the mailbox `MEM_TYPE` is `SRAM` or `SCRATCH`.

Microengine Assembler Syntax

```
mailbox_send (in_first_data, in_data, in_addr_or_thd, \
    in_offset_or_xfer, IN_MEM_TYPE, in_mb_size, IN_REQ_SIG, \
    in_wakeup_sigs, IN_Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_mailbox_send(
    uint32_t in_first_data,
    declspec(sram_write_reg) uint32_t *in_data,
    mbox_t in_addr_or_thd,
    mem_t IN_MEM_TYPE,
    uint32_t in_mb_size,
    SIGNAL *IN_REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t IN_Q_OPTION );
```

Input

`in_first_data`	The data bits 30 through zero of the first 32-bit longword, plus bits 31 through zero of subsequent 32-bit longwords to write. This should be same as `in_data[0]`.
`in_data`	The 32-bit long data word to be sent.
`in_addr_or_thd`	The mailbox address in bytes for `IN_MEM_TYPE` of `SRAM` or `SCRATCH`. The thread number for `IN_MEM_TYPE` of `XFER` registers. The thread number should be a constant.
`in_offset_or_xfer`	For `MEM_TYPE` of `SRAM` or `SCRATCH`, this is the offset to be added to the mailbox base address. If `MEM_TYPE` is `XFER`, this is the read transfer register used for the mailbox at the destination thread.
`IN_MEM_TYPE`	The type of memory for mailbox storage. The value is one of {`SRAM`, `SCRATCH`, `XFER`}. See Section 2.6.7, "mem_t."
`in_mb_size`	The mailbox size in 32-bit words. • Minimum size is one • Maximum size is eight

Input (Continued)

`IN_REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage." **NOTE:** For a `MEM_TYPE` of `XFER`, the value of the arguments `IN_REQ_SIG` and `in_wakeup_sigs` should be the same.
`in_wakeup_sigs`	The list of signals causing the thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage." **NOTE:** For a `MEM_TYPE` of `XFER`, the value of the arguments `IN_REQ_SIG` and `in_wakeup_sigs` should be the same.
`IN_Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Estimated Size

One to two instructions if `MEM_TYPE` is `XFER`—that is, if data is transferred across microengines.

Two to five instructions if `MEM_TYPE` is `SRAM` or `SCRATCH`—one memory write to SRAM or scratch memory.

Microengine Assembler Example Usage One

```
.sig SIG_MB
// Send 4 longwords to a SRAM type mailbox
immed[first_data, 0x2]
immed[$wr_xfer0, 0x2]
immed[$wr_xfer1, 0x3]
immed[$wr_xfer2, 0x4]
immed[$wr_xfer3, 0x5]
mailbox_send(first_data, $wr_xfer0, base_addr, offset, sram, 4, \
   SIG_MB, SIG_MB, ___)
mailbox_receive($rd_xfer0, base_addr, offset, sram, 4, SIG_MB, SIG_MB, ___)
// $rd_xfer0 = 0x2 (bit31 = 0 to indicate data is valid)
// $rd_xfer1 = 0x3, $rd_xfer2 = 0x4, $rd_xfer3 = 0x5
```

Microengine Assembler Example Usage Two

```
// ME0 sends data; ME1 receives data
// ME0 codes:
.sig SIG_MB
.addr SIG_MB 3
.reg remote $remote_xfer0 $remote_xfer1
.xfer_order $remote_xfer0 $remote_xfer1
// Some code to init mailbox
immed[$wr_xfer0, 0xAA]
immed[$wr_xfer1, 0xBB]
// send data to thread 5 of ME0
mailbox_send(0x0, $wr_xfer0, 13, $remote_xfer0, xfer, 2, \
   SIG_MB, SIG_MB, ___)
// ME1 codes:
.sig SIG_MB
.addr SIG_MB 3
.reg visible global $remote_xfer0 $remote_xfer1
.xfer_order $remote_xfer0 $remote_xfer1
// wait for data to be sent from ME0
mailbox_receive($remote_xfer0, 0, 0, xfer, 2, SIG_MB, SIG_MB, ___)
// check data received
// $remote_xfer0 = 0xAA, $remote_xfer1 = 0xBB
```

Microengine C Example Usage

```
uint32_t my_base = 0x200;
uint32_t in_wakeup_sigs, in_first_data;
SIGNAL sig1, sig2;
mem_t mem_type = SRAM;
__declspec(sram_write_reg) uint32_t sram_wr_reg[8];
mbox_t x;
//
x.sram_addr = (__declspec(sram) void*)my_base;
in_first_data = 0x11;
sram_wr_reg[0] = 0x11;
sram_wr_reg[1] = 0x12;
sram_wr_reg[2] = 0x13;
sram_wr_reg[3] = 0x14;
sram_wr_reg[4] = 0x15;
sram_wr_reg[5] = 0x16;
sram_wr_reg[6] = 0x17;
sram_wr_reg[7] = 0x18;
//
in_wakeup_sigs = __signals(&sig1);
ixp_mailbox_init (x, mem_type, 8, &sig1, in_wakeup_sigs, 0);
__implicit_read(&sig1);
//
in_wakeup_sigs = __signals(&sig2);
if (!ixp_mailbox_is_full (x, mem_type))
    ixp_mailbox_send (
        in_first_data,
        sram_wr_reg,
        x,
        mem_type,
        8,
        &sig2,
        in_wakeup_sigs,
        0);
__implicit_read(&sig2);
```

3.2.8 msgq

The msgq interface supports the queuing of messages in a circular first-in, first-out (FIFO) array also called a message queue (*msgq*). The circular array and indexes of the queue are initialized by msgq_init(). A microengine thread sends a message to the *msgq* using msgq_send(). Another microengine thread receives the message at the head of the *msgq* using msgq_receive(). The supported values of msg_size * q_size include 512, 1024, 2048, or 4096 bytes when specifying the size of scratch memory and 512, 1024, 2048, 4096, 8192, 16384, 32768 or 65536 32-bit words when specifying the size of SRAM memory. Also, the in_addr input parameter has to be aligned on the memory space.

The message structure is defined by the calling application and is common to both the sender—producer—and receiver—consumer. Bit 31 of the first 32-bit word of each message is reserved for a message valid flag.

Table 3-9. msgq API

Name	Description
msgq_init()	Initializes a circular message queue.
msgq_receive()	Receives a message from a queue.
msgq_send()	Sends a message to a queue.

3.2.8.1 API Functions

3.2.8.1.1 msgq_init()

Initializes a message queue. The message queue memory type may be SCRATCH or SRAM. There is one sender (producer) and one receiver (consumer). The sender must call msgq_init() before sending any messages. All messages in the queue are initialized to zero. The memory space for the *msgq* must be pre-allocated by the application.

Use the following equation to determine the amount of SRAM memory space to allocate:

```
size = msg_size * q_size
```

Use the following equation to determine the amount of scratch memory space to allocate:

```
size = msg_size * q_size * 4
```

Microengine Assembler Syntax

```
msgq_init (in_addr, in_ring_id_or_index, IN_MEM_TYPE, IN_q_size, \
    in_msg_size, IN_REQ_SIG, in_wakeup_sigs, IN_Q_OPTION);
```

Microengine C Syntax

```
void ixp_msgq_init(
    volatile void *in_addr,
    IMMED IN_RING_ID_OR_INDEX,
    mem_t IN_MEM_TYPE,
    IMMED IN_Q_SIZE,
    IMMED IN_MSG_SIZE,
    SIGNAL* IN_REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t IN_Q_OPTION);
```

Input

`in_addr`	The *msgq* address in bytes—a constant or GPR. The memory address must always be byte aligned. • If `IM_MEM_TYPE` is `SCRATCH` the value is one of {512, 1024, 2048, or 4096} • If `IM_MEM_TYPE` is `SRAM` the value is one of {512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536}
`IN_RING_ID_OR_INDEX`	This is a constant index into the msgq array. If `IN_MEM_TYPE` is `SCRATCH` the value ranges from zero to fifteen. If `IN_MEM_TYPE` is `SRAM` the value ranges from zero to 63.
`IN_MEM_TYPE`	The type of memory for *msgq* storage. This should be one of {`SRAM` or `SCRATCH`}. See Section 2.6.7, “mem_t.”
`IN_Q_SIZE`	The number of entries in queue, a constant.
`IN_MSG_SIZE`	The message size, a constant number of 32-bit words. Maximum message size is eight.
`IN_REQ_SIG`	The requested signal. See Section 2.6.13, “Signal Arguments and Usage.”
`in_wakeup_sigs`	The list of signals causing the thread to swap or wakeup. See Section 2.6.13, “Signal Arguments and Usage.”
`IN_Q_OPTION`	`queue_t`—the directive for memory controller queue selection. See Section 2.6.10, “queue_t.”

Estimated Size

Six to eight instructions if `MEM_TYPE` is `SCRATCH`—three control status register writes.

Eleven to twenty instructions if `MEM_TYPE` is `SRAM`—one SRAM write and two SRAM read ring operations.

Microengine Assembler Example Usage

```
.sig MY_SIG
#define_eval RING_NUMBER 4
// Init a message queue of SRAM type which has
// 128 messages and 8 longwords each
    immed(base_addr, 2048);
msgq_init(base_addr, RING_NUMBER, SRAM, 128, 8, MY_SIG, MY_SIG, ___);
```

Microengine C Example Usage

```
uint32_t my_base = 512;
mem_t mem_type = SCRATCH;
SIGNAL sig1;
SIGNAL_MASK wakeup_sig1;
// Initialize local variables
wakeup_sig1 = __signals(&sig1);
ixp_msgq_init(
    (volatile void *)my_base,
    0,
    mem_type,
    SCRATCH_RING_SIZE_128,
    &sig1,
    wakeup_sig1,
    0);
__implicit_read(&sig1);
// Now you can call ixp_msgq_send() or ixp_msgq_receive()
```

3.2.8.1.2 msgq_receive()

Returns a message from a *msgq*. The mailbox may be in scratch or SRAM memory. The *msgq* must first be initialized using `msgq_init()` before calling this function. If `out_data` is greater than zero then `out_data` contains a new message from the *msgq*. If `out_data` is zero then no message is available from the *msgq*.

Microengine Assembler Syntax

```
msgq_receive (out_data, in_addr, in_ring_id_or_index, IN_MEM_TYPE, \
   in_q_size, in_msg_size, IN_REQ_SIG, in_wakeup_sigs, IN_Q_OPTION);
```

Microengine C Syntax

```
ixp_msgq_receive(
   __declspec(sram_read_reg) void *out_data,
   volatile void *in_addr,
   IMMED IN_RING_ID_OR_INDEX,
   mem_t IN_MEM_TYPE,
   IMMED IN_Q_SIZE,
   IMMED IN_MSG_SIZE,
   SIGNAL *IN_REQ_SIG,
   SIGNAL_MASK in_wakeup_sigs,
   queue_t IN_Q_OPTION);
```

Input

`in_addr`	The *msgq* address in bytes—constant or GPR. The memory address must always be byte aligned. For `IN_MEM_TYPE` of `SCRATCH`: 512, 1024, 2048, or 4096. For `IN_MEM_TYPE` of `SRAM`: 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536. **NOTE:** Bits [31:30] of `in_addr` indicates the SRAM channel. If `ADDRESS_MODE` is `IXP2XXX`, this is a byte address—a constant or GPR.
`IN_RING_ID_OR_INDEX`	This is a constant index into the msgq array. This value ranges from zero to fifteen for `IN_MEM_TYPE` of `SCRATCH` and zero to 63 for `IN_MEM_TYPE` of `SRAM`.
`IN_MEM_TYPE`	The type of memory for *msgq* storage. This value is one of {`SRAM` or `SCRATCH`}. See Section 2.6.7, “mem_t.”
`IN_Q_SIZE`	The constant number of entries in the queue, in longwords.
`IN_MSG_SIZE`	The message size, a constant number of 32-bit words. Maximum message size is eight.

Input (Continued)

`IN_REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
`IN_Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Output

`out_data`	The message data. • If `out_data` is greater than zero this argument contains a new message from the *msgq* • If `out_data` is zero there is no message available from the *msgq*

Estimated Size

Two to seven instructions—one ring operation for SRAM or scratch.

Microengine Assembler Example Usage

See example usage in `msgq_send()` section.

Microengine C Example Usage

```
#define SCRATCH_RING_SIZE_128_116, 8
__declspec(sram_read_reg) uint32_t readreg[8];
__declspec(sram_write_reg) uint32_t xfer[8];
uint32_t my_base = 512, status;
mem_t mem_type = SCRATCH;
SIGNAL sig1, sig2, sig3;
SIGNAL_MASK wakeup_sig1, wakeup_sig2, wakeup_sig3;
SIGNAL_PAIR s1;
xfer[0] = 1;
xfer[1] = 2;
xfer[2] = 3;
xfer[3] = 4;
xfer[4] = 5;
xfer[5] = 6;
xfer[6] = 7;
xfer[7] = 8;
//
wakeup_sig1 = __signals(&sig1);
ixp_msgq_init(
       (volatile void *)my_base,
       3,
       mem_type,
       SCRATCH_RING_SIZE_128_1,
       &sig1,
       wakeup_sig1,
       0);
__implicit_read(&sig1);
//
wakeup_sig2 = __signals(&sig2);
ixp_msgq_send (
       &readreg[0],
       &xfer[0],
       (volatile void *)my_base,
       3,
       mem_type,
       SCRATCH_RING_SIZE_128_1,
       &sig2,
       wakeup_sig2,
       0);
__implicit_read(&sig2);
//
wakeup_sig3 = __signals(&sig3);
ixp_msgq_receive (
       &readreg[0],
       (volatile void *)my_base,
       3,
       mem_type,
       SCRATCH_RING_SIZE_128_1,
       &sig3,
       wakeup_sig3,
       0);
```

3.2.8.1.3 `msgq_send()`

Sends a message to a *msgq*. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the mailbox may be in scratch or SRAM memory. The *msgq* must first be initialized using `msgq_init()` before calling this function. Bit 31 of the first 32-bit word of `io_data` indicates memory status if `MEM_TYPE` is `SRAM`:

- One—the message was successfully sent
- Zero—the message was not sent because the *msgq* is full

Sets the valid bit of the message before calling this operation. This function writes the message to *msgq* at `in_index`.

Microengine Assembler Syntax

```
msgq_send (io_data, in_addr, in_index,
    MEM_TYPE, q_size, msg_size, REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
void ixp_msgq_send(__declspec(
    sram_read_reg) uint32_t *out_status,
    declspec(sram_write_reg) uint32_t *in_data,
    volatile void *in_addr,
    IMMED IN_RING_ID_OR_INDEX,
    mem_t IN_MEM_TYPE,
    IMMED IN_Q_SIZE,
    IMMED IN_MSG_SIZE,
    void *IN_REQ_SIG_OR_SIGPAIR,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t IN_Q_OPTION);
```

Input

`in_addr`	The msgq address in bytes—a constant or GPR. The memory address must always be byte aligned. • If `IN_MEM_TYPE` is `SCRATCH`: 512, 1024, 2048, or 4096 • IF `IN_MEM_TYPE` is `SRAM`: 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536 **NOTE:** Bits [31:30] of `in_addr` indicates SRAM channel.
`IN_RING_ID_OR_INDEX`	This is a constant index into the *msgq* array. The value ranges from zero to fifteen for `IN_MEM_TYPE` of `SCRATCH` and zero to 63 for `IN_MEM_TYPE` of `SRAM`.
`IN_MEM_TYPE`	The type of memory for *msgq* storage. This value is one of {`SRAM` or `SCRATCH`}. See Section 2.6.7, "mem_t."
`IN_Q_SIZE`	The constant number of entries in the queue, expressed in longwords.

Input (Continued)

`IN_MSG_SIZE`	The message size, a constant number of 32-bit words. Maximum message size is eight.
`IN_REQ_SIG`	The requested signal. See Section 2.6.13, “Signal Arguments and Usage.” The argument should be of type: • `SIGNAL`—for `IN_MEM_TYPE` of `SCRATCH` • `SIGNAL_PAIR`—for `IN_MEM_TYPE` of `SRAM`
`in_wakeup_sigs`	The list of signals causing thread to swap or wakeup. See Section 2.6.13, “Signal Arguments and Usage.”
`IN_Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, “queue_t.”

Input/Output

`io_data` *or* `in_data`	The message data to be sent.

Output

`io_data` *or* `out_status`	The resulting status when `MEM_TYPE` equals `SRAM` only. Bit 31 values: • `1`—the message was successfully sent to the *msgq* • `0`—the *msgq* is full and the message is not sent

Estimated Size

Two to seven instructions—one ring operation for SRAM or scratch memory.

Microengine Assembler Example Usage One

```
.sig MY_SIG
#define_eval ringId  10
// send/receive messages of SCRATCH type: 16 messages of 8 longwords each
immed[base_addr, 512]
msgq_send($wr_xfer0, base_addr, ringId, SCRATCH, 16, 8, MY_SIG, MY_SIG, ___)
msgq_receive($rd_xfer0, base_addr, ringId, SCRATCH, 16, 8, \
    MY_SIG, MY_SIG, ___)
```

Microengine Assembler Example Usage Two

```
.sig MY_SIG
#define_eval Qentry  30
```

```
// send/receive messages of SRAM type: 512 messages of 8 longwords each
immed[base_addr, 0x4000]
msgq_send($wr_xfer0, base_addr, Qentry, SRAM, 512, 8, MY_SIG, MY_SIG, ___)
msgq_receive($rd_xfer0, base_addr, Qentry, SRAM, 512, 8, MY_SIG, MY_SIG, ___)
```

Microengine C Example Usage One

```
__declspec(sram_read_reg) uint32_t readreg[8];
__declspec(sram_write_reg) uint32_t xfer[8];
uint32_t my_base = 4096, i;
mem_t mem_type = SRAM;
SIGNAL sig1;
SIGNAL_PAIR sig2;
SIGNAL_MASK wakeup_sig1, wakeup_sig2;
xfer[0] = 1;
xfer[1] = 2;
xfer[2] = 3;
xfer[3] = 4;
xfer[4] = 5;
xfer[5] = 6;
xfer[6] = 7;
xfer[7] = 8;
//
wakeup_sig1 = __signals(&sig1);
ixp_msgq_init((volatile void *)my_base, 0, mem_type, 128, 8, &sig1,
    wakeup_sig1, 0);
__implicit_read(&sig1);
//
for (i=0; i<10; i++) {
    wakeup_sig2 = __signals(&sig2);
    ixp_msgq_send (&readreg[0], &xfer[0], (volatile void *)my_base, 0,
        mem_type, 128, 8, &sig2, wakeup_sig2, 0);
    __implicit_read(&sig2);
    }
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg) uint32_t readreg[8];
__declspec(sram_write_reg) uint32_t xfer[8];
uint32_t my_base = 512;
mem_t mem_type = SCRATCH;
SIGNAL sig1, sig2;
SIGNAL_MASK wakeup_sig1, wakeup_sig2;
xfer[0] = 1;
xfer[1] = 2;
xfer[2] = 3;
xfer[3] = 4;
xfer[4] = 5;
xfer[5] = 6;
xfer[6] = 7;
xfer[7] = 8;
//
```

```
wakeup_sig1 = __signals(&sig1);
ixp_msgq_init((volatile void *)my_base, 3, mem_type, 16, 8, &sig1,
   wakeup_sig1, 0);
__implicit_read(&sig1);
//
//
wakeup_sig2 = __signals(&sig2);
ixp_msgq_send (&readreg[0], &xfer[0], (volatile void *)my_base, 3, mem_type,
   16, 8, &sig2, wakeup_sig2, 0);
__implicit_read(&sig2);
```

3.2.9 `rbuf`

The `rbuf` interface can be used for pipelined RBUF reads. RBUF is the interface buffer for data received from the network. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Note: In documentation for the Intel® IXP1200 Network Processor this is called the receive RFIFO.

Table 3-10. `rbuf` API

Name	Description
`rbuf_addr_from_elem()`	Converts RBUF element index to RBUF address.
`rbuf_elem_from_addr()`	Converts RBUF address to RBUF element index.
`rbuf_elem_free()`	Frees an RBUF element so it can be reused.
`rbuf_read()`	Reads `in_qw_count` quadwords from RBUF.

3.2.9.1 API Functions

3.2.9.1.1 `rbuf_addr_from_elem()`

Converts an RBUF element index to an RBUF address calculated using the following:

```
out_rbuf_addr = in_rbuf_elem  * ELEMENT_SIZE
```

Microengine Assembler Syntax

```
rbuf_addr_from_elem (out_rbuf_addr, in_rbuf_elem, ELEMENT_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(rbuf) void* ixp_rbuf_addr_from_elem (
    uint32_t in_rbuf_elem, netif_elem_size_t ELEMENT_SIZE);
```

Input

`in_rbuf_elem`	The RBUF element index.
`ELEMENT_SIZE`	The size of RBUF elements in bytes—either 64, 128, or 256. This value is one of {`NETIF_ELEM_64`, `NETIF_ELEM_128`, `NETIF_ELEM_256`}.

Output/Returns

`out_rbuf_addr` *or* Return Value	The resulting RBUF address. For an `ADDRESS_MODE` of `IXP2XXX` this is a byte address.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg out_addr
rbuf_addr_from_elem(out_addr, 8, NETIF_ELEM_64);
```

Microengine C Example Usage

```
__declspec(rbuf) void* rbuf_addr;
uint32_t rbuf_element = 4;
//
rbuf_addr = ixp_rbuf_addr_from_elem(rbuf_element, NETIF_ELEM_64);
```

3.2.9.1.2 `rbuf_elem_free()`

Frees an RBUF element so it can be reused. In the Intel® IXP2400, IXP2800 and IXP2850 Network Processors only, the on-chip media switch fabric hardware manages the elements and makes them available for use at reset or when microengine threads free them. Elements are consumed when data is received from the network. Refer to one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
rbuf_elem_free (in_rbuf_elem);
```

Microengine C Syntax

```
INLINE void ixp_rbuf_elem_free (uint32_t in_rbuf_elem);
```

Input

`in_rbuf_elem`	The index of the RBUF element to be freed.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
rbuf_elem_free (1);
```

Microengine C Example Usage

```
ixp_rbuf_elem_free (1);
```

3.2.9.1.3 `rbuf_elem_from_addr()`

Converts an RBUF address to an element index using the following calculation:

```
out_index = in_rbuf_addr / ELEMENT_SIZE
```

Microengine Assembler Syntax

```
rbuf_elem_from_addr (out_rbuf_elem, in_rbuf_addr, ELEMENT_SIZE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_rbuf_elem_from_addr (
    void __declspec(rbuf) *in_rbuf_addr,
    netif_elem_size_t ELEMENT_SIZE);
```

Input

`in_rbuf_addr`	The RBUF element address whose index is needed. This is a byte address.
`ELEMENT_SIZE`	The size of RBUF elements in bytes—either 64, 128, or 256 bytes. This value is one of {`NETIF_ELEM_64`, `NETIF_ELEM_128`, `NETIF_ELEM_256`}.

Output/Returns

`out_rbuf_elem` ***or*** Return Value	The resulting RBUF element index.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg in_addr, out_elem
move(in_addr, 0x100);
rbuf_elem_from_addr(out_elem, in_addr, NETIF_ELEM_64);
```

Microengine C Example Usage

```
__declspec(rbuf) void* rbuf_addr = 0x100;
uint32_t rbuf_element;
rbuf_element = ixp_rbuf_elem_from_addr(rbuf_addr, NETIF_ELEM_64);
```

3.2.9.1.4 `rbuf_read()`

Read `in_qw_count` quadwords from receive port buffer, `RBUF`.

Microengine Assembler Syntax

```
rbuf_read(out_data,in_rbuf_addr, in_rbuf_addr_offset, in_qw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_rbuf_read(
    void *out_data,
    volatile void __declspec(rbuf) *in_rbuf_addr,
    uint32_t in_qw_count,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_rbuf_addr`	The RBUF address. For an `ADDRESS_MODE` of `IXP2XXX` this is a byte address.
`in_rbuf_addr_offset`	For *Microengine Assembler only*, this is the offset to be added to `in_rbuf_addr` to form the address to be accessed.
`in_qw_count`	The register or constant 64-bit quadword count. The minimum quadword count is one. The maximum quadword count is 16.
`REQ_SIG`	The requested signal. See Section 2.6.13, “Signal Arguments and Usage.”
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, “Signal Arguments and Usage.”
`Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, “queue_t.”

Output

`out_data`	The data read from the RBUF. The caller declares `out_data` with the understanding that it is assigned to read transfer registers. The maximum size of `out_data` is 16 quadwords—128 bytes.

Estimated Size

One instruction if `in_qw_count` is an immediate value.

Three instructions if `in_qw_count` is not an immediate value.

Microengine Assembler Example Usage

```
#define_eval TEST_RBUF_ADDR 0x2000
#define_eval TEST_RBUF_OFFSET 0x50
.reg $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5 $xfer6 $xfer7
.reg in_addr offset
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5 $xfer6 $xfer7
.sig tmp_sig1

move(in_addr, TEST_RBUF_ADDR)
alu[offset, --, b, TEST_RBUF_OFFSET]
rbuf_read($xfer0, in_addr, offset, 4, tmp_sig1, tmp_sig1, ___);
```

Microengine C Example Usage

```
__declspec(sram_read_reg) uint32_t data[4];
SIGNAL sig_rbuf;
ixp_rbuf_read(data, addr, 2, &sig_rbuf, __signals(&sig_rbuf), ___);
```

3.2.10 scratch

The `scratch` interface can be used for concurrent scratch memory access, multiword reads and writes, and scratch operations that perform more than just simple read or write access.

Table 3-11. `scratch` API

Name	Description
`scratch_add()`	Adds `in_data` to the longword at a scratch location.
`scratch_bits_clr()`	Clears `in_mask` bits at the location `in_scratch_addr`.
`scratch_bits_set()`	Sets `in_mask` bits at location `in_scratch_addr`.
`scratch_bits_test_and_clr()`	Reads location `in_scratch_addr`, then clears `in_mask` bits.
`scratch_bits_test_and_set()`	Reads location `in_scratch_addr`, then sets `in_mask` bits.
`scratch_decr()`	Decrements the longword at a scratch location.
`scratch_incr()`	Increments the value at location `in_scratch_addr`.
`scratch_read()`	Reads `in_lw_count` longwords from location `in_scratch_addr`.
`scratch_test_and_add()`	Reads location `in_scratch_addr`, then adds `in_data` to the longword at the scratch location.
`scratch_test_and_decr()`	Reads then decrements a longword at a scratch location.
`scratch_test_and_incr()`	Reads then increments a longword at a scratch location.
`scratch_write()`	Writes `in_lw_count` longwords to location `in_scratch_addr`.
`scratch_sub()` See Section 3.2.10.2.12, "scratch_sub()."	Subtracts `in_data` from the value at the location `in_scratch_addr`.
`scratch_swap()`	Reads a scratch location and then writes `in_data` to that scratch location.

3.2.10.1 Common Arguments

The section describes the arguments used throughout scratch access operations.

Input

`in_data`	The value to be used in `scratch_add()`, `scratch_sub()`, or `scratch_swap()` operation.
`in_mask`	The 32-bit mask of bits to set or clear.
`in_scratch_addr`	The scratch address. If `ADDRESS_MODE` is `IXP2XXX` this is a byte address.
`in_addr_offset`	For *Microengine Assembler only*, this offset is added to `in_scratch_addr` to form the address to be accessed.
`REQ_SIG`	The requested signal. See Section 2.6.13, “Signal Arguments and Usage.”
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, “Signal Arguments and Usage.”
`Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, “queue_t.”

Output

`out_data`	The scratch location value prior to execution of the operation.

3.2.10.2 API Functions

3.2.10.2.1 `scratch_bits_clr()`

Clears `in_mask` bits at location `in_scratch_addr`. A one in any `in_mask` bit position clears the corresponding bit position at the location `in_scratch_addr`.

```
*in_scratch_addr &= ~in_mask;
```

Microengine Assembler Syntax

```
scratch_bits_clr (in_mask, in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_ bits_clr(
    uint32_t in_mask,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.sig sig1
move(addr, 0x100);
// clear left 4 bits and right 4 bits
scratch_bits_clr(0xf000000f, addr, 0, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
move(addr,0x100);
// clear left 4 bits and right 4 bits
scratch_bits_clr(0xf000000f, addr, 0, sig1, SIG_NONE, ___);
//
// other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(scratch) uint32_t *addr;
addr = 10;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
// clear left 4 bits and right 4 bits
ixp_scratch_bits_clr(0xf000000f, addr, &sig, mask, 0);
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
__declspec(scratch) uint32_t *addr;
addr = 10;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
// clear left 4 bits and right 4 bits
ixp_scratch_bits_clr(0xf000000f, addr, &sig, 0, 0);
       //
       // other code
       //
wait_for_all(sig_mask);
```

3.2.10.2.2 `scratch_bits_set()`

Sets `in_mask` bits at location `in_scratch_addr`. A one in any `in_mask` bit position sets the corresponding bit position at the location `in_scratch_addr`. This is equivalent to the expression: `*in_scratch_addr |= in_mask;`

Microengine Assembler Syntax

```
scratch_bits_set (in_mask, in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_ bits_set(
    uint32_t in_mask,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Usage Example One

```
.reg mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_set(mask, addr, offset, scratch_sig, scratch_sig, ___);
```

Microengine Assembler Usage Example Two

```
.reg mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_set(mask, addr, offset, scratch_sig, ___, ___);
ctx_swap[scratch_sig]
```

Microengine C Example Usage Example One

```
__declspec(scratch) uint32_t *addr;
addr = 10;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
// set left 4 bits and right 4 bits
ixp_scratch_bits_set(0xf000000f, addr, &sig, sig_mask, 0);
__implicit_read(&sig);
```

Microengine C Example Usage Example Two

```
__declspec(scratch) uint32_t *addr;
addr = 10;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
// set left 4 bits and right 4 bits
ixp_scratch_bits_set(0xf000000f, addr, &sig, 0, 0);
       //
       // other code
wait_for_all(sig_mask);
```

3.2.10.2.3 `scratch_bits_test_and_clr()`

Reads a scratch location into `out_data` then clears `in_mask` bits at the same scratch location. This is performed as a single atomic memory operation—that is, with no possibility of any other access to that location from the time of the *read* to the time of the *write*. A one in any `in_mask` bit position clears the corresponding bit position at the location `in_scratch_addr`.

Microengine Assembler Syntax

```
scratch_bits_test_and_clr (out_data, in_mask,
    in_scratch_addr, in_addr_offset,
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_ bits_test_and_clr(
    __declspec(sram_read_reg) uint32_t *out_data,
    __declspec(sram_write_reg) uint32_t *in_mask,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.reg $data mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_test_and_clr($data, mask, addr, offset, scratch_sig, \
        scratch_sig, ___);
```

Microengine Assembler Example Usage Two

```
.reg $data mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_test_and_clr($data, mask, addr, offset, scratch_sig, ___, ___);
ctx_swap[scratch_sig]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg) uint32_t data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
// get data, then clear left 4 bits and right 4 bits
ixp_scratch_bits_test_and_clr(&data, 0xf000000f, addr,
&sig_pr, sig_mask, 0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg) uint32_t data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
// get data, then clear left 4 bits and right 4 bits
ixp_scratch_bits_test_and_clr(&data, 0xf000000f, addr,
&sig_pr, SIG_NONE, 0);
       // other code
wait_for_all(sig_mask);
```

3.2.10.2.4 `scratch_bits_test_and_set()`

Reads a scratch location into `out_data` then sets `in_mask` bits at the same scratch location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*. A one in any `in_mask` bit position sets the corresponding bit position at the location `in_scratch_addr`.

Microengine Assembler Syntax

```
scratch_bits_test_and_set (out_data, in_mask,
    in_scratch_addr, in_addr_offset,
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE uint32_t ixp_scratch_ bits_test_and_set(
    __declspec(sram_read_reg)uint32_t *out_data,
    __declspec(sram_write_reg) uint32_t *in_mask,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.reg $data mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_test_and_set($data, mask, addr, offset, \
        scratch_sig, scratch_sig, ___);
```

Microengine Assembler Example Usage Two

```
.reg $data mask
.sig scratch_sig
immed32(mask, 0xf000000f);
scratch_bits_test_and_set($data, mask, addr, offset, scratch_sig, ___, ___);
ctx_swap[scratch_sig]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg) uint32_t data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
// get data, then set left 4 bits and right 4 bits
ixp_scratch_bits_test_and_set(&data, 0xf000000f, addr,
&sig_pr, sig_mask, 0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
uint32_t data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
// get data, then set left 4 bits and right 4 bits
ixp_scratch_bits_test_and_set(&data, 0xf000000f, addr,
&sig_pr, SIG_NONE, ___);
       // other code
wait_for_all(sig_mask);
```

3.2.10.2.5 `scratch_incr()`

Increments the value at the location `in_scratch_addr`.

Note: If there is a carry out, as after incrementing `0xFFFFFFFF`, the resulting value at `in_scratch_addr` is zero.

Microengine Assembler Syntax

```
scratch_incr (in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_incr(
    volatile __declspec(scratch) void* in_scratch_addr);
```

Inputs and Outputs

See Section 3.2.10.1, "Common Arguments" for argument descriptions.

Estimated Size

One or two instructions.

Microengine Assembler Example Usage

```
.reg address offset
move(address, 0x1000);
move(offset, 10);
scratch_incr(address, offset);
```

Microengine C Example Usage

```
__declspec(scratch) uint32_t *addr;
addr = 10;
ixp_scratch_incr(addr);
```

3.2.10.2.6 `scratch_read()`

Reads `in_lw_count` 32-bit longwords from location `in_scratch_addr`.

Microengine Assembler Syntax

```
scratch_read (out_data, in_scratch_addr, in_addr_offset, in_lw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_read(
    __declspec(sram_read_reg)void *out_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    uint32_t in_lw_count,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

`in_lw_count`	The number of 32-bit longwords to read. The minimum longword count is one. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the maximum longword count is 16. See Section 3.2.10.1, “Common Arguments” for other argument descriptions.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage One

```
.xfer_order $data0 $data1 $data2
.sig sig1
move(addr, 0x100);
scratch_read($data0, addr, 0, 3, sig1, sig1, ___); // read 3 32 bit words
```

Microengine Assembler Example Usage Two

```
.xfer_order $data0 $data1 $data2
.sig sig1
move(addr,0x100);
scratch_read($data, addr, 0, 3, sig1, ___, ___);// read 3 32 bit words
//
// some other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_Read_reg) uint32_t data[3];
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
ixp_scratch_read(     // read 3 32 bit words
    &data, addr,
    3,
    &sig, sig_mask,
    0);
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg) uint32_t data[3];
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
// read 3 32 bit words
ixp_scratch_read(&data, addr, 3, &sig, SIG_NONE, 0);
//
// some other code
//
wait_for_all(sig_mask);
```

3.2.10.2.7 `scratch_write()`

Writes `in_lw_count` 32-bit longwords to location `in_scratch_addr`.

Microengine Assembler Syntax

```
scratch_write (in_data, in_scratch_addr, in_, in_addr_offset, lw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_write(
    __declspec(sram_write_reg)void *in_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    uint32_t in_lw_count,
    SIGNAL* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

`in_lw_count`	The number of 32-bit longwords to write. The minimum longword count is one. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the maximum longword count is 16. See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage One

```
.xfer_order $data0 data1;
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
// write 2 32 bit words
scratch_write($data0, addr, 0, 2, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.xfer_order $data0 data1;
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
// write 2 32 bit words
scratch_write($data0, addr, 0, 2, sig1, ___, ___);
// ... some other code
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg) uint32_t data[2];
data[0] = 0x01234567;
data[1] = 0x89ABCDEF;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
// write 2 32 bit words
ixp_scratch_write(&data, addr, 2, &sig, sig_mask, 0);
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
uint32_t data[2];
data[0] = 0x01234567;
data[1] = 0x89ABCDEF;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
// write 2 32 bit words
ixp_scratch_write(&data, addr, 2, &sig, SIG_NONE, 0);
// ... some other code
wait_for_all(sig_mask);
```

3.2.10.2.8 `scratch_add()`

Adds `in_data` to the value at the location `in_scratch_addr`. This operation is equivalent to the expression:

```
*in_scratch_addr += in_data;
```

Note: There is no carry or overflow indication. The value wraps through zero.

Microengine Assembler Syntax

```
scratch_add (in_data, in_scratch_addr, in_addr_offset, \
   REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_ add(
   uint32_t in_data,
   volatile __declspec(scratch) void* in_scratch_addr,
   SIGNAL *REQ_SIG,
   SIGNAL_MASK in_wakeup_sigs,
   queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.reg $data
.sig sig1
immed32($data, 0x2);
scratch_add($data, address, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $data
.sig sig1
immed32($data, 0x2);
scratch_add($data, address, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
// clear left 4 bits and right 4 bits
ixp_scratch_add(0x2, addr, &sig, sig_mask, 0);    // add 2
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
addr = 10;
// clear left 4 bits and right 4 bits
ixp_scratch_add(0x2, addr, &sig, SIG_NONE, 0);    // add 2
__wait_for_all(&sig);
```

3.2.10.2.9 `scratch_decr()`

Decrements the value at the location `in_scratch_addr`.

Note: A decrement of the value zero results in zero.

Microengine Assembler Syntax

```
scratch_decr (in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_decr(
    volatile __declspec(scratch) void* in_scratch_addr);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One or two instructions.

Microengine Assembler Example Usage

```
.reg address offset
move(address, 0x1000);
move(offset, 10);
scratch_decr(address, offset);
```

Microengine C Example Usage

```
__declspec(scratch) uint32_t *addr;
addr = 10;
ixp_scratch_decr(addr);
```

3.2.10.2.10 `scratch_test_and_add()`

Reads a scratch location and then adds `in_data` to the value at the scratch location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Note: There is no carry or overflow indication. The value wraps through zero.

Microengine Assembler Syntax

```
scratch_test_and_add (out_data, in_data, in_scratch_addr, \
    in_addr_offset,REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_test_and_add(
    __declspec(sram_read_reg)uint32_t *out_data,
    __declspec(sram_write_reg)uint32_t *in_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL_PAIR* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_data`	A constant, a general-purpose register, or a read transfer register.
`in_scratch_addr`	A register or constant-base longword address.
`in_addr_offset`	A register or constant longword address offset.

Note: If `in_data` and `out_data` each use the pair of read/write transfer registers with the same name—for example, `$buffer0`—the data from the write transfer register—`$buffer0.write`, in this example—is written to `in_scratch_addr + in_addr_offset`. The data from `in_scratch_addr + in_addr_offset` is returned in the read transfer register—in this example, `$buffer0.read`.

Output/Returns

`out_data`	A read/write transfer register pair. The result is returned in the read part of the read/write transfer register pair.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.reg $out_data in_data
.sig scratch_sig
scratch_test_and_add($out_data, in_data, addr, offset, \
       scratch_sig, scratch_sig, ___);
```

Microengine Assembler Example Usage Two

```
.reg $out_data in_data
.sig scratch_sig
scratch_test_and_add($out_data, in_data, addr, offset, scratch_sig, ___, ___);
ctx_swap[scratch_sig]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg) uint32_t in_data = 2;
__declspec(sram_read_Reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
ixp_scratch_test_and_add(&out_data, &in_data, addr,
    &sig_pr, sig_mask, 0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_write_reg) uint32_t in_data = 2;
__declspec(sram_read_Reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR sig_pr;
addr = 10;
ixp_scratch_test_and_add(&out_data, &in_data, addr,
    &sig_pr, SIG_NONE, 0);
__wait_for_all(&sig_pr);
```

3.2.10.2.11 `scratch_swap()`

Reads a scratch location and then writes `in_data` to that scratch location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Microengine Assembler Syntax

```
scratch_swap (out_data, in_data, in_scratch_addr, in_addr_offset,
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_swap(
    __declspec(sram_read_reg) uint32_t *out_data,
    __declspec(sram_write_reg) uint32_t *in_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.reg $out_data in_data addr offset
.sig sig1
move(addr, 0x100);
move(offset, 0);
scratch_swap($out_data, in_data, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $out_data in_data addr offset
.sig sig1
move(addr, 0x100);
move(offset, 0);
scratch_swap($out_data, in_data, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg) uint32_t in_data = 2;
__declspec(sram_read_reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR *sig_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig_pr);
addr = 10;
ixp_scratch_swap(&out_data, &in_data, addr, &sig_pr, sig_mask, 0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_write_reg) uint32_t in_data = 2;
__declspec(sram_read_reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL_PAIR *sig_pr;
addr = 10;
ixp_scratch_swap(&out_data, &in_data, addr, &sig_pr, SIG_NONE, 0);
__wait_for_all(&sig_pr);
```

3.2.10.2.12 `scratch_sub()`

Subtracts `in_data` from the value at the location `in_scratch_addr`.

Note: If `in_data` is greater than the value at `in_scratch_addr`, the result is zero.

```
*in_scratch_addr -= in_data;
```

Microengine Assembler Syntax

```
scratch_sub (in_data, in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_ sub(
    uint32_t in_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, "Common Arguments" for argument descriptions.

Estimated Size

One to four instructions.

Microengine Assembler Example Usage One

```
.sig sig1
.reg sub_val addr offset
move(addr, 0x100);
move(offset, 0);
move(sub_val, 0x2);
scratch_sub(sub_val, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.sig sig1
.reg sub_val addr offset
move(addr, 0x100);
move(offset, 0);
//
move(sub_val, 0x2);
scratch_sub(sub_val, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig);
addr = 10;
// clear left 4 bits and right 4 bits
ixp_scratch_sub(0x2, addr, &sig, sig_mask, 0);// subtract 2
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
addr = 10;
// clear left 4 bits and right 4 bits
ixp_scratch_sub(0x2, addr, &sig, SIG_NONE, 0);// subtract 2
__wait_for_all(&sig);
```

3.2.10.2.13 `scratch_test_and_decr()`

Reads from the scratch location specified by `in_scratch_addr` then decrements the value at that scratch location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Note: A decrement of the value zero results in zero.

Microengine Assembler Syntax

```
scratch_test_and_decr (out_data, in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_test_and_decr(
    __declspec(sram_read_reg) uint32_t *out_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine C Example Usage One

```
uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL sig_mask = __signals(&sig);
addr = 10;
ixp_scratch_test_and_decr(&out_data, addr, &sig, sig_mask, 0);
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
addr = 10;
ixp_scratch_test_and_decr(&out_data, addr, &sig, SIG_NONE, 0);
__wait_for_all(&sig);
```

3.2.10.2.14 `scratch_test_and_incr()`

Reads a scratch location and then increments the value at that scratch location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Note: If there is a carry out—as after incrementing `0xFFFFFFFF`—the resulting value at `in_scratch_addr` is zero.

Microengine Assembler Syntax

```
scratch_test_and_incr (out_data, in_scratch_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_scratch_test_and_incr(
    __declspec(sram_read_reg) uint32_t *out_data,
    volatile __declspec(scratch) void* in_scratch_addr,
    SIGNAL* REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.10.1, “Common Arguments” for argument descriptions.

Estimated Size

One to four instructions.

Microengine C Example Usage One

```
__declspec(sram_read_reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
SIGNAL_MASK sig_mask = __signals(&sig);
addr = 10;
ixp_scratch_test_and_incr(&out_data, addr, &sig, sig_mask, 0);
__implicit_read(&sig);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg) uint32_t out_data;
__declspec(scratch) uint32_t *addr;
SIGNAL sig;
addr = 10;
ixp_scratch_test_and_incr(&out_data, addr, &sig, SIG_NONE, 0);
__wait_for_all(&sig);
```

3.2.11 `sig`

The `sig` interface is used for signaling. Signals are used by one thread to indicate to another thread that an event has occurred—each thread can be executing on any microengine. Signals are also used by on-chip hardware units other than the microengines to indicate the completion of a requested operation. These hardware units use the same hardware-supported signaling mechanism as the microengines. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Table 3-12. `sig` API

Name	Description
`sig_block()`	Blocks microengine execution until a signal is received.
`sig_post()`	Sends an inter-thread signal to a thread-ID.
`sig_wait()`	Swaps out and waits on a signal.
`sig_ifctx_block()`	If the context of interest is executing, blocks microengine execution until a signal is received.
`sig_ifctx_post()`	If the context of interest is executing, sends an inter-thread signal to a thread-ID.
`sig_ifctx_wait()`	If the context of interest is executing, swaps out and waits on an inter-thread signal.

3.2.11.1 API Functions

3.2.11.1.1 `sig_block()`

Blocks this thread and therefore this microengine's further execution until the appropriate signal or signals are received. This operation blocks without a context swap, that is, goes into a loop, not allowing any other threads on the microengine to run, until the required signal or signals are received. When the required signals are received, the thread's execution continues.

Microengine Assembler Syntax

```
sig_block (SIG_OP, in_sig_list);
```

Microengine C Syntax

```
INLINE void ixp_sig_block(signal_op_t SIG_OP, SIGNAL_MASK in_sig_list);
```

Input

`SIG_OP`	Determines if the thread is to block until *any* of the indicated signals are received or until *all* of the indicated signals are received. • `ALL`—block until all signals of `in_sig_list` have been received • `ANY`—block until any signals of `in_sig_list` have been received See Section 2.6.13.6, "signal_op_t."
`in_sig_list`	The list of signals causing a thread to resume. `SIG_NONE` is an invalid argument in this operation. This list is either a single signal or a list of signals in the form: `signals(sig1, sig2, …, sign)` For microengine use `__signals(&sig1, &sig2)` returns the mask. See Section 2.6.13, "Signal Arguments and Usage."

Estimated Size

One to nine instructions.

Microengine Assembler Example Usage

```
.sig SIG0 SIG1 SIG2 SIG3 SIG4 SIG5
sig_block(ALL, signals(SIG0, SIG1, SIG2))
sig_block(ANY, signals(SIG3, SIG4))
sig_block(ALL, SIG5)
```

Microengine C Example Usage

```
sig_mask = 0;
sig_mask = __signals(&sig1, &sig2, &sig3, &sig4, &sig5, &sig6, &sig7);
ixp_sig_block (ALL, sig_mask);
__implicit_read(&sig1);
__implicit_read(&sig2);
__implicit_read(&sig3);
__implicit_read(&sig4);
__implicit_read(&sig5);
__implicit_read(&sig6);
__implicit_read(&sig7);
```

3.2.11.1.2 `sig_post()`

Sends an inter-thread signal to thread-ID. Thread-IDs are globally unique identifiers of microengine threads. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
sig_post (in_thread_id, in_sig_id);
```

Microengine C Syntax

```
INLINE void ixp_sig_post (uint32_t in_thread_id, SIGNAL* in_sig_id);
```

Input

`in_thread_id`	The thread ID—a constant—of the target thread.
`in_sig_id`	The signal to be sent to the target thread.

Estimated Size

Three or four instructions.

Microengine Assembler Example Usage

```
.sig SIG0
sig_post(2, SIG0)
```

Microengine C Example Usage

```
ixp_sig_post (8, &sig2); // post SIG2 to thread 8.
__implicit_read(&sig2);
```

3.2.11.1.3 `sig_wait()`

Swaps out this thread. Enables thread execution to resume when the signal is received. Other threads on this microengine can continue to execute.

Microengine Assembler Syntax

```
sig_wait (SIG_OP, in_sig_list);
```

Microengine C Syntax

```
INLINE void ixp_sig_wait(signal_op_t SIG_OP, SIGNAL_MASK in_sig_list);
```

Input

`SIG_OP`	Determines if the thread is to block until *any* of the indicated signals are received or until *all* of the indicated signals are received. • `ALL`—block until all signals of `in_sig_list` have been received • `ANY`—block until any signals of `in_sig_list` have been received See Section 2.6.13.6, “signal_op_t.”
`in_sig_list`	The list of signals causing the thread to wakeup. `SIG_NONE` is an invalid argument for this operation. Please refer to Section 2.6.13, “Signal Arguments and Usage.” For microengine use `__signals(&sig1, &sig2)` returns the mask.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.sig SIG0 SIG1 SIG2 SIG3 SIG4 SIG5
sig_wait(ANY, signals(SIG0, SIG1))
sig_wait(ANY, SIG2)
sig_wait(ALL, signals(SIG3, SIG4, SIG5))
```

Microengine C Example Usage

```
sig_mask = 0;
sig_mask = __signals(&sig1, &sig2, &sig3, &sig4, &sig5, &sig6, &sig7);
ixp_sig_wait (ANY, sig_mask);
__implicit_read(&sig1);
__implicit_read(&sig2);
__implicit_read(&sig3);
__implicit_read(&sig4);
__implicit_read(&sig5);
__implicit_read(&sig6);
__implicit_read(&sig7);
```

3.2.11.1.4 `sig_ifctx_block()`

If `in_ctx` matches this thread's context ID, blocks this thread and therefore this microengine's further execution until a signal or signals are received. Blocks without a context swap—that is, goes into a loop—not allowing any other threads on the microengine to run until the required signals are received. When the required signals are received, continues this thread's execution.

Microengine Assembler Syntax

```
sig_ifctx_block(in_ctx, SIG_OP, sig_expr_t in_sig_list);
```

Input

`in_ctx`	The context ID, a constant.
`SIG_OP`	Determines if the thread is to block until *any* of the indicated signals are received or until *all* of the indicated signals are received. • `ALL`—block until all signals of `in_sig_list` have been received • `ANY`—block until any signals of `in_sig_list` have been received See Section 2.6.13.6, "signal_op_t."
`in_sig_list`	The list of signals causing the thread to wakeup. `SIG_NONE` is an invalid argument for this operation. See Section 2.6.13, "Signal Arguments and Usage."

Estimated Size

One to ten instructions.

Microengine Assembler Example Usage

```
// ctx 2 spin until interthread signal is received
sig_ifctx_block (2, ANY, SIG_INTER_THREAD);
```

3.2.11.1.5 `sig_ifctx_post()`

If `in_ctx` matches this thread's context ID, sends an inter-thread signal to thread-ID. Thread-IDs are globally unique identifiers of microengine threads. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
sig_ifctx_post (in_ctx, in_thread_id, in_sig_id);
```

Input

`in_ctx`	The context ID, a constant.
`in_thread_id`	The thread ID of the target thread, a constant.
`in_sig_id`	The signal ID for the target thread.

Estimated Size

One to five instructions.

Microengine Assembler Example Usage

```
// if ctx 2, send an interthread signal to thread 22
sig_ifctx_post(2, 22, SIG_INTER_THREAD);
```

3.2.11.1.6 `sig_ifctx_wait()`

If `in_ctx` matches this thread's context ID, swaps out this thread. Enables thread execution to resume when the appropriate signal is received. Other threads on this microengine can continue to execute.

Microengine Assembler Syntax

```
sig_ifctx_wait (in_ctx, SIG_OP, in_sig_list);
```

Input

`in_ctx`	The context ID, a constant.
`SIG_OP`	Determines if the thread is to block until *any* of the indicated signals are received or until *all* of the indicated signals are received. • `ALL`—blocks until all signals of `in_sig_list` have been received • `ANY`—blocks until any signals of `in_sig_list` have been received See Section 2.6.13.6, "signal_op_t."
`in_sig_list`	The list of signals causing the thread to wakeup. `SIG_NONE` is not a valid value for this argument. See Section 2.6.13, "Signal Arguments and Usage."

Estimated Size

One or two instructions.

Microengine Assembler Example Usage

```
// thread requests signal, does not swap
sram_read($data[0], addr, 2, SIG_SRAM, SIG_NONE, ___);
       //
       // other code
       //
// if ctx 2, swap out until sram signal is received
sig_ifctx_wait (2, ANY, SIG_SRAM);
```

3.2.12 sram

The `sram` interface can be used for concurrent SRAM memory access, specifying hardware queue types, multiword reads and writes, and SRAM operations that perform more than just simple read or write access. Unless otherwise noted, output is to a read transfer register.

Table 3-13. `sram` API

Name	Description
`sram_add()`	Adds `in_data` to a 32-bit word at an SRAM location.
`sram_bits_clr()`	Clears `in_mask` bits of a 32-bit word at an SRAM location.
`sram_bits_set()`	Sets `in_mask` bits of a 32-bit word at an SRAM location.
`sram_bits_test_and_clr()`	Performs an atomic test and clear of an SRAM location. Reads the 32-bit word at the SRAM location then clears selected bits indicated by `in_mask`.
`sram_bits_test_and_set()`	Performs an atomic test and set of an SRAM location. Reads the 32-bit word at the SRAM location then sets selected bits indicated by `in_mask`.
`sram_decr()`	Decrements a 32-bit word at an SRAM location.
`sram_incr()`	Increments a 32-bit word at an SRAM location.
`sram_read()`	Reads one or more 32-bit longwords from SRAM.
`sram_sub()`	Subtracts `in_data` from a 32-bit word at an SRAM location.
`sram_swap()`	Reads then writes a 32-bit word at an SRAM location.
`sram_test_and_add()`	Reads then adds `in_data` to a 32-bit word at an SRAM location.
`sram_test_and_decr()`	Reads then decrements a 32-bit word at an SRAM location.
`sram_test_and_incr()`	Reads then increments a 32-bit word at an SRAM location.
`sram_write()`	Writes one or more 32-bit longwords to SRAM.

3.2.12.1 Common Arguments

This section describes the commonly used arguments in SRAM access operations.

Input

`in_data`	The value to be used in an `add`, `sub` or `swap` operation. This can be a constant, general-purpose register, or write transfer register.
`in_mask`	The 32-bit mask specifying the bits to set or clear.
`in_sram_addr`	Specifies the SRAM address that is the focus of the operation. If `ADDRESS_MODE` is `IXP2XXX` this is a byte address.
`in_addr_offset`	For *Microengine Assembler only*, this is the offset to be added to `in_sram_addr` to form the address to be accessed.
`REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
`Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, "queue_t."

Output

`out_data`	Contains the output from the operation—must be a read transfer register.

3.2.12.2 API Functions

3.2.12.2.1 `sram_bits_clr()`

Clears `in_mask` bits at location `in_sram_addr`. A one in any `in_mask` bit position clears the corresponding bit position at the location `in_sram_addr`. This operation is equivalent to the expression:

```
*in_sram_addr &= ~in_mask;
```

Microengine Assembler Syntax

```
sram_bits_clr (in_mask, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
void ixp_sram_bits_clr(
    uint32_t in_mask,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write memory access.

Microengine Assembler Example Usage One

```
.sig sig1
move(addr, 0x100);
// clear left 4 bits and right 4 bits
sram_bits_clr(0xf000000f, addr, 0, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.sig sig1
move(addr, 0x100);
// clear left 4 bits and right 4 bits
sram_bits_clr(0xf000000f, addr, 0, sig1, SIG_NONE, ___);
// other code
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram) volatile void *addr;
SIGNAL sig1;
SIGNAL_MASK sig_mask;
sig_mask = __signals(&sig1);
addr = 24;
// clear left 4 bits and right 4 bits
ixp_sram_bits_clr(0xf000000f, addr, &sig1, sig_mask, 0);
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram) volatile void *addr;
addr = 10;
SIGNAL sig1;
// clear left 4 bits and right 4 bits
ixp_sram_bits_clr(0xf000000f, addr, &sig1, SIG_NONE,0);
wait_for_all(&sig1);
```

3.2.12.2.2 `sram_bits_set()`

Sets `in_mask` bits at location `in_sram_addr`. A one in any `in_mask` bit position sets the corresponding bit position at the location `in_sram_addr`. This is equivalent to the expression: `*in_sram_addr |= in_mask;`

Microengine Assembler Syntax

```
sram_bits_set (in_mask, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
void ixp_sram_ bits_set(
    uint32_t in_mask,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write memory access.

Microengine Assembler Example Usage One

```
.sig sig1
move(addr, 0x100);
// set left 4 bits and right 4 bits
sram_bits_set(0xf000000f, addr, 0, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.sig sig1
move(addr, 0x100);
// set left 4 bits and right 4 bits
ixp_sram_bits_set(0xf000000f, addr, 0, sig1, SIG_NONE, ___);
//
// other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram) volatile void *addr;
SIGNAL sig1;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig1);
addr = 24;
// set left 4 bits and right 4 bits
ixp_sram_bits_set(0xf000000f, addr, &sig1, sig_mask,0);
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram) volatile void *addr;
SIGNAL sig1;
addr = 10;
// set left 4 bits and right 4 bits
ixp_sram_bits_set(0xf000000f, addr, &sig1, SIG_NONE,0);
wait_for_all(&sig1);
```

3.2.12.2.3 `sram_bits_test_and_clr()`

Reads an SRAM location and then clears `in_mask` bits at the same SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*. A one in any `in_mask` bit position clears the corresponding bit position at the location `in_sram_addr`.

Microengine Assembler Syntax

```
sram_bits_test_and_clr (out_data, in_mask, in_sram_addr, \
    in_addr_offset, REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_ bits_test_and_clr(
    __declspec(sram_read_reg)uint32_t*out_data,
    __declspec(sram_write_reg)uint32_t *in_mask,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write memory access.

Microengine Assembler Example Usage One

```
.sig sig1
move(addr, 0x100);
// get data, then clear left 4 bits and right 4 bits
sram_bits_test_and_clr($data, 0xf000000f, addr, 0, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.sig sig1
move(addr, 0x100);
// get data, then clear left 4 bits and right 4 bits
sram_bits_test_and_clr($data, 0xf000000f, addr, 0, sig1, SIG_NONE, ___);
//
// other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg) uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR sg_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sg_pr.odd, &sig_pr.even);
addr = 24;
// get data, then clear left 4 bits and right 4 bits
ixp_sram_bits_test_and_clr(&data, 0xf000000f, addr, &sg_pr, sig_mask,0);
__implicit_read(&sg_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg) uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR sg_pr;
addr = 24;
// get data, then clear left 4 bits and right 4 bits
ixp_sram_bits_test_and_clr(&data, 0xf000000f, addr, &sg_pr, SIG_NONE,0);
__wait_for_all(&sg_pr);
```

3.2.12.2.4 sram_bits_test_and_set()

Reads an SRAM location and then sets `in_mask` bits at the same SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*. A one in any `in_mask` bit position sets the corresponding bit position at the location `in_sram_addr`.

Microengine Assembler Syntax

```
sram_bits_test_and_set (out_data, in_mask, in_sram_addr, \
    in_addr_offset, REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE uint32_t ixp_sram_ bits_test_and_set(
    __declspec(sram_read_reg)uint32_t *out_data,
    __declspec(sram_write_reg)uint32_t *in_mask,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write memory access.

Microengine Assembler Example Usage One

```
.sig sig1
move(addr, 0x100);
// get data, then set left 4 bits and right 4 bits
sram_bits_test_and_set($data, 0xf000000f, addr, 0, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.sig sig1
move(addr, 0x100);
// get data, then set left 4 bits and right 4 bits
sram_bits_test_and_set($data, 0xf000000f, addr, 0, sig1, SIG_NONE, ___);
//
// other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg)uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR *sg_pr;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sg_pr.odd, &sig_pr.even);addr = 24;
// get data, then set left 4 bits and right 4 bits
ixp_sram_bits_test_and_set(&data, 0xf000000f, addr, &sg_pr, sig_mask,0);
__implicit_read(&sg_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg)uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR *sg_pr;
//
addr = 24;
// get data, then set left 4 bits and right 4 bits
ixp_sram_bits_test_and_set(&data, 0xf000000f, addr, &sg_pr, SIG_NONE,0);
__wait_for_all(&sg_pr);
```

3.2.12.2.5 `sram_read()`

Reads `in_lw_count` 32-bit longwords from the location `in_sram_addr`.

Microengine Assembler Syntax

```
sram_read (out_data, in_sram_addr, in_addr_offset, in_lw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_read(
    __declspec(sram_read_reg)void *out_data,
    volatile __declspec(sram) void* in_sram_addr,
    uint32_t in_lw_count,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK *in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

`in_lw_count`	The number of 32-bit longwords to read. The minimum longword count is one. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the maximum longword count is 16.

See Section 3.2.12.1, “Common Arguments” for other argument descriptions.

Estimated Size

One to five instructions—one read SRAM access.

Microengine Assembler Example Usage One

```
.xfer_order $data0, $data1, $data2
move(addr, 0x100);
// read 3 32-bit words
sram_read($data0, addr, 0, 3, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.xfer_order $data0, $data1, $data2
move(addr, 0x100);
sram_read($data0, addr, 0, 3, sig1, SIG_NONE, ___);
//
// some other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg)uint32_t data[3];
__declspec(sram) volatile void *addr;
SIGNAL sig1;
SIGNAL_MASK sig_mask;
addr = 24;
sig_mask = __signals(&sig1);
// read 3 32-bit words
ixp_sram_read(data, addr, 3, &sig1, sig_mask,0);
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg)uint32_t data[3];
SIGNAL sig1;
__declspec(sram) volatile void *addr;
addr = 10;
ixp_sram_read(data, addr, 3, &sig1, SIG_NONE,0);
__wait_for_all(&sig1);
```

3.2.12.2.6 `sram_write()`

Writes `in_lw_count` 32-bit longwords to the location `in_sram_addr`.

Microengine Assembler Syntax

```
sram_write (in_data, in_sram_addr, in_, in_addr_offset lw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_write(
    __declspec(sram_write_reg) void *in_data,
    volatile __declspec(sram) void* in_sram_addr,
    uint32_t in_lw_count,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

`in_lw_count`	The number of 32-bit longwords to write. The minimum longword count is one. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processors the maximum longword count is 16.

See Section 3.2.12.1, “Common Arguments” for other argument descriptions.

Estimated Size

One to five instructions—one SRAM write access.

Microengine Assembler Example Usage One

```
.xfer_order $data0 $data1
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
// write 2 32-bit words
sram_write($data0, addr, 0, 2, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.xfer_order $data0 $data1
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
// write 2 32-bit words
ixp_sram_write($data0, addr, 2, sig1, sig1, ___);
//
// some other code
//
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg)uint32_t data[2];
data[0] = 0x01234567;
data[1] = 0x89ABCDEF;
__declspec(sram) volatile void *addr;
SIGNAL sig1;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig1);
addr = 24;
// write 2 32-bit words
ixp_sram_write(data, addr, 2, &sig1, sig_mask,0);
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram_write_reg)uint32_t data[2];
data[0] = 0x01234567;
data[1] = 0x89ABCDEF;
__declspec(sram) volatile void *addr;
SIGNAL sig1;
addr = 24;
// write 2 32-bit words
ixp_sram_write(data, addr, 2, &sig1, SIG_NONE,0);
__wait_for_all(&sig1);
```

3.2.12.2.7 `sram_add()`

Add `in_data` to the value at the location `in_sram_addr`. This is equivalent to the expression:
`*in_sram_addr += in_data;`

Note: There is no carry or overflow indication—the value wraps through zero.

Microengine Assembler Syntax

```
sram_add(in_data, in_sram_addr, in_addr_offset, REQ_SIG, \
   in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_ add(
   uint32_t in_data,
   volatile __declspec(sram) void* in_sram_addr,
   SIGNAL *REQ_SIG,
   SIGNAL_MASK in_wakeup_sigs,
   queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.xfer_order $data0 $data1
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
// write 2 32-bit words
sram_write($data0, addr, 0, 2, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.xfer_order $data0 $data1
.sig sig1
move($data0, 0x01234567);
move($data1, 0x89ABCDEF);
move(addr, 0x100);
ixp_sram_write($data0, addr, 2, sig1, sig1, ___);// write 2 32-bit words
       //
       // some other code
       //
ctx_arb[sig1]
```

Microengine C Example Usage One

```
__declspec(sram) volatile void *addr;
addr = 24;
SIGNAL sig1;
SIGNAL_MASK sig_mask;
Sig_mask = __signals(&sig1);
// clear left 4 bits and right 4 bits
ixp_sram_add(0x2, addr, &sig1, sig_mask,0);// add 2
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram) volatile void *addr;
addr = 24;
SIGNAL sig1;
// clear left 4 bits and right 4 bits
ixp_sram_add(0x2, addr, &sig1, SIG_NONE,0);// add 2
__wait_for_all(&sig1);
```

3.2.12.2.8 `sram_decr()`

Decrements the value at the location `in_sram_addr`.

Note: The decrement of an SRAM location whose current value is zero results in a value of zero.

Microengine Assembler Syntax

```
sram_decr (in_sram_addr, in_addr_offset);
```

Microengine C Syntax

```
INLINE void ixp_sram_decr(volatile __declspec(sram) void* in_sram_addr);
```

Inputs and Outputs

See Section 3.2.12.1, "Common Arguments" for argument descriptions.

Estimated Size

One to five instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage

```
sram_decr(addr, offset);
```

Microengine C Example Usage

```
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_decr(addr);
```

3.2.12.2.9 `sram_incr()`

Increments the value at the location `in_sram_addr`.

Note: If there is a carry out—after incrementing the value `0xFFFFFFFF`—the resulting value at `in_sram_addr` is zero.

Microengine Assembler Syntax

```
sram_incr(in_sram_addr, in_addr_offset);
```

Microengine C Syntax

```
INLINE void ixp_sram_decr(volatile __declspec(sram) void* in_sram_addr);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage

```
.reg addr offset
move(addr, 0x100)
move(offset, 0)
sram_incr(addr, offset);
```

Microengine C Example Usage

```
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_incr(addr);
```

3.2.12.2.10 `sram_test_and_add()`

Reads an SRAM location and then adds `in_data` to the value at that SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Note: There is no carry or overflow indication—the value wraps through zero.

Microengine Assembler Syntax

```
sram_test_and_add (out_data, in_data, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_test_and_add(
    __declspec(sram_read_reg)uint32_t *out_data,
    __declspec(sram_write_reg)uint32_t *in_data,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.reg $out_data in_data
.sig sig1
sram_test_and_add($out_data, in_data, addr, offset, sig1, sig1, ___)
```

Microengine Assembler Example Usage Two

```
.reg $out_data in_data
.sig sig1
sram_test_and_add($out_data, in_data, addr, offset, sig1, ___, ___)
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg)uint32_t in_data = 2;
__declspec(sram_read_reg)uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask = __signals(&sig_pr.odd, &sig_pr.even);
addr = 24;
ixp_sram_test_and_add(&out_data, &in_data, addr, &sig_pr, sig_mask,0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_write_reg)uint32_t in_data = 2;
__declspec(sram_read_reg)uint32_t out_data;
__declspec(sram) volatile void *addr;
SIGNAL_PAIR sig_pr;
addr = 24;
ixp_sram_test_and_add(&out_data, &in_data, addr, &sig_pr, SIG_NONE,0);
__wait_for_all(&sig_pr);
```

3.2.12.2.11 `sram_sub()`

Subtracts `in_data` from the value at the location `in_sram_addr`. This operation is equivalent to the expression:

```
*in_sram_addr -= in_data;
```

Note: If `in_data` is greater than the value at `in_sram_addr`, the result is zero.

Microengine Assembler Syntax

```
sram_sub (in_data, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_ sub(
    uint32_t in_data,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

Two to six instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.reg $sub_val
.sig sig1
sram_sub($sub_val, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $sub_val
.sig sig1
sram_sub($sub_val, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram) volatile void *addr;
SIGNAL sig1;
SIGNAL_MASK sig_mask = __signals(&sig1);
addr = 24;
// clear left 4 bits and right 4 bits
ixp_sram_sub(0x2, addr, &sig1, sig_mask, 0);// subtract 2
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram) volatile void *addr;
SIGNAL sig1;
addr = 24;
// clear left 4 bits and right 4 bits
ixp_sram_sub(0x2, addr, &sig1, SIG_NONE, 0);     // subtract 2
__wait_for_all(&sig1);
```

3.2.12.2.12 `sram_swap()`

Reads an SRAM location and then writes `in_data` to that SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Microengine Assembler Syntax

```
sram_swap (out_data, in_data, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_swap(
__declspec(sram_read_reg)uint32_t *out_data,
__declspec(sram_write_reg)uint32_t *in_data,
volatile __declspec(sram) void* in_sram_addr,
SIGNAL_PAIR *REQ_SIG, SIGNAL_MASK in_wakeup_sigs, queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to eight instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.reg $out_data, in_data
.sig sig1
sram_swap($out_data, in_data, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $out_data, in_data
.sig sig1
sram_swap($out_data, in_data, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_write_reg)uint32_t in_data = 2;
__declspec(sram_read_reg)uint32_t out_data;
SIGNAL_PAIR sig_pr;
SIGNAL_MASK sig_mask = __signals(&sig_pr.odd, &sig_pr.even);
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_swap(&out_data, &in_data, addr, &sig_pr, sig_mask,0);
__implicit_read(&sig_pr);
```

Microengine C Example Usage Two

```
__declspec(sram_write_reg)uint32_t in_data = 2;
__declspec(sram_read_reg)uint32_t out_data;
SIGNAL_PAIR sig_pr;
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_swap(&out_data, &in_data, addr, &sig_pr, SIG_NONE,0);
__wait_for_all(&sig_pr);
```

3.2.12.2.13 `sram_test_and_decr()`

Reads an SRAM location and then decrements the value at that SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the *read* to the time of the *write*.

Note: A decrement of an SRAM location whose value is zero has no effect on that value—that is, the result is still zero.

Microengine Assembler Syntax

```
sram_test_and_decr (out_data, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_test_and_decr(
    __declspec(sram_read_reg)uint32_t *out_data,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.reg $out_data
.sig sig1
sram_test_and_decr($out_data, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $out_data
.sig sig1
sram_test_and_decr($out_data, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg)uint32_t *out_data;
SIGNAL sig1;
SIGNAL_MASK sig_mask = __signals(&sig1);
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_test_and_decr(&out_data, addr, &sig1, sig_mask, 0);
__implict_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg)uint32_t *out_data;
SIGNAL sig1;
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_test_and_decr(&out_data, addr, &sig1, SIG_NONE, 0);
__wait_for_all(&sig1);
```

3.2.12.2.14 `sram_test_and_incr()`

Reads an SRAM location and then increments the value at that SRAM location. This is performed as a single atomic memory operation with no possibility of any other access to that location from the time of the read to the time of the write.

Note: If there is a carry out—as after incrementing `0xFFFFFFFF`—the resulting value at `in_sram_addr` is zero.

Microengine Assembler Syntax

```
sram_test_and_incr (out_data, in_sram_addr, in_addr_offset, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_sram_test_and_incr(
    __declspec(sram_read_reg)uint32_t *out_data,
    volatile __declspec(sram) void* in_sram_addr,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Inputs and Outputs

See Section 3.2.12.1, “Common Arguments” for argument descriptions.

Estimated Size

One to five instructions—one SRAM read-modify-write access.

Microengine Assembler Example Usage One

```
.reg $out_data
.sig sig1
sram_test_and_incr($out_data, addr, offset, sig1, sig1, ___);
```

Microengine Assembler Example Usage Two

```
.reg $out_data
.sig sig1
sram_test_and_incr($out_data, addr, offset, sig1, ___, ___);
ctx_swap[sig1]
```

Microengine C Example Usage One

```
__declspec(sram_read_reg)uint32_t* out_data;
SIGNAL sig1;
SIGNAL_MASK sig_mask = __signals(&sig1);
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_test_and_incr(&out_data, addr, &sig1, sig_mask, 0);
__implicit_read(&sig1);
```

Microengine C Example Usage Two

```
__declspec(sram_read_reg)uint32_t* out_data;
SIGNAL sig1;
__declspec(sram) volatile void *addr;
addr = 24;
ixp_sram_test_and_incr(&out_data, addr, &sig1, SIG_NONE, 0);
__wait_for_all(&sig1);
```

3.2.13 `tbuf`

The `tbuf` interface can be used for TBUF writes. The TBUF is the interface buffer for data to be transmitted to the network. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Note: In documentation for the Intel® IXP1200 Network Processor this is called the transmit TFIFO.

Table 3-14. `tbuf` API

Name	Description
`tbuf_write()`	Writes `in_qw_count` quadwords to TBUF.
`tbuf_addr_from_elem()`	Converts an element index to a TBUF address.
`tbuf_elem_from_addr()`	Converts a TBUF address to an element index.

3.2.13.1 API Functions

3.2.13.1.1 `tbuf_write()`

Writes `in_qw_count` quadwords to the transmit port buffer (TBUF).

Microengine Assembler Syntax

```
tbuf_write (in_data,in_tbuf_addr, in_addr_offset, in_qw_count, \
    REQ_SIG, in_wakeup_sigs, Q_OPTION);
```

Microengine C Syntax

```
INLINE void ixp_tbuf_write(
    void *in_data,
    volatile void __declspec( tbuf) *in_tbuf_addr,
    uint32_t in_qw_count,
    SIGNAL *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    queue_t Q_OPTION);
```

Input

`in_data`	The data to write. The caller declares `in_data` with the understanding that it is assigned to write transfer registers.
`in_tbuf_addr`	The TBUF address. For an `ADDRESS_MODE` of `IXP2XXX` this is a byte address.
`in_addr_offset`	For *Microengine Assembler only*, this offset is added to `in_tbuf_addr` to form the address to be accessed.
`in_qw_count`	The register or 64-bit constant—that is, a quadword—containing the count of quadwords to write. The minimum quadword count is one. The maximum quadword count is sixteen.
`REQ_SIG`	The requested signal. See Section 2.6.13, “Signal Arguments and Usage.”
`in_wakeup_sigs`	The list of signals causing a thread to swap or wakeup. See common section Section 2.6.13, “Signal Arguments and Usage.”
`Q_OPTION`	The directive for memory controller queue selection. See Section 2.6.10, “queue_t.”

Estimated Size

One to seven instructions.

Microengine Assembler Example Usage

```
#define_eval TEST_TBUF_ADDR 0x2000

.reg cnt $x1 $x2 $x3 $x4 $x5 $x6 $x7 $x8
.xfer_order $x1 $x2 $x3 $x4 $x5 $x6 $x7 $x8
.sig sig1

move($x1, 0x12345671);
move($x2, 0x12345672);
move($x3, 0x12345673);
move($x4, 0x12345674);
move($x5, 0x12345675);
move($x6, 0x12345676);
move($x7, 0x12345677);
move($x8, 0x12345678);
move(cnt, 4);
tbuf_write($x1, TEST_TBUF_ADDR, 0, cnt, sig1, sig1, ___);
```

Microengine C Example Usage

```
__declspec(sram_write_reg) uint32_t sram_wr_xfer[4];
SIGNAL sig_tbuf;
ixp_tbuf_write(
    sram_wr_xfer,
    addr,
    2,
    &sig_tbuf,
    __signals(&sig_tbuf),
    ___);
```

3.2.13.1.2 `tbuf_addr_from_elem()`

Converts a TBUF element index to a TBUF address. This is equivalent to the expression:

```
out_tbuf_addr = out_index  * ELEMENT_SIZE
```

Microengine Assembler Syntax

```
tbuf_addr_from_elem (out_tbuf_addr, in_tbuf_elem, ELEMENT_SIZE);
```

Microengine C Syntax

```
INLINE __declspec(tbuf) void* ixp_tbuf_addr_from_elem (
    uint32_t in_tbuf_elem, netif_elem_size_t ELEMENT_SIZE);
```

Input

`in_tbuf_elem`	The TBUF element index.
`ELEMENT_SIZE`	The constant size of TBUF elements in bytes—either 64, 128, or 256 bytes. The value is one of {`NETIF_ELEM_64`, `NETIF_ELEM_128`, `NETIF_ELEM_256`}.

Output

`out_tbuf_addr` *or* Return Value	The address of the TBUF element. For an `ADDRESS_MODE` of `IXP2XXX` this is a byte address.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg out_addr
tbuf_addr_from_elem(out_addr, 8, NETIF_ELEM_64);
```

Microengine C Example Usage

```
__declspec(tbuf) void* tbuf_addr;
uint32_t tbuf_element = 4;
tbuf_addr = ixp_tbuf_addr_from_elem(tbuf_element, NETIF_ELEM_64);
```

3.2.13.1.3 `tbuf_elem_from_addr()`

Returns a TBUF element index given the element address using the following equation:

```
out_index = in_tbuf_addr / ELEMENT_SIZE
```

Microengine Assembler Syntax

```
tbuf_elem_from_addr (out_tbuf_elem, in_tbuf_addr, ELEMENT_SIZE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ tbuf_elem_from_addr (
    void __declspec(tbuf) *in_tbuf_addr,
    netif_elem_size_t ELEMENT_SIZE);
```

Input

`in_tbuf_addr`	The address of the TBUF element. For an `ADDRESS_MODE` of `IXP2XXX` this is a byte address.
`ELEMENT_SIZE`	The constant size of TBUF elements in bytes—either 64, 128, or 256 bytes. The value is one of {`NETIF_ELEM_64`, `NETIF_ELEM_128`, `NETIF_ELEM_256`}.

Output/Returns

`out_tbuf_elem` ***or*** Return Value	The TBUF element index.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg out_elem, in_addr
move(in_addr, 0x100);
tbuf_elem_from_addr(out_elem, in_addr, NETIF_ELEM_64);
```

Microengine C Example Usage

```
__declspec(tbuf) void* tbuf_addr=0x100;
uint32_t tbuf_element;
tbuf_element = ixp_tbuf_elem_from_addr(tbuf_addr, NETIF_ELEM_64);
```

3.2.14 `cam` Sharing

In applications where more than one microblock is running independently on one microengine, Conten Address Memory (CAM) sharing is required.

The CAM sharing API hides the underlying complexity of CAM sharing from the microblock developer.

The CAM sharing API supports the following:

- CAM sharing implementation across different microblocks running on one microengine.
- Original Least Recently Used (LRU) CAM implementation supported by hardware (used in absence of CAM sharing).

Table 3-15 summarizes the CAM sharing API. The following assumptions and dependencies apply when using the CAM sharing API:

- Microblocks do not directly access CAM. Microblocks use the CAM API for any reference to CAM, regardless of the number of microblocks running on one microengine. If a single microblock is running on a microengine, it uses the CAM API to access the LRU implementation supported by the hardware.This way, the CAM can be shared with other microblocks without code modifications.
- The CAM API requires *dl_microblock_cam_handle* to be defined for each microblock. This is the handle assigned to each microblock by the application-specific dispatch loop.

  Note: The handle structure should not be modified. *dl_microblock_cam_handle* = unique BID (microblock ID), total CAM entries allocated to this microblock , absolute first CAM entry number for this microblock (i.e. CAM entry number from [0 thru 15])
- The following variables are required for the CAM sharing implementation only. These variables are not used in the original LRU CAM implementation:
  - *ixp_cam_bit_vector*: The bit vector is maintained to find the free CAM entry within the microblock.
  - *ixp_cam_ref_cnt1* and *ixp_cam_ref_cnt2*: *ixp_cam_ref_cnt1* holds the reference count for CAM entries 0 through 7. *ixp_cam_ref_cnt2* holds the reference count for CAM entries 8 through 15. Each CAM entry has 4 bits assigned for reference count.

    Note: By using registers instead of local memory to keep reference count, CAM usage (shared or not shared) becomes transparent to the user. Otherwise for reference count in the Local Memory case, user will be required to pass LM handle and LM base. Number of instructions are almost comparable in both cases for microcode.
- In CAM sharing implementation, the input parameter `in_entry_num` to the API can not exceed the total number of CAM entries assigned to a particular microblock.
- In CAM sharing implementation, the API returns the relative CAM entry number within the microblock.
- In the original LRU implementation (absence of CAM sharing), all the CAM operation conditions specified in *Intel® IXP2xxx Programmer's Reference Manual* apply.

Table 3-15. CAM Sharing API

Name	Description
`cam_init()`	Initializes CAM variables and clears all CAM entries
`cam_entry_read_state()`	Reads the 4-bit state value for the specified CAM entry
`cam_entry_write_state()`	Writes the 4-bit state value for the specific CAM entry
`cam_entry_read_tag()`	Reads the 32-bit tag value for the specified CAM entry
`cam_clearall()`	Clears the CAM entries
`cam_entry_write()`	Writes the 32-bit tag value and the 4-bit state value for the specified CAM entry.
`cam_exit_using_entry()`	Thread notifies that it is done using the CAM entry
`cam_entry_lookup()`	Do a CAM lookup for the specified CAM entry
`cam_entry_lookup_with_lm()` (provided for microcode macro only)	Do a CAM lookup for the specified CAM entry. And also set the Local Memory pointer to point to the data structure in Local Memory according to the look up result.

3.2.14.1 API Functions

3.2.14.1.1 `cam_init()`

Initializes the internal variables used by the CAM API. It also clears all CAM entries by writing 0x0 to the tag, clearing all the state bits, and putting the LRU into an initial state where CAM entry 0 is LRU, ..., CAM entry 15 is Most Recently Used (MRU)).

The CAM is not reset by microengine reset. Software must call `ixp_cam_init()` before using any CAM API. It is assumed that after calling this function, all CAM entries are initialized to unique values using `cam_write()` prior to doing CAM lookup.

Microengine C Syntax

```
void ixp_cam_init()
```

Microengine C Example Usage

```
ixp_cam_init();
```

Microcode Macro Syntax

```
cam_init()
```

Microcode Macro Example Usage

```
cam_init()
```

Input

```
None
```

Output/Returns

None

3.2.14.1.2 `cam_entry_read_state()`

Reads the 4-bit state value for the specified CAM entry. The value is placed in bits 11:8 of the output variable with all other bits 0.

Microengine C Syntax

```
ixp_cam_lookup_t cam_entry_read_state(
    uint32_t in_entry_num,
    uint32_t in_bid,
    uint32_t in_num_entries,
    uint32_t in_first_entry)
```

Microengine C Example Usage

```
camState = ixp_cam_entry_read_state(camEntry, AAL2_RX_CAM_HANDLE);
    where (#define AAL2_RX_CAM_HANDLE 1, 8, 0 )
```

Microcode Macro Syntax

```
cam_entry_read_state(
    out_state_val,
    in_entry_num,
    in_bid,
    in_num_entries,
    in_first_entry)
```

Microcode Macro Example Usage

```
cam_entry_read_state(cam_state, cam_entry, AAL2_RX_CAM_HANDLE)
        where (#define AAL2_RX_CAM_HANDLE 1, 8, 0 )
```

Input

`in_entry_num`	CAM entry number (relative CAM entry number within the microblock, if CAM is shared)
`in_dl_microblock_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- no of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0-15]

Output/Returns

`out_state_value`	4-bit state value of the specified CAM entry. The value is placed in bits 11:8 of the `out_state_val` with all other bits 0.

3.2.14.1.3 `cam_entry_write_state()`

Writes the 4-bit constant value into the state bits of the specified CAM entry. The tag value is not changed. This function does not modify the LRU list.

Microengine C Syntax

```
void ixp_cam_entry_write_state(
    uint32_t in_entry_num,
    uint32_t in_state_val,
    uint32_t  in_bid,
    uint32_t in_num_entries,
    uint32_t in_first_entry)
```

Microengine C Example Usage

```
ixp_cam_entry_write_state(camEntry, CAM_STATE_VALID, AAL2_RX_CAM_HANDLE);
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
cam_entry_write_state(
    in_entry_num,
    in_state_val,
    in_bid,
    in_num_entries,
    in_first_entry)
```

Microcode Macro Example Usage

```
cam_entry_write_state(cam_entry, CAM_STATE_VALID, AAL2_RX_CAM_HANDLE)
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_entry_num`	CAM entry number (relative CAM entry number within the microblock, if CAM is shared)
`in_state_val`	4-bit constant specifying the state value to write to the CAM entry
`in_dl_microblock_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 -15]

Output/Returns

```
None
```

3.2.14.1.4 `cam_entry_read_tag()`

Reads the 32-bit tag value for the specified CAM entry. Out of this 32-bit tag value, the API returns SRC_KEY_BIT_SZ bits as the tag.

Microengine C Syntax

```
uint32_t ixp_cam_entry_read_tag(
    uint32_t in_entry_num,
    uint32_t in_bid,
    uint32_t in_num_entries,
    uint32_t  in_first_entry)
```

Microengine C Example Usage

```
camTag = ixp_cam_entry_read_tag(camEntry, AAL2_RX_CAM_HANDLE);
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
cam_entry_read_tag(
    out_tag,
    in_entry_num,
    in_bid,
    in_num_entries,
    in_first_entry)
```

Microcode Macro Example Usage

```
cam_entry_read_tag(cam_tag, cam_entry, AAL2_RX_CAM_HANDLE)
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_entry_num`	CAM entry number (relative CAM entry number within the microblock, if CAM is shared)
`in_dl_microblock_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* microblock block id • *in_num_entries* number of CAM entries for this microblock • *in_first_entry* absolute first CAM entry number for this microblock [0 - 15]

Output/Returns

`out_cam_tag`	tag value of the specified CAM entry (Out of this 32-bit tag value, API returns SRC_KEY_BIT_SZ bits as the tag)

3.2.14.1.5 `cam_clearall()`

Clears the CAM entries.

When the CAM is shared, it clears the CAM entries for the specific microblock. The CAM tags and state bits of entries are not cleared.

When CAM is not shared and original LRU implementation is used, it clears all the CAM entries. The CAM tags and state bits are also cleared.

Microengine C Syntax

```
ixp_cam_clearall(
    uint32_t in_bid,
    uint32_t in_num_entries,
    uint32_t in_first_entry)
```

Microengine C Example Usage

```
ixp_cam_clearall(AAL2_RX_CAM_HANDLE);
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
cam_clearall(
    in_bid,
    in_num_entries,
    in_first_entry)
```

Microcode Macro Example Usage

```
cam_clearall(AAL2_RX_CAM_HANDLE)
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_dl_microblockname_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 - 15]

Output/Returns

```
None
```

3.2.14.1.6 `cam_entry_write()`

Writes a 32-bit tag value and 4-bit constant state value for the specified CAM entry.

BID of size BID_BIT_SZ is concatenated with `in_key` of size SRC_KEY_BIT_SZ to form the 32-bit CAM tag. In case of LRU implementation this entry will be marked as MRU (Most Recently Used).

No two tags can have the same value. If this rule is violated, then the result of CAM lookup and LRU state are unpredictable.

Microengine C Syntax

```
    ixp_cam_entry_write(
        uint32_t in_entry_num,
        uint32_t in_key,
        uint32_t in_state_val,
        uint32_t in_bid,
        uint32_t in_num_entries,
        uint32_t in_first_entry)
```

Microengine C Example Usage

```
    ixp_cam_entry_write(camEntry, srcKey, CAM_STATE_VALID,
AAL2_RX_CAM_HANDLE);
        where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
    cam_entry_write(
        in_entry_num,
        in_key,
        in_state_val,
        in_bid,
        in_num_entries,
        in_first_entry)
```

Microcode Macro Example Usage

```
cam_entry_write(cam_entry, src_key, CAM_STATE_VALID, AAL2_RX_CAM_HANDLE)
        where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_entry_num`	CAM entry number (relative CAM entry number within the microblock, if CAM is shared)
`in_state_val`	4-bit constant specifying state value to write to the CAM entry
`in_key`	32-bit key to write to the tag of CAM entry. (Out of this 32-bit key, SRC_KEY_BIT_SZ bits of `in_key` are concatenated with BID_BIT_SZ bits of BID to form the 32-bit tag.)
`in_dl_microblockname_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 - 15].

Output/Returns

None

3.2.14.1.7 cam_exit_using_entry()

Notifies that the thread is exiting the usage of the CAM entry. It should be called when a thread is done using the CAM entry.

Microengine C Syntax

```
void ixp_cam_exit_using_entry(
    uint32_t in_entry_num,
    uint32_t in_bid,
    uint32_t in_num_entries,
    uint32_t in_first_entry)
```

Microengine C Example Usage

```
ixp_cam_exit_using_entry(camEntry, AAL2_RX_CAM_HANDLE);
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
cam_exit_using_entry(
    in_entry_num,
    in_bid,
    in_num_entries,
    in_first_entry)
```

Microcode Macro Example Usage

```
cam_exit_using_entry(cam_entry, AAL2_RX_CAM_HANDLE)
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_entry_num`	CAM entry number (relative CAM entry number within the microblock, if CAM is shared)
`in_dl_microblockname_CAM_handle`	microblock specific CAM handle assigned by the dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 - 15].

Output/Returns

None

3.2.14.1.8 `cam_entry_lookup()`

Performs a CAM lookup and returns the lookup result. In case of CAM sharing implementation, the lookup result has the relative CAM entry number within the microblock.

Microengine C Syntax

```
    ixp_cam_lookup_t cam_entry_lookup(
        uint32_t in_key,
        uint32_t in_bid,
        uint32_t in_num_entries,
        uint32_t in_first_entry)
```

Microengine C Example Usage

```
    lookupResult = ixp_cam_entry_lookup(srcKey, AAL2_RX_CAM_HANDLE);
        where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Microcode Macro Syntax

```
    cam_entry_lookup(
        out_lookup_result,
        in_key, in_bid,
        in_num_entries,
        in_first_entry)
```

Microcode Macro Example Usage:

```
cam_entry_lookup(lookup_result, src_key, AAL2_RX_CAM_HANDLE)
        where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_key`	32-bit key for the CAM entry lookup. (Out of this 32-bit key, SRC_KEY_BIT_SZ bits of `in_key` are concatenated with BID_BIT_SZ bits of BID to form the 32-bit lookup key.)
`in_dl_microblockname_CAM_handle`	microblock specific CAM handle assigned by dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 - 15].

Output/Returns

`out_lookup_result`	bits 3: 6 specify CAM entry number, bit 7 specifies the CAM hit/miss and bits 8:11 specify the state bits. All other bits are 0. (Function returns relative CAM entry number within the microblock if CAM is shared.)

3.2.14.1.9 `cam_entry_lookup_with_lm()`

Similar to `cam_entry_lookup()` except that this function also sets the local memory pointer to point to the data structure in local memory corresponding to the CAM entry.

Note: This function is not supported in Microengine C.

Microengine C Syntax

Not supported.

Microcode Macro Syntax

```
    cam_entry_lookup_with_lm(
        out_lookup_result,
        in_key,
        in_lm_handle,
        in_lm_start_addr,
        in_bid,
        in_num_entries,
        in_first_entry)
```

Microcode Macro Example Usage

```
cam_entry_lookup_with_lm(lookup_result, src_key, RXC_LM_HANDLE,
RXC_INFO_LM_BASE, AAL2_RX_CAM_HANDLE)
    where (#define AAL2_RX_CAM_HANDLE1, 8, 0 )
```

Input

`in_key`	32-bit key for the CAM entry lookup. (Out of this 32-bit key, SRC_KEY_BIT_SZ bits of `in_key` are concatenated with BID_BIT_SZ bits of BID to form the 32-bit lookup key.)
`in_lm_handle`	local memory handle number (0 or 1)
`in_lm_start_addr`	2-bit local memory address where the data structure is located. (Value 0, 1, 2, 3 represents 0, 1024, 2048, 3072 bytes in local memory, respectively.)
`in_dl_microblockname_CAM_handle`	microblock specific CAM handle assigned by the dispatch loop. It consists of three parameters: • *in_bid* -- microblock block id • *in_num_entries* -- number of CAM entries for this microblock • *in_first_entry* -- absolute first CAM entry number for this microblock [0 - 15].

Output/Returns

`out_lookup_result`	bits 3: 6 specify CAM entry number, bit 7 specifies the CAM hit/miss and bits 8:11 specify the state bits. All other bits are 0. (This function returns relative CAM entry number within the microblock if CAM is shared)

3.2.14.2 Using the Content Address Memory Sharing API

This section provides a brief overview of how to use the CAM sharing API:

- Define `dl_microblock_cam_handle` for each microblock in case of both CAM sharing and original LRU CAM implementation. This is defined by the application-specific dispatch loop header file (for example, in *dl_system.h*)
  - — Example `#define AAL2_RX_CAM_HANDLE 1, 8, 0` (where 1 = unique BID , 8 = total CAM entries for this microblock, 0 = absolute first CAM entry for this microblock)
- Define the following for CAM sharing implementation only:
  - — Define BID and SRC_KEY distribution for the formation of 32-bit CAM key for lookup.This is defined by the application-specific dispatch loop header file (for example, in *dl_system.h*). The CAM key structure can be changed according to the application needs but it must be less than 32-bits.

```
#define SRC_KEY_BIT_SZ 28
#define BID_BIT_SZ 4
```

  - — Define CAM_SHARED in header file of each of the microblocks.

Utilities for the Microengine 4

The Utility Library provides a range of data structures and algorithm support including generic table lookups, endian swaps, and other useful functions.

4.1 bytefield

The `bytefield` interface performs field extraction, modification, and insertion respecting the endian format of the data. The field value is swapped to form a numeric field in network order—that is, in big-endian order. Finally the function performs the specified operation—for example a decrement operation—on the field.

Table 4-1. bytefield API

Name	Description
`bytefield_dbl_extract()`	Extracts the byte field from two 32-bit values.
`bytefield_decr()`	Decrements the byte field.
`bytefield_extract()`	Extracts the field from a 32-bit value.
`bytefield_incr()`	Increments a byte field.
`bytefield_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` into `io_a`.
`bytefield_select()`	Returns bytes of `in_src` specified by `in_byte_mask`.
`bytefield_br_eq()`	Compares a byte field to a constant and branches if they are equal.
`bytefield_br_gtr()`	Compares a byte field to a constant and branches if the field is greater than the constant.
`bytefield_br_gtreq()`	Compares a byte field to a constant and branches if the field is greater than or equal to the constant.
`bytefield_br_less()`	Compares a byte field to a constant and branches if the field is less than the constant.
`bytefield_br_lesseq()`	Compares a byte field to a constant and branches if the field is less than or equal to the constant.
`bytefield_comp()`	Compares a byte field to a constant and sets the condition code.
`bytefield_shf_left_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` in `out_result` after shifting `in_b` left by `in_shift_amt`.
`bytefield_shf_right_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` in `out_result` after shifting `in_b` right by `in_shift_amt`.

Table 4-1. `bytefield` API (Continued)

Name	Description
`bytefield_clr_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` into `out_result`, clearing `out_result` prior to performing the insertion.
`bytefield_clr_shf_left_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` into `out_result` after shifting `in_b` left by `in_shift_amt`, clearing `out_result` prior to performing the insertion.
`bytefield_clr_shf_right_insert()`	Inserts bytes of `in_b` specified by `in_byte_mask` into `out_result` after shifting `in_b` right by `in_shift_amt`, clearing `out_result` prior to performing the insertion.

4.1.1 API Functions

4.1.1.1 `bytefield_dbl_extract()`

Extracts a contiguous byte field that crosses the boundary of two 32-bit values. The maximum field size is 4 bytes.

Microengine Assembler Syntax

```
bytefield_dbl_extract(out_result, in_a, IN_A_START_BYTE, \
    in_b, IN_B_END_BYTE, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_dbl_extract(
    uint32_t in_a,
    IMMED IN_A_START_BYTE,
    uint32_t in_b,
    IMMED IN_B_END_BYTE,
    IMMED IN_LOAD_CC);
```

Input

`in_a`	The lefthand `uint32_t` source.
`IN_A_START_BYTE`	The constant index of the beginning byte position within `in_a`—the value can be from zero to three, inclusive. Valid combinations of `IN_A_START_BYTE` and `IN_B_END_BYTE` are those that result in at most 4 bytes between them—that is, the maximum field size that can be extracted is 4 bytes.
`in_b`	The righthand `uint32_t` source. For *Microengine Assembler only*, if this argument is located in a GPR, it must be on the opposite bank from `out_result`.
`IN_B_END_BYTE`	The constant index of the last byte position within `in_b`—the value can be from zero to three, inclusive. Since a maximum of 4 bytes can be extracted from `in_a` and `in_b`, `IN_B_END_BYTE` must be less than `IN_B_START_BYTE`. Valid combinations of `IN_A_START_BYTE` and `IN_B_END_BYTE` are those that result in at most 4 bytes between them—that is, the maximum field size that can be extracted is 4 bytes.
`IN_LOAD_CC`	A constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Output/Returns

`out_result` *or* Return Value	The requested byte field, 2 to 4 bytes in extent.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
#define BIG_ENDIAN
immed32(a, 0x01234567);
immed32(b, 0x89ABCDEF);
bytefield_dbl_extract(result, a, 2, b, 1, NOT_LOAD_CC); // extract quadword
              // result is 0x456789AB
```

Microengine C Example Usage One

```
#define BIG_ENDIAN
uint32_t a = 0x01234567;
uint32_t b = 0x89ABCDEF;
uint32_t result;
                                       //extract quadword
result = ixp_bytefield_dbl_extract(a, 2, b, 1, NOT_LOAD_CC);
              // result is 0x456789AB
```

Microengine C Example Usage Two

```
#define BIG_ENDIAN
uint32_t a[4] = {0x112233, 0x44556677, 0x8899AABB, 0xCCDDEEFF};
uint32_t result, a1=a[1], a2=a[2];
          // extract 3 bytes
result = ixp_bytefield_dbl_extract(a[1], 2, a[2], 0, NOT_LOAD_CC);
          // result is 0x667788
```

4.1.1.2 bytefield_decr()

Decrements a byte range within `in_src`. If decrementing the value zero, the bytefield wraps to ones in all bit positions. For example, decrementing `0x0000` results in `0xFFFF`.

Microengine Assembler Syntax

```
bytefield_decr(out_result, io_a, IN_START_BYTE, IN_END_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_decr(uint32_t io_a, IMMED IN_START_BYTE,
    IMMED IN_END_BYTE);
```

Input

`io_a`	The source 32-bit word.
`IN_START_BYTE`	The constant index of the beginning byte. Its value must be less than or equal to three and less than or equal to `IN_END_BYTE`.
`IN_END_BYTE`	The index of the last byte. Its value must be less than or equal to three and greater than or equal to `IN_START_BYTE`.

Output/Returns

`out_result`	The copy of `in_src` with a decremented byte field.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(data, 0x11220044);
bytefield_decr(result, data, 1, 2)
// result is 0x1121ff44
```

Microengine C Example Usage

```
uint32_t data = 0x11220044;// big-endian byte order is 0, 1, 2, 3
uint32_t result;
// decrement the field spanning bytes 1-2
result = ixp_bytefield_decr(data, 1, 2);
// result is 0x1121ff44
```

4.1.1.3 `bytefield_extract()`

Extracts a byte range from the source.

Microengine Assembler Syntax

```
bytefield_extract(out_result, in_src, IN_START_BYTE, IN_END_BYTE, \
    IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_extract(uint32_t in_src, IMMED IN_START_BYTE,
    IMMED IN_END_BYTE, IMMED IN_LOAD_CC);
```

Input

`in_src`	The source 32-bit word from which a byte field is to be extracted. For Microengine Assembler code, if this argument is in a GPR, it must be on the opposite bank from `out_result`.
`IN_START_BYTE`	The constant index of the beginning byte of the byte field to be extracted. Its value must be less than or equal to three and less than or equal to `IN_END_BYTE`.
`IN_END_BYTE`	The constant index of the last byte of the byte field to be extracted. Its value must be less than or equal to three and greater than or equal to `IN_START_BYTE`.
`IN_LOAD_CC`	A constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Output/Returns

`out_result` *or* Return Value	The byte range extracted from `in_src`.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(data, 0x1234567);
```

```
bytefield_extract(result, data, 2, 3)
// result is 0x4567
```

Microengine C Example Usage

```
uint32_t data = 0x01234567;
uint32_t result;
result = ixp_bytefield_extract(data, 2, 3, NO_LOAD_CC);
// result is 0x4567
```

4.1.1.4 `bytefield_incr()`

Increments a byte range within `io_a`. If a carry out occurs, the bytefield wraps to zero.

Microengine Assembler Syntax

```
bytefield_incr(io_a, IN_START_BYTE, IN_END_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_incr(uint32_t io_a, IMMED IN_START_BYTE,
    IMMED IN_END_BYTE);
```

Input

`IN_START_BYTE`	The constant index of the beginning byte of the byte field to be incremented. Its value must be less than or equal to three and less than or equal to `IN_END_BYTE`.
`IN_END_BYTE`	The constant index of the last byte of the byte field to be incremented. Its value must be less than or equal to three and greater than or equal to `IN_START_BYTE`.

Input/Output/Returns

`io_a` *or* Return Value	On input the source 32-bit word whose byte field is to be incremented. On output, the copy of `io_a` with an incremented byte field.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(data, 0x1122ff44);
bytefield_incr(result, data, 1, 2)
// result is 0x11230044
```

Microengine C Example Usage

```
uint32_t data = 0x1122ff44; // big-endian byte order is 0, 1, 2, 3
uint32_t result;
// increment the field spanning bytes 1-2
result =ixp_bytefield_incr(data, 1, 2);
// result is 0x11230044
```

4.1.1.5 `bytefield_insert()`

Inserts bytes of `in_b` specified by `IN_BYTE_MASK` into `io_a`.

Microengine Assembler Syntax

```
bytefield_insert(io_a, IN_BYTE_MASK, in_b, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_insert (uint32_t io_a, IMMED IN_BYTE_MASK,
    uint32_t in_b, IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The hex byte mask where the four least significant digits select the bytes to be extracted from `in_b` and inserted into `io_a`. • In Microengine C, it is a hex number. For example: 0x0001 selects righthand byte. 0x1000 selects lefthand byte. • In Microengine Assembler, it is a decimal number. For example: 0001 selects righthand byte. 1000 selects lefthand byte.
`in_b`	The source 32-bit word serving as the source of the byte field to be inserted into `io_a`. For Microengine Assembler code, if both `io_a` and `in_b` are located in GPRs, the GPRs must be stored in opposite banks from each other.
`IN_LOAD_CC`	A constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Input/Output/Returns

`io_a` *or* Return Value	On input the 32-bit word into which to insert a byte field. On return, the masked bytes of `io_b` have been inserted. For Microengine Assembler code, if both `io_a` and `in_b` are located in GPRs, the GPRs must be stored in opposite banks from each other.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
// insert y bytes 1,2 into x bytes 1.2
bytefield_insert(x, 0110, y, NO_LOAD_CC);
// x is now 0x11667744
```

Microengine C Example Usage

```
uint32_t x = 0x11223344;
uint32_t y = 0x55667788;
// insert y bytes 1,2 into x bytes 1.2
x = ixp_bytefield_insert(x, 0x0110, y, NO_LOAD_CC);
    // x is now 0x11667744
```

4.1.1.6 `bytefield_select()`

Returns the bytes of `in_data` specified by those bits of `IN_BYTE_MASK` whose value is one. Clears those bytes—that is, resets their value to zero—where the corresponding bit position in `IN_BYTE_MASK` has a value of zero.

Microengine Assembler Syntax

```
bytefield_select(out_result, IN_BYTE_MASK, in_src, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_select (IMMED IN_BYTE_MASK, uint32_t in_src,
    IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant byte mask where the four least significant digits specify the bytes to be extracted from `in_src`. • In Microengine C, it is a hex number. For example: 0x0001 selects righthand byte. 0x1000 selects lefthand byte. • In Microengine Assembler, it is a decimal number. For example: 0001 selects righthand byte. 1000 selects lefthand byte.
`in_src`	The 32-bit word that is the source of the byte field. For Microengine Assembler code, if this argument is stored in a GPR, it must be stored in the opposite bank from the location of `out_result`.
`IN_LOAD_CC`	A constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Output/Returns

`out_result` ***or*** Return Value	The selected byte fields.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(y, 0x01234567);
bytefield_select(result, 0110, y, NO_LOAD_CC);
// result is 0x234500
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t result;
result = ixp_bytefield_select(0x0110, y, NO_LOAD_CC);
// result is 0x234500
```

4.1.1.7 `bytefield_br_eq()`

Compares a byte field, specified by a byte range, to a constant and branches if they are equal.

Microengine Assembler Syntax

```
bytefield_br_eq(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL, \
    target_label);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `IN_START_BYTE` to `IN_END_BYTE`.
`target_label`	The label to serve as the branch target.

Output/Returns

Function Behavior	Based on the input arguments, the branch is taken or not taken.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffffff);
bytefield_br_eq(a, 1, 3, 0xffffff, lab#)
```

4.1.1.8 `bytefield_br_gtr()`

Compares a byte field—specified by a byte range—to a constant and branches if the field is greater than the constant.

Microengine Assembler Syntax

```
bytefield_br_gtr(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL, \
    target_label);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `in_start_byte` to `in_end_byte`.
`target_label`	The label to serve as the branch target.

Output/Returns

Function Behavior	Based on the input arguments, the branch is taken or not taken.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffff11);
bytefield_br_gtr(a, 1, 3, 0xffffff, lab#)
```

4.1.1.9 `bytefield_br_gtreq()`

Compares a byte field—specified by a byte range—to a constant and branches if the field is greater than or equal to the constant.

Microengine Assembler Syntax

```
bytefield_br_gtreq(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL, \
    target_label);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `IN_START_BYTE` to `IN_END_BYTE`.
`target_label`	The label to serve as the branch target.

Output/Returns

Function Behavior	Based on the input arguments, the branch is taken or not taken.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffff11);
bytefield_br_gtreq(a, 1, 3, 0xffffff, lab#)
```

4.1.1.10 `bytefield_br_less()`

Compares a byte field—specified by a byte range—to a constant and branches if the field is less than the constant.

Microengine Assembler Syntax

```
bytefield_br_less(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL, \
    target_label);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `IN_START_BYTE` to `IN_END_BYTE`.
`target_label`	The label to serve as the branch target.

Output/Returns

Function Behavior	Based on the input arguments, the branch is taken or not taken.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffff11);
bytefield_br_less(a, 1, 3, 0xffffff, lab#)
```

4.1.1.11 bytefield_br_lesseq()

Compares a byte field—specified by a byte range—to a constant and branches if the field is less than or equal to the constant.

Microengine Assembler Syntax

```
bytefield_br_lesseq(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL, \
    target_label);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `IN_START_BYTE` to `IN_END_BYTE`.
`target_label`	The label to serve as the branch target.

Output/Returns

Function Behavior	Based on the input arguments, the branch is taken or not taken.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffff11);
bytefield_br_lesseq(a, 1, 3, 0xffffff, lab#)
```

4.1.1.12 `bytefield_comp()`

Compares a byte field—specified by a byte range—to a constant and sets the condition code accordingly.

Microengine Assembler Syntax

```
bytefield_comp(in_src, IN_START_BYTE, IN_END_BYTE, COMPARE_VAL);
```

Microengine C Syntax

Not supported.

Input

`in_src`	A general-purpose register or read transfer register containing a byte field to compare.
`IN_START_BYTE`	The starting byte position whose value is zero through three.
`IN_END_BYTE`	The ending byte position whose value is zero through three.
`COMPARE_VAL`	The constant value to compare with the field within `in_src` specified by the range, `IN_START_BYTE` to `IN_END_BYTE`.

Output/Returns

Return Value	Condition code is set based on the inputs.

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
immed32(a, 0x01ffffff);
bytefield_comp(a, 1, 3, 0xffffff)
beq[my_label#]
```

4.1.1.13 `bytefield_shf_left_insert()`

Inserts bytes of `in_b` specified by `IN_BYTE_MASK` into `io_a` after shifting `in_b` left by `IN_SHIFT_AMT`. In the Microengine C implementation the value of `io_a` is returned.

Microengine Assembler Syntax

```
bytefield_shf_left_insert(io_a, IN_BYTE_MASK, in_b, IN_SHIFT_AMT, \
    IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_shf_left_insert (
    uint32_t io_a,
    IMMED IN_BYTE_MASK,
    uint32_t in_b,
    IMMED IN_SHIFT_AMT,
    IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant hex byte mask where the four least significant digits select bytes from `in_b`. For example: • 0x1—selects the righthand byte • 0x1000—selects the lefthand byte
`in_b`	The source 32-bit word. In Microengine Assembler code if this is a GPR, it must be stored on the opposite bank from the argument `io_a`.
`IN_SHIFT_AMT`	The constant amount by which to shift `in_b`.
`IN_LOAD_CC`	The constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Input/Output

`io_a` *or* Return Value	On input, the variable into which to insert bytes. On output the modified value. In Microengine Assembler code, this argument must be stored in a GPR.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
// insert y bytes 1,2 into x bytes 1.2
bytefield_shf_left_insert(x, 0110, y, 4, 0);
    // x is now 0x11677844
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t x = 0xabcdefab;
//
//insert bytes 1&2 of y into x after shifting y left by 4bit position.
x = ixp_bytefield_shf_left_insert(x, 0x0110, y, 4, NO_LOAD_CC);
    // x is now 0xab3456ab
```

4.1.1.14 `bytefield_shf_right_insert()`

Shifts `in_b` right by `IN_SHIFT_AMT` then inserts bytes of `in_b` specified by `IN_BYTE_MASK` into `io_a`.

Microengine Assembler Syntax

```
bytefield_shf_right_insert(io_a, IN_BYTE_MASK, in_b, IN_SHIFT_AMT, \
    IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_shf_right_insert (uint32_t io_a, IMMED
IN_BYTE_MASK,
    uint32_t in_b, IMMED IN_SHIFT_AMT, IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant hex byte mask where the four least significant digits select bytes from `in_b`. For example: • 0x1—selects the righthand byte • 0x1000—selects the lefthand byte
`in_b`	The source 32-bit word. In Microengine Assembler code, if this argument is stored in a GPR, it must on the opposite bank from the argument `io_a`
`IN_SHIFT_AMT`	The constant amount to shift `in_b` by.
`IN_LOAD_CC`	The constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Input/Output/Returns

`io_a` ***or*** Return Value	On input, the variable into which to insert bytes. On output the modified value. For Microengine Assembler code, this argument must be stored in a GPR.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
    // insert y bytes 1,2 into x bytes 1.2
bytefield_shf_right_insert(x, 0110, y, 4, 0);
    // x is now 0x11566744
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t x = 0xabcdefab;
    //insert bytes 1&2 of y into x after shifting y right by 8.
x = ixp_bytefield_shf_right_insert(x, 0x0110, y, 8, NO_LOAD_CC);
    // x is now 0xab0123ab
```

4.1.1.15 `bytefield_clr_insert()`

Inserts bytes of `in_src` specified by `IN_BYTE_MASK` into `out_result`, the return value. Clears `out_result` prior to performing the insert.

Microengine Assembler Syntax

```
bytefield_clr_insert(out_result, IN_BYTE_MASK, in_src, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_clr_insert (IMMED IN_BYTE_MASK,
    uint32_t in_src, IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant hex byte mask where the four least significant digits select bytes from `in_b`. For example: • 0x1—selects the righthand byte • 0x1000—selects the lefthand byte
`in_b`	The source 32-bit word. In Microengine Assembler code, if this argument is stored in a GPR, it must be on the opposite bank from the location of `out_result`.
`IN_LOAD_CC`	The constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Output/Returns

`out_result` ***or*** Return Value	On output the bytes of `in_src` specified by `IN_BYTE_MASK`.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
    // insert y bytes 1,2 into x bytes 1.2
bytefield_clr_insert(x, 0110, y, 4, 0);
    // x is now 0x00667700
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t x = 0xabcdefab;
    //insert bytes 1&2 of y into x bytes of 1,2 after clearing x.
x = ixp_bytefield_clr_insert(0x0110, y, NO_LOAD_CC);
    // x is now 0x00cdef00
```

4.1.1.16 `bytefield_clr_shf_left_insert()`

Shifts `in_src` left by `IN_SHIFT_AMT` then inserts the shifted bytes of `in_src` specified by `IN_BYTE_MASK` into `out_result`. Clears `out_result` before performing the insertion.

Microengine Assembler Syntax

```
bytefield_clr_shf_left_insert(out_result, IN_BYTE_MASK, in_src, \
    IN_SHIFT_AMT, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_clr_shf_left_insert (
    IMMED IN_BYTE_MASK,
    uint32_t in_src,
    IMMED IN_SHIFT_AMT,
    IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant hex byte mask where the four least significant digits select bytes from `in_b`. For example: • 0x1—selects the right-most byte • 0x1000—selects the left-most byte
`in_src`	The source 32-bit word with which to combine with the inserted bytes. In Microengine Assembler code, if both `out_result` and in_src are stored in GPRs, they must be located in opposite banks from each other.
`IN_SHIFT_AMT`	The constant amount by which to shift `in_src`.
`IN_LOAD_CC`	The constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Output/Returns

`out_result` *or* Return Value	The value of `in_src` modified by shifting left by `IN_SHIFT_AMT` then masking with `IN_BYTE_MASK`.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
    // insert y bytes 1,2 into x bytes 1.2
bytefield_clr_shf_left_insert(x, 0110, y, 4, 0);
    // x is now 0x00677800
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t x = 0xabcdefab;
    //insert bytes 1&2 of y into x after
    //clearing x & shifting y left by 4.
x = ixp_bytefield_clr_shf_left_insert(0x0110, y,4, NO_LOAD_CC);
    // x is now 0x00345600
```

4.1.1.17 `bytefield_clr_shf_right_insert()`

Shifts `in_src` right by `IN_SHIFT_AMT` then inserts the shifted bytes of `in_src` specified by `IN_BYTE_MASK` into `out_result`. Clears `out_result` before performing the insertion.

Microengine Assembler Syntax

```
bytefield_clr_shf_right_insert(out_result, IN_BYTE_MASK, in_src, \
    IN_SHIFT_AMT, IN_LOAD_CC);
```

Microengine C Syntax

```
INLINE uint32_t ixp_bytefield_clr_shf_right_insert (
    IMMED IN_BYTE_MASK,
    uint32_t in_src,
    IMMED IN_SHIFT_AMT,
    IMMED IN_LOAD_CC);
```

Input

`IN_BYTE_MASK`	The constant hex byte mask where the four least significant digits select bytes from `in_b`. For example: • 0x1—selects the righthand byte • 0x1000—selects the lefthand byte
`in_src`	The source 32-bit word into whose value to insert bytes.
`IN_SHIFT_AMT`	The constant amount by which to shift `in_src`.
`IN_LOAD_CC`	The constant specifying whether or not to load the ALU condition codes based on the result of the operation. This value is one of: • `NO_LOAD_CC`—do not load the condition code This is the suggested value when an application must select byte or bytes without any side effects. • `DO_LOAD_CC`—load condition code Side effects may occur as a result of specifying this value.

Input/Output

`out_result` ***or*** Return Value	First cleared, on return this argument contains the shifted bytes of `in_src`.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
immed32(x, 0x11223344);
immed32(y, 0x55667788);
    // insert y bytes 1,2 into x bytes 1.2
bytefield_clr_shf_right_insert(x, 0110, y, 4, 0);
    // x is now 0x00566700
```

Microengine C Example Usage

```
uint32_t y = 0x01234567;
uint32_t x = 0xabcdefab;
    // insert bytes 1&2 of y into x
    // after clearing x & shifting y right by 4.
x = ixp_bytefield_clr_shf_right_insert(0x0110, y, 4, NO_LOAD_CC);
    // x is now 0x00123400
```

4.2 crc

The `crc` interface is used to read or write the cyclic redundancy check (CRC) residue at the CRC unit, and perform a CRC calculation. For the Intel® IXP2400, IXP2800 and IXP2850 Network Processorsthe CRC unit is local to the microengine. See one of the following manuals for more information:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Table 4-2. `crc` API

Name	Description
`crc_load_crc10_table()`	This macro populates the CRC lookup table.
`crc_10()`	Performs a CRC-10 computation.
`crc_ccitt()`	Performs a CRC CCITT computation.
`crc_32()`	Performs a CRC-32 computation.
`crc_read()`	Reads the CRC residue from previous CRC computation.
`crc_write()`	Writes (initializes) the CRC unit with a CRC residue.

4.2.1 API Functions

4.2.1.1 crc_load_crc10_table()

This macro populates the CRC lookup table. It requires a local memory base address indicating where to populate the table. It uses 128 32-bit words of local memory starting from the local memory base address.

Note: This macro should be called before any CRC-10 computation is performed.

After all CRC-10 computations have been performed the local memory containing the lookup table can be reused for other purposes—if local memory is reused no other CRC-10 computations can be performed until a new CRC-10 table is loaded.

The table values are generated using the CRC10 generator polynomial:

$$x^{10} + x^9 + x^5 + x^4 + x + 1. \text{ (0x633)}$$

Note: This macro is for *Microengine Assembler use only.*

Microengine Assembler Syntax

```
crc_load_crc10_table(local_mem_lookuptable_base_address);
```

Input

`in_base_addr`	The local memory base address specifying where to load the lookup table.

Estimated Size

One hundred and thirty instructions.

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
crc_load_crc10_table(0);     //Need to be called only once.
                             //Could be done at init time if required.
immed32(src, 0x33343132);
crc_write(0);
crc_10(out_reg, src, big_endian, 0, 3);
```

4.2.1.2 `crc_10()`

Performs a CRC-10 computation. The previous residue must be in the CRC unit prior to performing this operation. Each time this operation is performed, the residue is updated at the CRC unit. `in_start_byte` must be less than `in_end_byte`.

Note: Loading the CRC lookup table through a call to `crc_load_crc10_table()` is only required for Microengine Assembler implementations.

Note: There is no IXP hardware support for this operation—it requires a table lookup.

Microengine Assembler Syntax

```
crc_10 (out_residue, in_data, ENDIAN, IN_START_BYTE, IN_END_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_crc_10 (
    uint32_t in_data,
    endian_t ENDIAN,
    IMMED IN_START_BYTE,
    IMMED IN_END_BYTE);
```

Input

`in_data`	The data for which a CRC operation is requested.
`ENDIAN`	The byte-order setting for this operation: • `little_endian`—indicates little endian order data Used when the data is in little endian mode, so the bytes are swapped before the CRC is performed. • `big_endian`—indicates big endian order data Used when the data is in big endian mode, so the bytes are not swapped before the CRC is performed.
`in_start_byte`	The index of the beginning byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte
`in_end_byte`	The index of the end byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte

Output/Returns

`out_residue` *or* Return Value	The resulting residue.

Estimated Size

- For Microengine Assembler:
  - For 1 byte—30 instructions
  - For 2 bytes—45 instructions
  - For 3 bytes—60 instructions
  - For CRC-10, four bytes—75 instructions
- For Microengine C:
  - For 1 byte—35 instructions
  - For 2 bytes—60 instructions
  - For 3 bytes—85 instructions
  - For 4 bytes—100 instructions

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
crc_load_crc10_table(0);     //Need to be called only once.
                             //Could be done at init time if required.
immed32(src, 0x33343132);
crc_write(0);
crc_10(out_reg, src, big_endian, 0, 3);
```

Microengine C Example Usage

```
uint32_t residue = 0xffffffff;
uint32_t data0 = 0x01234567;
uint32_t data1 = 0x89abcdef;
ixp_crc_write(residue);
residue = ixp_crc_10(data0, big_endian, 1, 3);
residue = ixp_crc_10(data1, big_endian, 0, 3);
```

4.2.1.3 `crc_ccitt()`

Performs a CRC CCITT computation. The previous remainder must be in the CRC unit prior to performing this operation. Each time this operation is performed, the remainder is updated at the CRC unit. The value of `in_start_byte` must be less than the value of `in_end_byte`.

Microengine Assembler Syntax

```
crc_ccitt (out_residue, in_data, ENDIAN, IN_START_BYTE, IN_END_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_crc_ccitt (
    uint32_t in_data,
    endian_t ENDIAN,
    IMMED IN_START_BYTE,
    IMMED IN_END_BYTE);
```

Input

`in_data`	The data on which a CRC operation is to be performed.
`ENDIAN`	Specifies the byte ordering with the values: • `little_endian`—indicates little endian byte order data Used when the data is in little endian mode, so the bytes are swapped before the CRC is performed. • `big_endian`—indicates big endian order data Used when the data is in big endian mode, so the bytes are not swapped before the CRC is performed.
`in_start_byte`	The index of the beginning byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte
`in_end_byte`	The index of the ending byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte

Output/Returns

`out_residue` *or* Return Value	The resulting CRC residue.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
immed32(src, 0x33343132);
crc_write(0);
crc_ccitt(out_reg, src, big_endian, 0, 3);
```

Microengine C Example Usage

```
uint32_t residue = 0xffffffff;
uint32_t data0 = 0x01234567;
uint32_t data1 = 0x89abcdef;
ixp_crc_write(residue);
residue = ixp_crc_ccitt(data0, big_endian, 1, 3);
residue = ixp_crc_ccitt(data1, big_endian, 0, 3);
```

4.2.1.4 `crc_32()`

Performs a CRC-32 computation. The previous residue must be in the CRC unit prior to performing this operation. Each time this operation is performed, the residue is updated at the CRC unit. The value of `in_start_byte` must be less than the value of `in_end_byte`.

Microengine Assembler Syntax

```
crc_32 (out_residue, in_data, ENDIAN, IN_START_BYTE, IN_END_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_crc_32 (
    uint32_t in_data,
    endian_t ENDIAN,
    IMMED IN_START_BYTE,
    IMMED IN_END_BYTE);
```

Input

`in_data`	The data on which a CRC operation is performed.
`ENDIAN`	Specifies the byte ordering with the values: • `little_endian`—indicates little endian byte order data Used when the data is in little endian mode, so the bytes are swapped before the CRC is performed. • `big_endian`—indicates big endian order data Used when the data is in big endian mode, so the bytes are not swapped before the CRC is performed.
`in_start_byte`	The index of the beginning byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte
`in_end_byte`	The index of the ending byte. When `ENDIAN` is big-endian the coding is: • `0`—specifies the left-most byte • `3`—specifies the right-most byte

Output/Returns

`out_residue` ***or*** Return Value	The resulting CRC residue.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
immed32(src, 0x33343132);
crc_write(0);
crc_32(out_reg, src, big_endian, 0, 3);
```

Microengine C Example Usage

```
uint32_t residue = 0xffffffff;
uint32_t data0 = 0x01234567;
uint32_t data1 = 0x89abcdef;
ixp_crc_write(residue);
residue = ixp_crc_32(data0, big_endian, 1, 3);
residue = ixp_crc_32(data1, big_endian, 0, 3);
```

4.2.1.5 `crc_read()`

Reads the CRC residue from a previous CRC computation.

Microengine Assembler Syntax

```
crc_read (out_residue);
```

Microengine C Syntax

```
INLINE uint32_t ixp_crc_read ();
```

Output/Returns

`out_residue` *or* Return Value	The CRC residue from the previous computation.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
immed32(src, 0x33343132);
crc_write(0);
crc_32(out_reg, src, big_endian, 0, 3);
crc_read(out_reg);
```

Microengine C Example Usage

```
uint32_t residue = 0xffffffff;
uint32_t data0 = 0x01234567;
uint32_t data1 = 0x89abcdef;
ixp_crc_write(residue);
ixp_crc_32(data0, big_endian, 1, 3);
ixp_crc_32(data1, big_endian, 0, 3);
residue = ixp_crc_read();
```

4.2.1.6 `crc_write()`

Writes a CRC residue to the CRC unit. Use this operation to initialize the CRC unit or to provide the unit with the residue from a previous CRC computation.

Microengine Assembler Syntax

```
crc_write (in_residue);
```

Microengine C Syntax

```
INLINE void ixp_crc_write (uint32_t in_residue);
```

Input

`in_residue`	Either the initial residue or residue from a previous CRC computation.

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
.reg src out_reg in_remainder
immed32(src, 0x33343132);
crc_write(0);
crc_32(out_reg, src, big_endian, 0, 3);
crc_read(out_reg);
```

Microengine C Example Usage

```
uint32_t residue = 0xffffffff;
uint32_t data0 = 0x01234567;
uint32_t data1 = 0x89abcdef;
ixp_crc_write(residue);
ixp_crc_32(data0, big_endian, 1, 3);
ixp_crc_32(data1, big_endian, 0, 3);
residue = ixp_crc_read();
```

4.3 Hash Utilities

The *hash* interface is used to access table entries where the index to the table is so large that a direct memory access is not feasible. For example a 16-bit SRAM array index would require 256KB of 32-bit data elements, an 18-bit index would require 1MB of 32-bit data elements, and so on. Hashing is used to enable large indices to store and lookup data in less memory space. The IXP hardware hash feature is used by this interface to reduce conflicts in the lookup.

This table lookup functionality is typically used for bridge or connection lookup-table search. However, the functionality is generic and can be used for any type of large-index lookup.

For example:

- MAC address (48-bit) lookup. Source and destination MAC address lookups may be performed in parallel for standard bridging or Layer 2 filtering.
- IP 5-Tuple (104-bit) lookup. This uses two HDBM tables. The index is comprised of IP source address (32 bits), IP source port (16 bits), IP destination address (32 bits), IP destination port (16 bits), and IP protocol (8 bits).

4.3.1 Hash Algorithm

The hash lookup uses different algorithm based on the `TRIE_TYPE` specified. This section describes the `HASH_16_4` and `HASH_FOLD_16` lookup algorithms.

4.3.1.1 `HASH_16_4 Index`

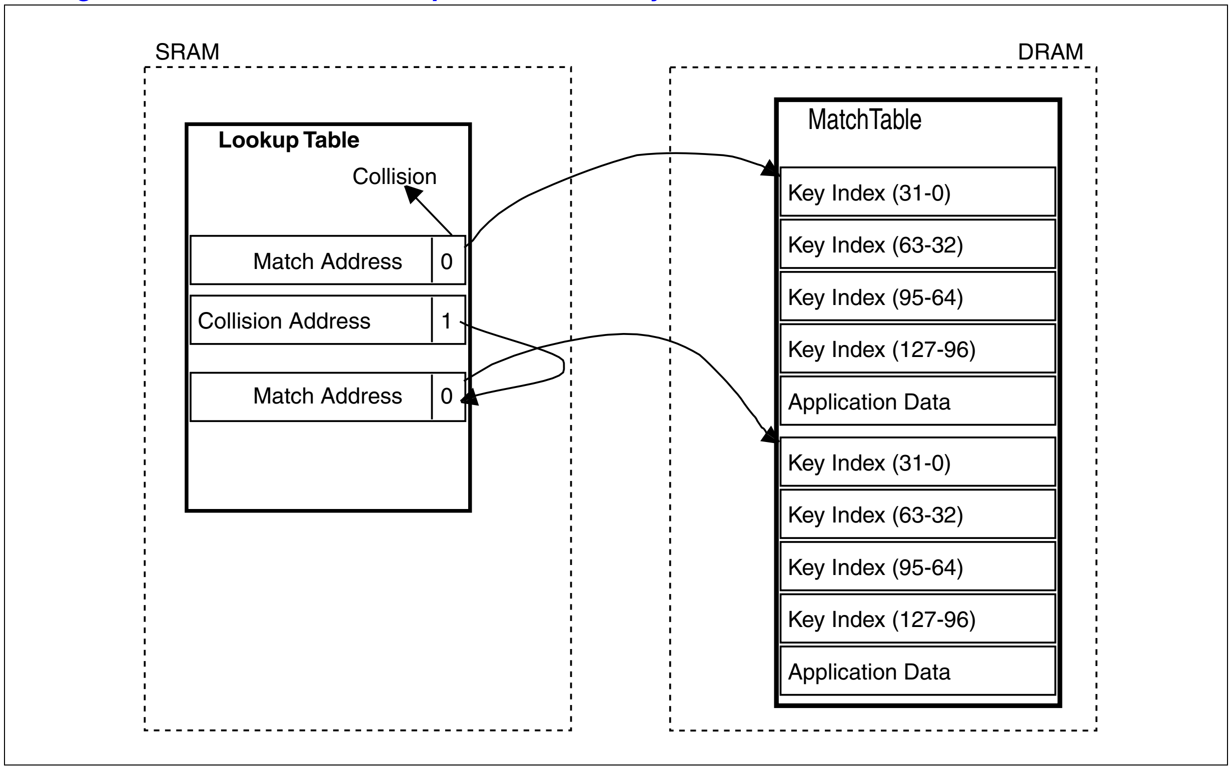
The `Hash_16_4` lookup uses a multi-table database and a sub-set of the hashed key to index tables in SRAM that contain an index to either an entry in SDRAM or to a trie table in SRAM to resolve collisions. The size of the primary-table is 16 bits. It uses two freelists to manage the collision tables (in SRAM) and the key-data elements—in SDRAM.

A lookup of an entry is done by hashing the input-key and then extracting the primary-index (16 bits) from the hashed result. The primary-index is then used to index the primary table. If there is an entry—the primary value read not equal to zero—and the collision bit is not set, then the primary-entry index is extracted from the primary value and is used to retrieve the data and key from SDRAM.

The input key is then compared to the stored key to verify a match. If the collision bit was set, then the primary-entry index is used to index the collision table. A fixed sub-set of the four bits from the input key generates the collision table index. On subsequent collisions, the next sub-set of four bits is extracted from the input key to index the next level of collision table.

The following illustrates lookup for 128-bit key.

Figure 4-1. HASH_16_4 Lookup for a 128-Bit Key

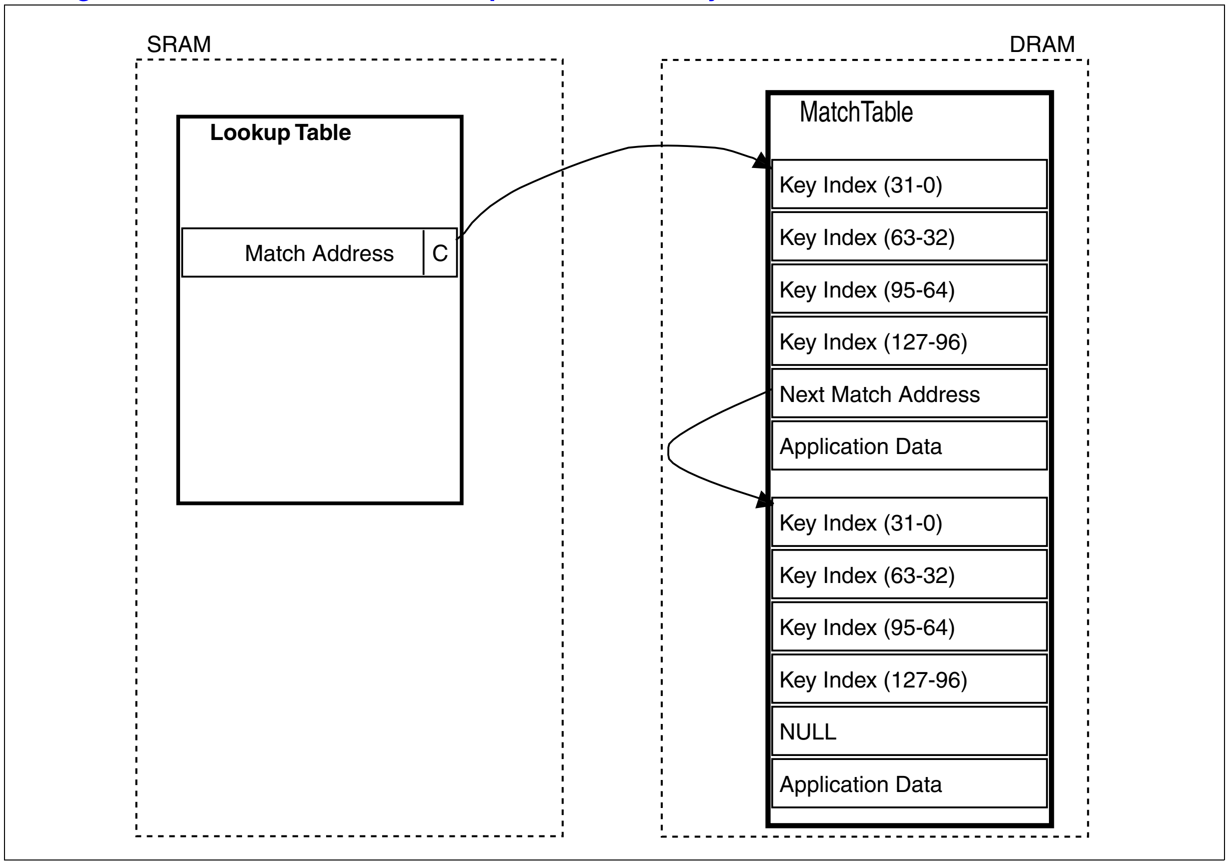


4.3.1.2 HASH_FOLD_16 Index

The hash result in this method is folded—that is, exclusive ored—until all the bits in the hash result are reduced to the size of the table being indexed. When more than one 5-tuple rule resolves to a single entry, the collision on the entry results in the respective entries being chained to the same hash bucket. Each entry in the table stores the 5-tuple data corresponding to that entry. During lookup, since the hash result is folded, finding a match in the hash table does not guarantee a perfect match. To resolve a perfect match, 5-tuple data (pre-hash) needs to be compared with the 5-tuple index stored in the matched entry.

The following illustrates lookup for 128-bit key.

Figure 4-2. HASH_FOLD_16 Lookup for a 128-Bit Key



The number of bits used to index into the hash table determines the lookup table size. Each entry in the primary table is 32 bits long. The *C* field indicates collision, the *Next* field contains the address to the next chained entry, and the *Match addr* field contains the address to the index data stored in the match table. The lookup table is usually stored in SRAM and the match table is usually stored in SDRAM or in SRAM if space permits.

4.3.1.3 Creating an entry

The following steps create a hash entry.

1. If the indexed entry in the lookup table is unoccupied, reset the *C* and *Next* fields. Get an entry from a freelist for the match table to store the original 5-tuple index and filter action. The *Match Addr* points to the SRAM or DRAM entry.
2. If the indexed entry in the lookup table is occupied, set the *C* field if it is not already set to indicate a collision.
   a. If the *Next* field is unset, get a 32-bit entry from a freelist for chained entries. Set the address of the just obtained freelist entry in the *Next* field (entry in lookup table). Get an entry from a freelist for the match table to store the original 5-tuple index and filter action for the colliding entry.
   b. If the *Next* field is set, walk the list of chained entries until an entry whose *Next* field is unset. Then, perform the previous substep (a).

4.3.1.4 Lookup an entry

1. Use the *Match Addr* value to access the match table. Compare the pre-hashed 5-tuple data with what is stored in the match table entry.
   a. If they match, it is a perfect 5-tuple match.
   b. If they do not match and the *C* field in the lookup table entry is set, use the address in the *Next* field to access the chained entry. Then perform step 1.

4.3.2 `hash`

Table 4-3. `hash` API

Name	Description
`hash_dual_lookup()`	Performs a dual hash lookup without hardware hash using trie. Two indexes are searched in parallel.
`hash_init()`	Initializes the hash multiplier.
`hash_lookup()`	Performs a single hash lookup using trie.
`hash_translate()`	Performs a hash translation on the lookup index using the hardware polynomial conversion.

4.3.2.1 API Functions

4.3.2.1.1 `hash_dual_lookup()`

Performs a lookup on two table entries in parallel using up to a 128-bit index for each. The reads of the trie structure are done in parallel. This utilizes the trie structure and tables written by the Intel® XScale™ core hash table database manager. See Section 4.3.1, "Hash Algorithm" for details.

A typical use of this call is the parallel lookup of source and destination 48-bit Ethernet MAC addresses.

Note: For *Microengine Assembler only*, if `__DONT_TRANSLATE_KEYS` is defined, then no hash translation is performed on the keys.

Microengine Assembler Syntax

```
hash_dual_lookup (out_index0, out_index1, in_key0, \
in_key1, KEY_SIZE, trie_base_addr, TRIE_TYPE, KEY_DATA_SD_BASE);
```

Microengine C Syntax

```
INLINE void ixp_hash_dual_lookup (
    uint32_t *out_index0,
    uint32_t *out_index1,
    _declspec(sram_write_reg) void *in_key0,
    __declspec(sram_write_reg) void *in_key1,
    uint32_t KEY_SIZE,
    volatile __declspec(sram) void *trie_base_addr,
    uint32_t TRIE_TYPE,
    uint32_t KEY_DATA_SD_BASE);
```

Input

`in_key0`	The buffer of write transfer registers with the starting first index size of up to 128 bits in length.
`in_key1`	The buffer of write transfer registers with the starting second index size of up to 128 bits in length.
`KEY_SIZE`	The constant indicating the length, in bits, of `in_key0` and `in_key1`.
`trie_base_addr`	The address of the trie table.
`TRIE_TYPE`	Specifies the index bits used to address the hash trie with a value of either `HASH_16_4` or `HASH_FOLD_16`. • `HASH_16_4`—first lookup uses 16 bits of index, subsequent lookups use 4 bits of index • `HASH_FOLD_16`—first lookup exclusive ors the initial index to reduce it by half, then performs a table lookup using 16 bits of half-index, with subsequent chain search until there is no collision.
`KEY_DATA_SD_BASE`	The DRAM base table address.

Output

`out_index0`	The index of a hash table entry associated with `in_key0` if the lookup is successful and zero otherwise—that is, the entry was not found.
`out_index1`	The index of a hash table entry associated with `in_key1` if the lookup is successful and zero otherwise—that is, the entry was not found.

Estimated Size

- For Microengine Assembler:
  - For a `TRIE_TYPE` of `HASH_16_4`—32 to 37 instructions (two SRAM accesses and two DRAM accesses) plus 22 instructions times the number of iterations (two SRAM accesses per iteration)
  - For a `TRIE_TYPE` of `HASH_FOLD_16`—32 to 40 instructions (two SRAM accesses) plus 28 instructions times the number of iterations (one DRAM access per iteration)
- For Microengine C:
  - For a `TRIE_TYPE` of `HASH_16_4`—70 to 86 instructions (two SRAM accesses and two DRAM accesses) plus 60 instructions times the number of iterations (one SRAM access per iteration)
  - For a `TRIE_TYPE` of `HASH_FOLD_16`—38 to 48 instructions (two SRAM accesses) plus 34 instructions times the number of iterations (one DRAM access per iteration)

Microengine Assembler Example Usage

```
hash_dual_lookup(source_addr_index, dest_addr_index, &source_addr, \
       &dest_addr, trie_addr, HASH16_4, DRAM_TABLE_BASE);
```

Microengine C Example Usage

```
uint32_t out_index1, out_index2;
__declspec(sram_write_reg) uint32_t in_key1[4], in_key2[4];
//
in_key1[0] = 1000;
in_key1[1] = 0;
in_key2[0] = 0x12345678;
in_key2[1] = 0x9abcdef1;
//
ixp_hash_dual_lookup(
    &out_index1,
    &out_index2,
    &in_key1,
    &in_key2,
    64,
    (volatile __declspec(sram) void *) trie_addr,
    HASH_16_4,
    DRAM_TABLE_BASE);
```

4.3.2.1.2 `hash_init()`

Initializes the hash multiplier of the hardware hash translation unit. See one of the following manuals for a description of this hardware functionality:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Note: Hash trie tables must be initialized by the hash table database manager running on the Intel® XScale™ core.

Microengine Assembler Syntax

```
hash_init (in_multiplier, MULTIPLIER_SIZE);
```

Microengine C Syntax

```
INLINE void ixp_hash_init (
    __declspec(sram_write_reg) void *in_multiplier,
    hw_hash_t MULTIPLIER_SIZE);
```

Input

`in_multiplier`	The buffer of write transfer registers containing the multiplier.
`MULTIPLIER_SIZE`	The size of the multiplier as defined in `hw_hash_t`. One of {`HW_HASH_48`, `HW_HASH_64`, or `HW_HASH_128`}. See Section 2.6.4, “hw_hash_t.”

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
xbuf_alloc($multiplier, 2);
move($multiplier[0], 0);
move($multiplier[1], 1);
hash_init($multiplier[0], HW_HASH_64);
xbuf_free($multiplier);
```

Microengine C Example Usage

```
__declspec(sram_write_reg) uint32_t multiplier[2];
multiplier[0] = 0;
multiplier[1] = 1;
ixp_hash_init(multiplier, HW_HASH_64);
```

4.3.2.1.3 `hash_lookup()`

Performs a lookup on a table entry using up to a 128-bit index and tables written by Intel® XScale™ core hash table database manager. See Section 4.3.1, “Hash Algorithm” for a description of the algorithm.

Note: For *Microengine Assembler only*, if `__DONT_TRANSLATE_KEYS` is defined then no hash translation is performed on the keys.

Microengine Assembler Syntax

```
hash_lookup (out_table_index, in_big_index, BIG_INDEX_SIZE,
    trie_addr, TRIE_TYPE, KEY_DATA_SD_BASE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_hash_lookup(
    __declspec(sram_write_reg) void *in_big_index,
    uint32_t BIG_INDEX_SIZE,
    volatile __declspec(sram) void* trie_addr,
    uint32_t TRIE_TYPE,
    uint32_t KEY_DATA_SD_BASE);
```

Input

`in_big_index`	The buffer of write transfer registers with the starting first index whose size is up to 128 bits in length.
`trie_addr`	The address of the trie table.
`TRIE_TYPE`	Specifies the index bits used to address the hash trie. This value is one of {`HASH_16_4`, `HASH_FOLD_16`}. • `HASH_16_4`—first lookup uses 16 bits of index and subsequent lookups use 4 bits of index • `HASH_FOLD_16`—first lookup folds the index by XORing with itself to reduce it by half, then performs a table lookup using 16 bits of half-index, with subsequent chain search until there is no collision
`KEY_DATA_SD_BASE`	The DRAM base table address.

Output/Returns

`out_table_index` ***or*** Return Value	Returns the index of a hash table entry if successful and zero otherwise—that is if no entry is found.

Estimated Size

- For Microengine Assembler:
  - — For a `TRIE_TYPE` of `HASH_16_4`—14 to 16 instructions (one DRAM access and one DRAM access) plus eleven instructions times the number of iterations (one SRAM access per iteration)
  - — For a `TRIE_TYPE` of `HASH_FOLD_16`—12 to 19 instructions (one SRAM access) plus fourteen instructions times the number of iterations (one DRAM access per iteration)
- For Microengine C:
  - — For a `TRIE_TYPE` of `HASH_16_4`—35 to 46 instructions (one SRAM access and one DRAM access) plus 30 instructions times the number of iterations (one SRAM access per iteration)
  - — For a `TRIE_TYPE` of `HASH_FOLD_16`—19 to 28 instructions (one SRAM access) plus 17 instructions times the number of iterations (one DRAM access per iteration)

Microengine Assembler Example Usage

```
hash_lookup(table_entry_index, $wide_index[0], 102, trie_addr, HASH16_4,\
    DRAM_TABLE_BASE);
```

Microengine C Example Usage

```
__declspec(sram_write_reg) uint32_t in_key[2];
uint32_t index;
in_key[0] = 0x12345678;
in_key[1] = 0x9abcdef1;
index = ixp_hash_lookup(
    &in_key,
    64,
    (volatile __declspec(sram) void *)TRIE_ADDR_BASE,
    HASH_16_4,
    KEY_DATA_SD_BASE);
```

4.3.2.1.4 `hash_translate()`

Translates a large index, up to 128 bits, using the hardware hash translation unit. See one of the following manuals for more information about this hardware functionality:

- *Intel® IXP2400 Network Processor Hardware Reference Manual*,
- *Intel® IXP2800 Network Processor Hardware Reference Manual*
- *Intel® IXP2850 Network Processor Hardware Reference Manual*

Microengine Assembler Syntax

```
hash_translate (out_index, in_index, INDEX_SIZE);
```

Microengine C Syntax

```
INLINE void ixp_hash_translate (
    __declspec(sram_read_reg)void *out_index,
    __declspec(sram_write_reg) void *in_index,
    uint32_t count,
    SIGNAL_PAIR *REQ_SIG,
    SIGNAL_MASK in_wakeup_sigs,
    hw_hash_t INDEX_SIZE);
```

Input

`REQ_SIG`	The requested signal. See Section 2.6.13, "Signal Arguments and Usage."
`in_wakeup_sigs`	The list of signals causing the thread to swap or wakeup. See Section 2.6.13, "Signal Arguments and Usage."
`count`	The number of hash operations.
`in_index`	The buffer of write transfer registers containing the index to be translated.
`INDEX_SIZE`	Specifies the size of index which is one of {`HW_HASH_48`, `HW_HASH_64`, `HW_HASH_128`}. See Section 2.6.4, "hw_hash_t."

Output

`out_index`	The buffer of read transfer registers with the translated index.

Estimated Size

- For Microengine Assembler—one to two instructions
- For Microengine C—three to six instructions

Microengine Assembler Example Usage

```
hash_translate($jumbalaya, $big_index, HW_HASH_128);
```

Microengine C Example Usage

```
__declspec(sram_read_reg) unsigned int hash_output[2];
__declspec(sram_write_reg) unsigned int hash_input[2];
SIGNAL_PAIR hash_done;
SIGNAL_MASK sig_mask;
//
sig_mask = __signals(&hash_done);
hash_input[0] = 0x12345678;
hash_input[1] = 0xcafebabe;
//
ixp_hash_translate(
    &hash_output,
    &hash_input,
    1,
    &hash_done,
    sig_mask,
    HW_HASH_64);
```

4.4 thread

The `thread` interface consists of utilities related to microengine threads.

Table 4-4. `thread` API

Name	Description
`thread_id()`	Given an *UENG_ID*, returns the thread ID of a context.
`thread_init()`	Initializes the number of contexts configured for the current microengine and the previous microengine.
`thread_next()`	Given an *UENG_ID*, returns the thread ID of the current context plus one.
`thread_prev()`	Given an *UENG_ID*, returns the thread ID of the current context minus one.

4.4.1 API Functions

4.4.1.1 `thread_id()`

Given a microengine identifier—`UENG_ID`—this function returns the thread ID for the active context.

Microengine Assembler Syntax

```
thread_id(out_tid, UENG_ID);
```

Microengine C Syntax

```
INLINE uint32_t ixp_thread_id(uint32_t UENG_ID);
```

Input

`UENG_ID`	The global chip microengine identifier—an integer between zero and the maximum number of microengines. The maximum number is dependent on the chip version. • For IXP2400 Network Processor the range is zero through three—that is, `0x10-0x13` • For IXP2800 and IXP2850 the range is zero through seven—that is, `0x10-0x17`

Output/Returns

`out_tid`	Global chip microengine thread identifier.

Estimated Size

Three instructions.

Microengine Assembler Example Usage

```
.local my_tid, next_tid, prev_tid, me_id
immed[me_id, 0]
thread_init (4, 0);
thread_id(my_tid, UENGINE_ID);
thread_next(next_tid, 0, 0, 30);
thread_prev(prev_tid, me_id, 0, 15);
```

Microengine C Example Usage

```
my_tid = ixp_thread_id(__ME());
// if ME id was 1 and,
// if context was 3, my_tid is 10
```

4.4.1.2 `thread_init()`

Initializes the number of contexts configured for the current and previous microengines.

Note: This operation is *not* supported in Microengine C since `thread_next()` and `thread_prev()` functions have input arguments for `curr_context_mode` and the `prev_ctx_mode`.

Microengine Assembler Syntax

```
thread_init(in_curr_num_contexts, in_prev_num_contexts);
```

Microengine C Syntax

Not supported.

Input

`in_curr_num_contexts`	The number of contexts configured for the current microengine. The value is either eight or four.
`in_prev_num_contexts`	The number of contexts configured for the previous microengine. The value is either eight or four, or, if there is no previous microengine—such as when the current microengine is microengine0—or when the previous microengine is disabled, the value is zero.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
thread_init(8, 0);
```

4.4.1.3 `thread_next()`

Given the microengine identifier, `UENG_ID`, returns one plus the thread ID for the active context.

Microengine Assembler Syntax

```
thread_next(out_tid, UENG_ID, in_min_thread, in_max_thread);
```

Microengine C Syntax

```
INLINE uint32_t ixp_thread_next(
    uint32_t UENG_ID,
    uint32_t in_min_thread,
    uint32_t in_max_thread,
    uint32_t in_curr_ctx_mode);
```

Input

`UENG_ID`	The global chip microengine identifier, an integer between zero and `n` inclusive, where `n` is chip dependent. The maximum values for `n` are: • For the IXP1200 the maximum is five • For the IXP 2400 Network Processor the maximum is three • For the IXP 2800 the maximum is seven
`in_min_thread`	The first thread of a sequence of threads.
`in_max_thread`	The last thread of a sequence of threads.
`in_curr_ctx_mode`	The context mode of the current microengine.

The range formed by `in_min_thread` and `in_max_thread` must conform to the following:

- `in_min_thread <= current_thread_id <= in_max_thread`
- `in_min_thread` through `in_max_thread` must be thread contiguous. This means that if a microengine is disabled, the range cannot cover that microengine's threads
- If a microengine is configured with four contexts and if `in_min_thread` or `in_max_thread` is in that microengine, the value of `in_min_thread` and `in_max_thread` must be an even number—for example, for ME0 this value is one of {`0`, `2`, `4`, or `6`}

Output/Returns

`out_tid`	The global chip microengine thread identifier.

Estimated Size

Six to seven instructions.

Microengine Assembler Example Usage

```
.reg my_tid, next_tid, prev_tid, me_id
    immed[me_id, 0]
    thread_init (4, 0)
thread_id(my_tid, UENGINE_ID);
thread_next(next_tid, 0, 0, 30);
thread_prev(prev_tid, me_id, 0, 15);
```

Microengine C Example Usage

```
#define UENGINE_ID 1
uint32_t min_thd = 0, max_thd = 7, curr_ctx_mode = 8;
uint32_t next_tid;
next_tid = ixp_thread_next(UENGINE_ID, min_thd, max_thd, curr_ctx_mode);
    // if context was 7, next_tid will be 0, current context mode is 8
```

4.4.1.4 `thread_prev()`

Given the microengine identifier, `UENG_ID`, returns the thread ID for the active context minus one. This is the thread identifier for the previously active thread.

Microengine Assembler Syntax

```
thread_prev(out_tid, UENG_ID, in_min_thread, in_max_thread);
```

Microengine C Syntax

```
INLINE uint32_t ixp_thread_prev(
    uint32_t UENG_ID,
    uint32_t in_min_thread,
    uint32_t in_max_thread,
    uint32_t in_curr_ctx_mode,
    uint32_t in_prev_ctx_mode);
```

Input

`UENG_ID`	The global chip microengine identifier, an integer between zero and `n` inclusive, where `n` is chip dependent. The maximum values for `n` are: • For the IXP 2400 Network Processor the maximum is three • For the IXP 2800 the maximum is seven
`in_min_thread`	The first thread of a sequence of threads.
`in_max_thread`	The last thread of a sequence of threads.
`in_curr_ctx_mode`	The context mode of the current microengine.
`in_prev_ctx_mode`	The context mode of the previous microengine.

The range formed by `in_min_thread` and `in_max_thread` must conform to the following:

- `in_min_thread <= current_thread_id <= in_max_thread`
- `in_min_thread` through `in_max_thread` must be thread contiguous. This means that if a microengine is disabled, the range cannot cover that microengine's threads
- If a microengine is configured with four contexts and if `in_min_thread` or `in_max_thread` is in that microengine, the value of `in_min_thread` and `in_max_thread` must be an even number—for example, for ME0 the value is one of {`0`, `2`, `4`, or `6`}

Output/Returns

`out_tid`	The global chip microengine thread identifier of the active context minus one.

Estimated Size

Seven to nine instructions.

Microengine Assembler Example Usage

```
.local my_tid, next_tid, prev_tid, me_id
    immed[me_id, 0]
    thread_init (4, 0)
thread_id(my_tid, UENGINE_ID);
thread_next(next_tid, 0, 0, 30);
thread_prev(prev_tid, me_id, 0, 15);
```

Microengine C Example Usage

```
#define UENGINE_ID 0
uint32_t min_thd = 0, max_thd = 7, curr_ctx_mode = 8, prev_ctx_mode = 4;
uint32_t next_tid;
prev_tid = ixp_thread_prev(UENGINE_ID, min_thd, max_thd,
     curr_ctx_mode, prev_ctx_mode);
// if context was 0, prev_tid will be 7
// current ME context mode is 8, previous ME context mode is 4
```

4.5 xbuf

The XBUF macros are a set of utility macros that manipulate a block of microengine transfer registers as an array—the transfer registers can be accessed through indexes.

The XBUF macros considerably simplify the code for processing packet and cell headers. By allowing the header to be treated as a block of data, and then making use of a set of offsets into the data block, specific fields within the header can be more easily extracted. The XBUF macros provide byte-addressable access into this virtual array of registers.

The `xbuf_extract()` and `xbuf_insert()` macros can be used to provide packet parsing functionality for IP and Ethernet packets.

The `xbuf_alloc` and `xbuf_free` macros are used to allocate and free blocks of registers. They generate the appropriate `.xfer_order` and `.local` assembler directives so that the user is freed from that task. While the `xbuf_alloc` macro is most often used to allocate blocks of registers, it is also frequently used to allocate just one register, thereby providing a `standard` means of allocating and freeing register resources—that is, `xbuf_alloc` is used as an alias for `.local`.

The XBUF macros are actually a set of assembler preprocessor macros. They generate the appropriate `.reg`, `.xfer_order`, `.xfer_order_read`, `.xfer_order_write`, and other assembler statements to provide the naming conventions that allow byte addressable buffer names to be used. For example, consider the following statement:

```
xbuf_alloc ($packet_buf, 8, read_write);
```

Once this statement is used, an SRAM buffer of eight 32-bit longwords is allocated, and any register within the block can be accessed. If an application required access to the fourth longword the expression `$packet_buf[3]` would provide this access.

The XBUF macros provide simple error checking at the assembler preprocessor level. For example, XBUF macros prevent an SRAM buffer from being linked to an SDRAM buffer.

The XBUF macros can be used with GPRs (general-purpose registers), local memory registers, as well as SRAM or SDRAM transfer registers. As is the case with direct access, care must be taken when using transfer registers as the same name is used for the read and write transfer registers.

The `xbuf_link` macro can be used to chain two previously defined buffers together so that they can be used as one larger, contiguous buffer. Furthermore, `xbuf_link` can be used to implement circular buffers, where the last register of the last buffer can be linked to the beginning of the first buffer. For example:

```
xbuf_alloc ($first_buf, 4, read_write);
xbuf_alloc ($second_buf, 2, read_write);
xbuf_link ($first_buf, $second_buf);
xbuf_link ($second_buf, $first_buf);
```

The XBUF macros make use of the `BIG_ENDIAN` compiler define. Note that `xbuf_link` should be used only to extract data across multiple linked buffers and should not be used for read operations across the linked buffers.

Table 4-5. `xbuf` API

Name	Description
`xbuf_activate()`	Maps the specified `ACTIVE_LM_ADDR` control and status register to the specified buffer.
`xbuf_alloc()`	Allocates a contiguous transfer register buffer consisting of `NUMBER_OF_REGS` transfer registers.
`xbuf_bind_address()`	Binds a buffer name to an address identifying a buffer within a pool of buffers inside local memory.
`xbuf_deactivate()`	Unbinds the specified buffer from its current local memory handle.
`xbuf_extract()`	Extracts a numeric byte field from a transfer register buffer.
`xbuf_extract_frm_linked_bufs()`	Extracts a numeric byte field from a register buffer.
`xbuf_find()`	Given a transfer register name, removes special characters and array notation and checks if the XBUF token name is an allocated XBUF. Make this XBUF the current XBUF.
`xbuf_free()`	Frees a transfer register buffer.
`xbuf_link()`	Links two transfer register buffers.
`xbuf_param_set()`	Defines the tokens `_cur_xbuf_name`, `_cur_xbuf_size`, and `_cur_xbuf_next`.
`xbuf_type_extract()`	Extracts a value from a transfer register buffer based on a datatype.
`xbuf_xfer_set()`	Defines the tokens `_XFER0` through `_XFER7`.
`xbuf_copy()`	Copies the source XBUF from a specified byte offset into destination XBUF.
`xbuf_insert()`	Inserts the specified bytes of a numeric byte field into an XBUF buffer at the specified start byte offset.

4.5.1 API Functions

4.5.1.1 `xbuf_alloc()`

Allocates a contiguous transfer register buffer consisting of `NUMBER_OF_REGS` transfer registers. This is done with compile-time defines only—no instructions are used.

The maximum number of transfer buffers that can be allocated is four.

The maximum number of local memory buffers that can be allocated is 16.

The maximum number of registers in a transfer buffer is sixteen for the Intel® IXP2400, IXP2800 and IXP2850 Network Processors.

Microengine Assembler Syntax

```
xbuf_alloc (in_xbuf_name, NUMBER_OF_REGS, READ_WRITE_INDICATOR);
```

Input

`in_xbuf_name`	The token name for the transfer buffer. Required naming convention: • DRAM transfer registers buffer: $$*name* • SRAM transfer registers buffer: $*name* • GPR buffer: *name* Where *name* is any string. • local memory: the internal name is `lmem_buf`*x*—where *x* is zero through 16 To support more descriptive names, use the following code: `#define_eval any_namelmem_buf`*x*—where *x* is zero through 16 . For example: `#define_eval ipv4header lmem_buf2`
`NUMBER_OF_REGS`	The constant number of transfer registers to assign.
`READ_WRITE_INDICATOR`	Specifies the intended use of the buffers. The value should be one of {`read`, `write`, or `read_write`}.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
xbuf_alloc(ipv4_hdr, 8, READ_WRITE);
```

4.5.1.2 xbuf_activate()

Maps the specified ACTIVE_LM_ADDR control and status register to the specified buffer. The absolute address of the buffer is put in the ACTIVE_LM_ADDR local control and status register. The absolute address is calculated from the parameters supplied by the xbuf_bind_address() call. After this macro is called, the specified buffer can be accessed via *l$index0 and *l$index1. To be certain the control and status register is properly set for the correct local memory index, the calling application must make this call every time it accesses the buffer.

Note that there are only two active buffers at a time for the two local memory handles. When xbuf_activate() is called, it searches for the current active buffer of the same local memory handle and removes the binding between that buffer and the local memory handle. It then sets up the new binding for the specified buffer.

The reasons for this design are:

- It is compatible with the hardware design—new
  local_csr_wr[ACTIVE_LM_ADDR<x>, addr] overrides the previous address
- Allow local memory buffer link requirement—only two local memory buffers of opposite handles are allowed to be linked—this is easily checked
- Allow comparisons to work properly with assembler's pre-processing

Microengine Assembler Syntax

```
xbuf_activate(name, INDEX, thread_id, WAIT_ACTION)
```

Input

name	The name of the buffer. The name must start with lmem_buf. For example, lmem_buf1. If the name doesn't start with lmem_buf, the macro assumes that buffer is not in local memory and nothing is done.
INDEX	Selects the index to use—the value is one of the following: • 0—to map *l$index0 to the specified buffer • 1—to map *l$index1 to the specified buffer
thread_id	The run-time value for thread ID. This value can be a general-purpose register or a constant.
WAIT_ACTION	Selects the wait behavior after setting up the local Control and Status Register—the value is one of the following: • 1—wait three cycles after setting up local the local Control and Status Register—equivalent to executing three NOP instructions • 0—do not wait—equivalent to executing zero NOP instructions

Output

Sets ACTIVE_LM_ADDR_0 or ACTIVE_LM_ADDR_1. Sets _xbuf[i]_lmem_index equal to INDEX.

Estimated Size

- For `WAIT_ACTION` equals zero—three to four instructions
- For `WAIT_ACTION` equals one—six to seven instructions

Microengine Assembler Example Usage

```
#define_eval ipv4_hdr lmem_buf0
xbuf_alloc(ipv4_hdr, 8, READ_WRITE)
xbuf_bind_address(ipv4_hdr, POOLS_BASE, POOL_SIZE, BUF_OFF)
xbuf_activate(ipv4_hdr, 0, 3, 1)
                         // activate ipv4_hdr for thread3, LMEM handle0
                         // now ipv4_hdr[0], ipv4_hdr[1], …, ipv4_hdr[7]
                         // are associated with
                         // *l$index0[0], *l$index0[1], …, *l$index0[7].
```

Applications that run with a context mode of four must explicitly define the following:

```
#define CONTEXT_MODE_4
```

Note: A context mode of eight is set by default.

The buffer absolute address is:

```
(thread ID << log_base2_of(_xbuf[i]_lmem_pool_size) ) +
       _xbuf[i]_lmem_pools_base + _xbuf[i]_lmem_offset
```

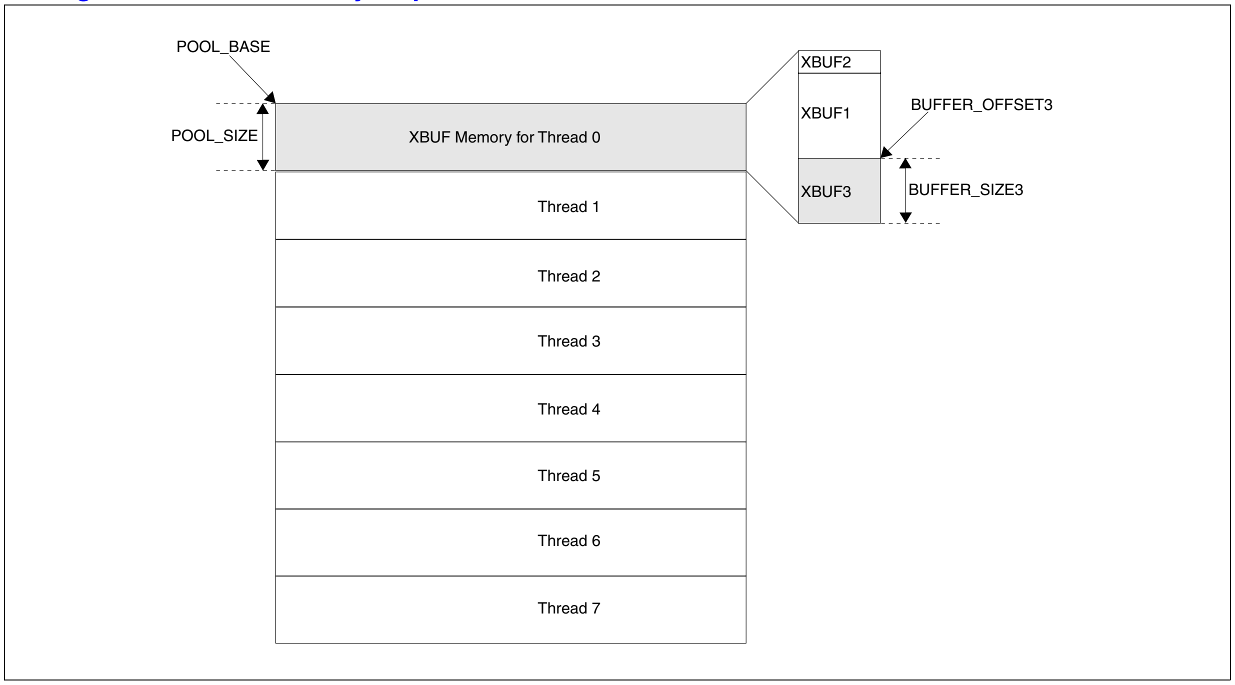
Where `_xbuf[i]_lmem` is the buffer with the specified name.

Note: `xbuf_bind_address()`, `xbuf_activate()`, and `xbuf_deactivate()` are applicable to local memory buffers. These buffers are transparent—no additional cycles are spent relative to other buffer types.

4.5.1.3 xbuf_bind_address()

Binds a buffer name to an address identifying a buffer within a pool of buffers inside local memory. Common designs, for example the memory map shown in Figure 4-3, have local memory divided into several blocks of pools. In each block, the pools are contiguous, and the number of pools is equal to the number of active threads so that each thread has its own pool. Inside a pool, there can be several buffers.

Figure 4-3. xbuf Memory Map



Microengine Assembler Syntax

```
xbuf_bind_address(name, POOLS_BASE, POOL_SIZE, BUFFER_OFFSET)
```

Input

`name`	The buffer name. The name has to start with `lmem_buf`—for example, `lmem_buf1`. If `name` doesn't start with `lmem_buf`, the macro assumes that the buffer is not in local memory and nothing is done.

Input

`POOLS_BASE`	The base of all pools in bytes—a constant multiple of `POOL_SIZE`.
`POOL_SIZE`	The size of each pool in bytes—must be a constant power of two.
`BUFFER_OFFSET`	The offset in bytes of the buffer inside its pool—a constant. The offset should be aligned on a longword boundary which is calculated as the next higher power of two of the buffer size.

For example:

`xbuf_alloc(lmem_buf0, 2, read_write)`
`buffer_offset` must be aligned on an 8 byte boundary.
`xbuf_alloc(lmem_buf0, 3, read_write)`
`buffer_offset` must be aligned on a 16 byte boundary.
`xbuf_alloc(lmem_buf0, 4, read_write)`
`buffer_offset` must be aligned on a 16 byte boundary.
`xbuf_alloc(lmem_buf0, 5, read_write)`
`buffer_offset` must be aligned on a 32 byte boundary.

to

`xbuf_alloc(lmem_buf0, 8, read_write)`
`buffer_offset` must be aligned on a 32 byte boundary.
`xbuf_alloc(lmem_buf0, 9, read_write)`
`buffer_offset` must be aligned on a 64 byte boundary.

to

`xbuf_alloc(lmem_buf0, 16, read_write)`
`buffer_offset` must be aligned on a 64 byte boundary.

Note: The operations `xbuf_bind_address()`, `xbuf_activate()`, and `xbuf_deactivate()` are used with local memory buffers. Use of these buffers is transparent—no additional cycles are needed relative to other buffer types.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
#define_eval ipv4_hdrlmem_buf2
immed[thread_id, 2]
xbuf_alloc(ipv4_hdr, 8, READ_WRITE)
xbuf_bind_address(ipv4_hdr, 0x100, 0x100, 0)
xbuf_activate(ipv4_hdr, 1, thread_id, 1)
```

4.5.1.4 xbuf_deactivate()

Unbinds the specified buffer from its current local memory handle. This macro undoes the work of xbuf_activate(). This macro is called at the end of a block that defines the scope of the binding. Users have to call xbuf_activate() again to associate the buffer name with either local memory handle zero or local memory handle one before the buffer can be used.

Microengine Assembler Syntax

```
xbuf_deactivate(name)
```

Input

name	The buffer name. The name must start with *lmem_buf*—as for example, lmem_buf1. If the name doesn't start with *lmem_buf*, the macro assumes the buffer is not in local memory and nothing is done.

Output

The buffer name is no longer associated with a local memory handle.

Estimated Size

Zero instructions.

Note: xbuf_bind_address(), xbuf_activate(), and xbuf_deactivate() are applicable to local memory buffers. The use of these buffers is transparent—no additional cycles are consumed relative to other buffer types.

Microengine Assembler Example Usage One

```
#define_eval ipv4_hdrlmem_buf0
xbuf_alloc(ipv4_hdr, 8, read_write)
xbuf_bind_address(ipv4_hdr, pools_base, pool_size, buf_offset)
xbuf_activate(ipv4_hdr, 0, 3, 1)
       // uses of ipv4_hdr
xbuf_deactivate(ipv4_hdr)
xbuf_activate(ipv4_hdr, 1, 3, 1)
       // uses of ipv4_hdr
xbuf_deactivate(ipv4_hdr)
xbuf_free(ipv4_hdr)
```

Microengine Assembler Example Usage Two

```
#define_eval ipv4_hdrlmem_buf1
xbuf_alloc(ipv4_hdr, 8, read_write)
xbuf_bind_address(ipv4_hdr, pools_base, pool_size, buf_offset)
#macro first_block(name)
xbuf_activate(name, 1, 2, 1)// select handle1
xbuf_extract(..., name,...)
xbuf_deactivate(name)
#endm
#macro second_block(name)
xbuf_activate(name, 0, 2, 1)// switch to handle0
xbuf_extract(..., name, ...)
xbuf_deactivate(name)
#endm
first_block(ipv4_hdr)
second_block(ipv4_hdr)
```

4.5.1.5 `xbuf_extract()`

Extracts a numeric byte field from a transfer register buffer.

Microengine Assembler Syntax

```
xbuf_extract (out_byte_field, in_xbuf_name, \
   window_start, field_start, number_of_bytes);
```

Input

`in_xbuf_name`	The token name for the transfer buffer.
`window_start`	The starting byte position of the source buffer. This value is a constant or a general-purpose register.
`field_start`	The starting byte offset of the field from the beginning of the window of the source buffer. This value is a constant or a general-purpose register.
`number_of_bytes`	The number of bytes to extract—this value must be less than or equal to four. The value is a constant or a general-purpose register.

Output

`out_byte_field`	The extracted byte field, right-justified.

Estimated Size

- For constant offsets and size—1 to 3 instructions
- For run-time offsets and constant size:
  - For transfer register buffers—10 instructions
  - For local memory buffers—15 instructions
  - For general-purpose register buffers—the number of instructions varies depending on where the data located in the buffer
- For run-time offsets and run-time size:
  - For transfer register buffers—11 instructions
  - For local memory buffers—16 instructions
  - For general-purpose register buffers—the number of instructions varies depending on where the data located in the buffer

Microengine C Example Usage

```
.begin
    .reg base out_byte_field
    .set out_byte_field

    xbuf_alloc($wbuf, 8, write);
    xbuf_alloc($rbuf, 8, read);
    immed32[$wbuf[0], 0x01020304]

    #define_eval TEST_VAL 0x11111111
    #define_eval INDEX 1
    #while (INDEX < 8)
        move($wbuf[/**/INDEX], TEST_VAL)
        #define_eval TEST_VAL (TEST_VAL * 2)
        #define_eval INDEX (INDEX + 1)
    #endloop
    move(base, 0x200)
        //write buffer into sram
    sram_write($wbuf[0], base, 0, 8, sig1, sig1, ___);
    sram_read($rbuf[0], base, 0, 8, sig1, sig1, ___);
/////////////////////////////////////////////////////////////////////////////
    #define_eval SIZE 2
        // extract two bytes from 0 offset
    xbuf_extract(out_byte_field, $rbuf, 0, 1, SIZE)
.end
```

4.5.1.6 xbuf_extract_frm_linked_bufs()

Extracts a numeric byte field from a register buffer.

The source of data may spread across two buffers: `str_xbuf_name` and the buffer linked to it. The `DATA_SPREAD` parameter indicates whether or not data spreads across two buffers. If the `DATA_SPREAD` parameter is set to zero this macro is equivalent to `xbuf_extract()`.

Microengine Assembler Syntax

```
xbuf_extract_frm_linked_bufs(out_byte_field, str_xbuf_name, \
    window_start, field_start, number_of_bytes, DATA_SPREAD)
```

Input

`in_xbuf_name`	The token name for the transfer buffer.
`window_start`	The start byte position of the source buffer—a constant or general-purpose register.
`field_start`	The start byte offset of the field from the window start of the source buffer—a constant or general-purpose register.
`Number_of_bytes`	The number of bytes to extract—a constant or general-purpose register. This value must be less than or equal to four.
`DATA_SPREAD`	A constant indicating whether or not source data spreads across two buffers. This value is one of: • One—part of the data to be extracted spreads into the buffer linked to `in_src_xbuf` • Zero—all data to be extracted resides in `str_xbuf_name`

Output/Returns

`out_byte_field`	The right-justified extracted byte field.

Estimated Size

- For `DATA_SPREAD` equal to 0—same size as `xbuf_extract()`
- For `DATA_SPREAD` equal to 1 and with constant offsets and size—1 to 3 instructions
- For `DATA_SPREAD` equal to 1 and with run-time offsets and constant size:
  - 16 instructions for transfer registers buffers
  - 15 instructions for local memory buffers
  - Depending on where the data located in the buffer, the size varies for GPR buffers
- For `DATA_SPREAD` equal to one and with run-time offsets and run-time size:
  - 21 instructions for transfer registers buffers
  - 19 instructions for local memory buffers
  - Depending on where the data located in the buffer, the size varies for GPR buffer

4.5.1.7 xbuf_find()

Given a transfer register name, removes special characters and array notation then checks if the XBUF token name is an allocated XBUF. Makes this XBUF the current XBUF using `xbuf_param_set()`. This is done with compile-time defines only—no instructions are used.

Microengine Assembler Syntax

```
xbuf_find (in_xfer_name);
```

Input

`in_xfer_name`	The application variable used as a transfer register. Array notation may be used, such as `$packet_buf[2]`.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
.begin
    .reg x_reg, case_num, expected
    #define_eval TEST_CASE_NUM XBUF_TEST_NUM+25
    immed32(case_num, TEST_CASE_NUM);
    xbuf_alloc(x_reg, 10, read_write);
    xbuf_find(x_reg[2]);

    #ifdef _xbuf_found
        // XBUF token name x_reg[2] is an allocated XBUF
    #else
        // XBUF token name x_reg[2] is not an allocated XBUF
    #endif
    xbuf_free(x_reg);
.end
```

4.5.1.8 `xbuf_free()`

Frees a contiguous transfer register buffer. Disconnects any links with other transfer register buffers.

This is done with compile-time defines only—no instructions are used.

Microengine Assembler Syntax

```
xbuf_free (in_xbuf_name);
```

Input

`in_xbuf_name`	The token name for the transfer buffer.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
xbuf_free(x_reg);
```

4.5.1.9 `xbuf_link()`

Links a transfer register buffer to a next transfer register buffer. This is done with compile-time defines only—no instructions are used.

Microengine Assembler Syntax

```
xbuf_link (in_xbuf_name, str_nextxbuf_name);
```

Input/Output

`in_xbuf_name`	The token name for the transfer buffer.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
.begin
    .reg $x, $$x, $y, $$y
    xbuf_alloc($x, 4, read_write);
    xbuf_alloc($y, 4, read_write);
    // Link a $x to a $y
    xbuf_link($x, $y)
    //do some processing on XBUF $x & $y
    xbuf_free($x);
    xbuf_free($y);
.end
```

4.5.1.10 `xbuf_param_set()`

Makes this XBUF the current XBUF. Defines tokens `_cur_xbuf_name`, `_cur_xbuf_size`, and `_cur_xbuf_next`. This is done with compile-time defines only—no instructions are used.

Microengine Assembler Syntax

```
xbuf_param_set (in_xbuf_name);
```

Input

`in_xbuf_name`	The token name for a transfer buffer or GPR buffer.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
.begin
    .reg case_num, a_reg, b_reg
//
    xbuf_alloc(a_reg, 2, read_write);
//
    immed32(a_reg[0], 0x01000CCC);
    immed32(a_reg[1], 0xCCCC0030);
//
    xbuf_param_set(a_reg);
    xbuf_xfer_set(_BUF0, a_reg, 0);
    // now BUF0_REG0 is token name for a_reg[0] & BUF0_REG1 token name for
    // a_reg[1]
//
    xbuf_free(a_reg)
.end
```

4.5.1.11 xbuf_type_extract()

Extracts a numeric byte field from a transfer register buffer. The bytes of the field are swapped—that is, the byte-order is reversed—if DATATYPE is set to LITTLE_ENDIAN_BYTES.

Microengine Assembler Syntax

```
xbuf_type_extract (out_byte_field, in_xbuf_name, \
    WINDOW_START, FIELD_START, NUMBER_OF_BYTES, DATATYPE);
```

Input

in_xbuf_name	The token name for the transfer buffer.
WINDOW_START	The starting location of a buffer window. The location is relative to the start of the transfer buffer and is a constant value.
FIELD_START	The start byte within a buffer window. This is a constant value.
NUMBER_OF_BYTES	The number of sequential bytes to extract. The minimum number of bytes to extract is one and the maximum number is four. This is a constant value.
DATATYPE	The type and order of data whose value is one of {BIG_ENDIAN_BYTES, LITTLE_ENDIAN_BYTES}. This is a constant value.

Output

out_byte_field	The extracted byte field, right-justified.

Estimated Size

One to three instructions.

Microengine Assembler Example Usage

```
xbuf_alloc(wbuf, 8, read_write);
//
    #define_eval WIN_START 0
    #define_eval FIELD_START 0
    #define_eval NUM_OF_BYTES 1
//
    immed32[wbuf[0], 0x01020304]
    xbuf_type_extract(out_byte_field, wbuf, WIN_START, FIELD_START, \
        NUM_OF_BYTES, BIG_ENDIAN_BYTES)
//out_byte_field should be equal to 1
```

4.5.1.12 `xbuf_xfer_set()`

Traverses `xbufs`, following links, until `BYTE_OFFSET` is reached. Starting with that transfer register, defines tokens for a sequence of up to eight transfer registers `_XFER0` through `_XFER7`. These tokens may then be used in macros for extracting byte fields or for other operations. This is done with compile-time defines only—no instructions are used.

Microengine Assembler Syntax

```
xbuf_xfer_set(in_xbuf_name, BYTE_OFFSET);
```

Input

`in_xbuf_name`	The token name for the transfer buffer.
`BYTE_OFFSET`	The constant byte location from starting XBUF to the start of the transfer register.

Estimated Size

Zero instructions.

Microengine Assembler Example Usage

```
xbuf_alloc(a_reg, 4, read_write);
xbuf_alloc(b_reg, 4, read_write);
//
immed32(a_reg[0], 0x01000CCC);
immed32(a_reg[1], 0xCCCC0030);
immed32(a_reg[2], 0x788D43B7);
immed32(a_reg[3], 0x0032AAAA);
xbuf_param_set(a_reg);
xbuf_xfer_set(_BUF0, a_reg, 0);
//_BUF0_REG0 should be equal to 0x01000CCC
//_BUF0_REG1 should be equal to 0xCCCC0030
//_BUF0_REG2 should be equal to 0x788D43B7
//_BUF0_REG3 should be equal to 0x0032AAAA
```

4.5.1.13 `xbuf_copy()`

Copies the source XBUF from a specified byte offset into the destination XBUF. This is the generic copy macro for all combinations of alignments of start locations.

Microengine Assembler Syntax

```
Xbuf_copy(out_dest_buf, out_last_element, dest_start_byte, in_src_buf,\
src_start_byte, in_prepend, total_bytes_to_copy, DATA_SPREAD)
```

Input

`dest_start_byte`	The absolute offset in bytes from the beginning of the output buffer to the location where copied data resides. This value can be longword aligned or not—that is, `dest_start_byte % 4` can equal zero, one, two, or three. This value can be a constant or general-purpose register.
`in_src_buf`	The input buffer name. This value can be in an SRAM read register, a DRAM read register, local memory, or a general-purpose register. For `xbuf_copy()` operation with constant offsets and size—that is, where `dest_start_byte`, `src_start_byte`, `total_bytes_to_copy` are all constants—a `DATA_SPREAD` value of zero or one is allowed. In other words, the data to be copied can completely reside in `in_src_xbuf`, or it can spread between `in_src_buf` and the buffer that is linked to `in_src_buf`. In all other cases, `xbuf_copy()` does not support linked buffers. In other words, `xbuf_copy()` with run-time parameters does not support the case where data spreads beyond `in_src_xbuf` into other buffers linked to it.
`src_start_byte`	The absolute offset in bytes from the beginning of the input buffer. This value can be longword aligned or not—that is, `src_start_byte % 4` can equal zero, one, two, or three. This value can be a constant or general-purpose register.

Input (Continued)

`in_prepend`	A general-purpose register or a constant containing bytes to be merged with the first word in the destination—prepended if byte alignment of the destination is not zero. The bytes to be merged must be at the exact byte locations that they are to occupy in the first word of the destination. All other bytes must be zero. When `in_prepend` is not needed the calling application should pass the constant zero.
`total_bytes_to_copy`	The total number of bytes to copy. This value can be a constant or a general-purpose register. The maximum number of bytes is dependent on the maximum size of the output buffer. Currently the maximum size is 64 bytes—16 registers—for buffers in transfer registers, general-purpose registers, and local memory registers.
`DATA_SPREAD`	A constant flag to signal whether the data to copy spreads into the buffer that is linked with `in_src_buf`. • `1`—part or all of the data to copy resides in the buffer linked to `in_src_xbuf` This option is only available for `xbuf_copy()` with constant offsets and size. • `0`—all source data resides in `in_src_xbuf` `DATA_SPREAD` equal to zero is *required* for `xbuf_copy()` with run-time offsets or run-time size.

Output

`out_dest_buf`	The output buffer name. This can be an SRAM write register, a DRAM write register, local memory, or a general-purpose register. The maximum number of bytes that `xbuf_copy` can fill is equal to the size of `out_dest_xbuf`—this is the case when `dest_start_byte` is zero. In other words, if `out_dest_xbuf` is linked to another buffer, users can't specify a `total_bytes_to_copy` bigger than the size of `out_dest_xbuf` and expect that the extra bytes go into the linked buffer.
`out_last_element`	A general-purpose register used to contain the last element of the destination buffer. This output is useful when the destination buffer elements are in a write transfer register and the copy operation results in an incomplete last element—not all four bytes are filled. The need for `out_last_element` arises when the calling-application copies another buffer into a partially filled destination buffer. The last longword element in the destination may be incomplete as the result of the previous copy. In this case, the calling application must pass the `out_last_element` of the previous copy as the `in_prepend` for the subsequent `xbuf_copy()` operation. Otherwise, the bytes in the partially filled write register element are cleared. **NOTE:** If the calling-application is sure that this value is not needed, the application can pass in the constant zero to save one instruction.

Estimated Size

When the source and the destination are the same XBUF of the type general-purpose register or local memory and when the start bytes are the same the estimated size is zero instructions.

When source and destination are not identical—that is, not the same buffer and the same start byte:

- With constant offsets and size:
  - — Best case—`n` plus two instructions where `n` is the number of longwords copied
  - — Worst case—`n` plus 99 instructions where `n` is the number of longwords copied
- With a run-time destination offset—size varies depending on whether or not the destination is 4 byte aligned:
  - — Best case—approximately 2 instructions times the number of 32 byte words, then add 3 instructions
  - — Worst case—approximately 2 instructions times the number of 32 byte words, then add 28 instructions
- With a run-time source offset:
  - — Best case—approximately 2 instructions times the number of 32 byte words, then add 3 instructions
  - — Worst case—approximately 4 instructions times the number of bytes, then add 1 instruction—the worst case is when the number of bytes is less than 1 longword
- With a run-time destination offset and run-time size:
  - — Best case—21 instructions—when copying exactly 4 bytes and when both source and destination offsets are aligned
  - — Worst case—when doing unaligned copy to and from local memory registers because `byte_align_be` doesn't work with index local memory registers

    This is especially bad when `total_number_of_bytes` is small. In this case, the size can be as high as 8 instructions times the number of bytes. For number of bytes greater than 16, the size is better as the number of bytes gets closer to the maximum value of 64—approximately 4 times the number of bytes.
- Run-time source offset and run-time size:
  - — Best case—1.6 instructions times the number of bytes—when the copy is of bytes from an aligned source close to an aligned destination
  - — Worst case—when doing unaligned copy to and from local memory registers because `byte_align_be` does not work with index local memory registers

    This is especially bad when `total_number_of_bytes` is small. In this case, the size can be as high as eight instructions times the number of bytes. For number of bytes greater than 16, the size is better as the number of bytes gets closer to the maximum value of 64—approximately two times the number of bytes.

Microengine Assembler Example Usage

```
xbuf_alloc($$xbuf_dest, 4, read_write);
xbuf_alloc($$xbuf_dest, 4, read_write);
// ...
// copy 8 bytes starting w/ byte offset 14 from src w/ 2 byte offset into dest
xbuf_copy($xbuf_dest, 2, $xbuf_src, 14, 8)
```

4.5.1.14 `xbuf_insert()`

Insert the specified bytes of a numeric byte field into an XBUF buffer at the specified start byte offset.

Note: This interface is structured in a way analogous to the `xbuf_extract()` interface.

Microengine Assembler Syntax

```
xbuf_insert(io_str_xbuf_name, in_byte_field, window_start, field_start, \
    number_of_bytes);
```

Input

`in_byte_field`	The byte field to be inserted. **NOTE:** This field is right-justified with respect to the number of bytes—that is, if the number of bytes specified is two then the right-most two bytes, the least significant bytes, are inserted.
`window_start`	The start byte position of the source buffer. This value can be a constant or a general-purpose register.
`field_start`	The start byte offset of the field from the window start of source buffer. This value can be a constant or a general-purpose register.
`number_of_bytes`	The number of bytes to insert—this value must be less than or equal to four. This value can be a constant or a general-purpose register.

Input/Output

`io_str_xbuf_name`	The name of the XBUF register where the byte field is to be inserted.

Estimated Size

When all parameters are constants:

- Best case—1 instruction
- Worst case—4 instructions

When `window_start` and `field_start` are constants and size is run-time:

- Best case—2 instructions
- Worst case—8 instructions

When `window_start` or `field_start` are run-time and size is constant:

- Best case—11 instructions
- Worst case
  - 15 instructions—for transfer register buffers
  - 24 instructions—for local memory buffers

When `window_start` or `field_start` are run-time and size is run-time:

- Best case—11 instructions
- Worst case
  - 24 instructions—for transfer register buffers
  - 30 instructions—for local memory buffers

Microengine Assembler Example Usage One

```
xbuf_alloc($$io_xbuf_dest, 8, read_write);
//...
move(in_byte_field, 0x12345678);
    //Insert in_byte_field into io_xbuf_dest starting at byte offset 12
xbuf_insert($io_xbuf_dest, in_byte_field, 0, 12, 4)
```

Microengine Assembler Example Usage Two

```
xbuf_alloc(my_buf, 8, read_write);
//...
move(in_byte_field, 0x1);
    // Insert 1 byte of in_byte_field into my_buf starting at byte offset 2
    // Before my_buf = 0xaabbccdd
xbuf_insert(my_buf, in_byte_field, 0, 2, 1)
    // After xbuf_insert
    // my_buf = 0xaabb01dd
```

4.6 ixp_mem

The reloctable data structure API provides the flexibility to change memory type during compilation. The functions provide the same features as normal I/O functions except for the changing memory types feature. Any code written using this API can be moved to any memory type without modifications.

Table 4-6 summarizes the relocatable data structure API. The following assumptions and dependencies apply when using the API:

- User understands the *IXP2xxx* programming model.
- User takes care of alignment wherever needed; it is mostly needed while using DRAM.
- All options are not supported for all the memory types. An error message will indicate illegal use.
- When `sync` type `sig_done` is passed to the API, user is responsible for waiting for the signal. Failing to do so may cause unusual results.

Table 4-6. Relocatable Data Structure API

Name	Description
`ixp_mem_read()`	Reads constant number of longwords (LWs) from memory. Max number of LWs read is limited to 8 LWs. Memory types supported are *SRAM, DRAM, SCRATCH* and *LM.*
`ixp_mem_read_ind()`	Reads variable number of LWs from memory. Memory types supported are *SRAM, DRAM, SCRATCH* and *LM.*
`ixp_mem_write()`	Writes constant number of LWs to memory. Max number of LWs is limited to 8 LWs. Memory types supported are *SRAM, DRAM, SCRATCH* and *LM.*
`ixp_mem_write_ind()`	Writes variable number of LWs to memory. Memory types supported are *SRAM, DRAM, SCRATCH* and *LM.*
`ixp_mem_atomic()`	Perform different atomic operations. Memory types supported are *SRAM, SCRATCH* and *LM.*

4.6.1 Memory Types

The first parameter for all functions in the relocatable data structure API is memory type. Table 4-7 describes all the memory types in detail. The memory type supported by a given function varies. Check individual API details in Section 4.6.4, "API Functions" for supported memory types.

Table 4-7. Memory Types

MEM_TYPE	Description
MEM_DRAM	DRAM is used as memory. Register type used is DRAM transfer register.
MEM_DRAM_S	DRAM is used as memory. Register type used is SRAM transfer register.
MEM_SRAM	SRAM is used as memory. Register type used is SRAM transfer register.

Table 4-7. Memory Types

MEM_TYPE	Description
MEM_SRAM_D	SRAM is used as memory. Register type used is DRAM transfer register.
MEM_SCRATCH	SCRATCH is used as memory. Register type used is SRAM transfer register.
MEM_SCRATCH_D	SCRATCH is used as memory. Register type used is DRAM transfer register.
MEM_LM	Local Memory is used as memory. Register type used is general purpose register.

4.6.2 Operation Types

The second parameter for the API's `ixp_mem_atomic( )` function is operation type. Operation type describes the type of operation to be performed.

Signal macros described in Section 4.6.3, "Signal and Register Allocation Macros" also requires operation type as an input parameter.

Table 4-8 summarizes the supported operation types:

Table 4-8. Operation Types

OP_TYPE	Description
RW	Operation to indicate read and write API calls. Used with signal macro only.
SWAP	Swap atomic operation, used with atomic API and signal macro also.
SET	Set atomic operation, used with atomic API and signal macro also.
CLR	Clear atomic operation, used with atomic API and signal macro also.
INCR	Increment atomic operation, used with atomic API and signal macro also.
DECR	Decrement atomic operation, used with atomic API and signal macro also.
ADD	Add atomic operation, used with atomic API and signal macro also.
SUB	Subtract atomic operation, used with atomic API and signal macro also.
TEST_AND_SET	Test and set atomic operation, used with atomic API and signal macro also.
TEST_AND_CLR	Test and clear atomic operation, used with atomic API and signal macro also.
TEST_AND_INCR	Test and increment atomic operation, used with atomic API and signal macro also.

Table 4-8. Operation Types

OP_TYPE	Description
TEST_AND_DECR	Test and decrement atomic operation, used with atomic API and signal macro also.
TEST_AND_ADD	Test and add atomic operation, used with atomic API and signal macro also.
TEST_AND_SUB	Test and subtract atomic operation, used with atomic API and signal macro also.

4.6.3 Signal and Register Allocation Macros

Different types of signals and registers are needed when different memory types are passed to the relocatable data structure API functions.

To make sure that proper signal and register is allocated, you need to use macros described in Table 4-9 with the memory type as the input.

Table 4-9. Signal and Register Allocation Macros

Macro	Description
SIG_COMB( )	Takes memory type and operation type as input and declares proper signal.
DS_READ_REG( )	Takes memory type as input and declares proper read register type.
DS_WRITE_REG( )	Takes memory type as input and declares proper write register type.
DS_READ_WRITE_REG( )	Takes memory type as input and declares proper read/write register type.

4.6.4 API Functions

This section provides details on the functions of the relocatable data structure API.

4.6.4.1 `ixp_mem_read()`

This function reads count longwords from the specified memory type at the specified address and places the data into the structure addressed by `rd_reg`. `ixp_mem_read( )` reads the data into the SRAM transfer register, DRAM transfer register and General Purpose register depending on the type of memory specified. The `count` argument must be constant and must be in the range of 1 to 8. The `sig_ptr` argument should be the address of the user signal variable passed by direct reference. When memory type is DRAM, `sig_ptr` should point to signal pair.

Function Syntax

```
ixp_mem_read(
    MEM_TYPE mem_type,
    void *rd_reg,
    uint32_t address,
uint32_t count,
sync_t sync,
void *sig_ptr)
```

Input

`mem_type`	Memory type to read data. Check below for detail.
`rd_reg`	Address of the data buffer to read.
`address`	Address to read from.
`count`	Number of longwords to read.
`sync`	Type of synchronization to use. This argument must be a constant.
`sig_ptr`	Signal to raise upon completion.

Memory Type

The following memory types, as shown in Table 4-7, are valid inputs to this function:

- MEM_DRAM
- MEM_DRAM_S
- MEM_SRAM
- MEM_SRAM_D
- MEM_SCRATCH
- MEM_SCRATCH_D
- MEM_LM

Note: Signal declaration and read register declaration should use the same memory type as passed to this function.

Operation Type

This API doesn't take operation type as parameter. But the signal declaration macro for this API call needs to pass 'RW' as operation type.

Example

```
#define MyMem1MEM_DRAM_S
#define MyMem2MEM_SRAM

DS_READ_REG(MyMem1) uint32_t data[4];
DS_READ_REG(MyMem2) uint32_t data1[4];

SIG_COMB(MyMem1, RW) sig_rd;
SIG_COMB(MyMem2, RW) sig_rd1;
```

```
uint32_t address, address1, count, count1;
address = 0;
address1 = 123;
count = 4;
count 1= 6;

ixp_mem_read(MyMem1, (void *)data, address, count, sig_done, (void *)&sig_rd);
ixp_mem_read(MyMem2, (void *)data1, address1, count1, ctx_swap, (void *)&sig_rd1);

wait_for_all(&sig_rd);
```

4.6.4.2 `ixp_mem_read_ind()`

This function reads count longwords from the memory type specified at the specified address and places the data into the structure addressed by `rd_reg`. `ixp_mem_read_ind( )` reads the data into SRAM transfer register, DRAM transfer register and General Purpose register depending on the type of memory specified. The `count` argument can be constant or variable. It must be in the range of 0 to 15, where 0 means 1 LW and 15 means 16 LW. The `max_nn` argument must be constant and must indicate the maximum number of LWs possible. The `sig_ptr` argument should be the address of the user signal variable passed by direct reference. When memory type is DRAM, `sig_ptr` should point to signal pair.

Function Syntax

```
ixp_mem_read_ind(
    MEM_TYPE mem_type,
    void *rd_reg,
    uint32_t address,
    uint32_t max_nn,
    uint32_t count,
    sync_t sync,
    void *sig_ptr)
```

Input

`mem_type`	Memory type to read data. Check below for detail.
`rd_reg`	Address of the data buffer to read.
`address`	Address to read from.
`max_nn`	Maximum value of count variable possible.
`count`	Number of longwords to read. Value starts from 0, so if count is n, pass n-1.
`sync`	Type of synchronization to use. This argument must be a constant.
`sig_ptr`	Signal to raise upon completion.

Memory Type

The following memory types, as shown in Table 4-7, are valid inputs to this function:

- MEM_DRAM
- MEM_DRAM_S
- MEM_SRAM
- MEM_SRAM_D
- MEM_SCRATCH
- MEM_SCRATCH_D
- MEM_LM

Note: Signal declaration and read register declaration should use the same memory type as passed to this function.

Operation Type

This API doesn't take operation type as parameter. But signal declaration macro for this API call needs to pass 'RW' as operation type.

Example

```
#define MyMem1MEM_DRAM
#define MyMem2MEM_SCRATCH

DS_READ_REG(MyMem1) uint32_t data[12];
DS_READ_REG(MyMem2) uint32_t data1[10];

SIG_COMB(MyMem1, RW) sig_rd;
SIG_COMB(MyMem2, RW) sig_rd1;

uint32_t address, address1, count, count1;
address = 0;
address1 = 123;
count = 12;
count 1= 10;

ixp_mem_read_ind(MyMem1, (void *)data, address, 12, count, sig_done, (void *)&sig_rd);
ixp_mem_read_ind(MyMem2, (void *)data1, address1, 10, count1, ctx_swap, (void *)&sig_rd1);

wait_for_all(&sig_wr);
```

4.6.4.3 `ixp_mem_write()`

This function writes count longwords to the memory type specified at the specified address from the data into the structure addressed by `wr_reg`. `ixp_mem_write( )` writes the data to SRAM transfer register, DRAM transfer register and General Purpose register depending on type of memory specified. The `count` argument must be constant and must be in the range of 1 to 8. The argument `sig_ptr`, should be the address of the user signal variable passed by direct reference. When memory type is DRAM, `sig_ptr` should point to a signal pair.

Function Syntax

```
ixp_mem_write(
    MEM_TYPE mem_type,
    void *wr_reg,
    uint32_t address,
    uint32_t count,
    sync_t sync,
    void *sig_ptr)
```

Input

`mem_type`	Memory type to read data. Check below for detail.
`wr_reg`	Address of the data buffer to write.
`address`	Address to write to.
`count`	Number of longwords to write.
`sync`	Type of synchronization to use. This argument must be a constant.
`sig_ptr`	Signal to raise upon completion.

Memory Type

The following memory types, as shown in Table 4-7, are valid inputs to this function:

- MEM_DRAM
- MEM_DRAM_S
- MEM_SRAM
- MEM_SRAM_D
- MEM_SCRATCH
- MEM_SCRATCH_D
- MEM_LM

Note: Signal declaration and read register declaration should use the same memory type as passed to this function.

Operation Type

This API doesn’t take operation type as parameter. But signal declaration macro for this API call needs to pass ‘RW’ as operation type.

Example

```
#define MyMem1MEM_DRAM
#define MyMem2MEM_SCRATCH

DS_WRITE_REG(MyMem1) uint32_t data[2];
DS_WRITE_REG(MyMem2) uint32_t data1[3];

SIG_COMB(MyMem1, RW) sig_wr;
SIG_COMB(MyMem2, RW) sig_wr1;

uint32_t address, address1, count, count1;
address = 0;
address1 = 123;
count = 4;
count 1= 6;

// Write information in variable data and data1.

ixp_mem_write(MyMem1, (void *)data, address, count, sig_done, (void *)&sig_wr);
ixp_mem_write(MyMem2, (void *)data1, address1, count1, ctx_swap, (void *)&sig_wr1);

wait_for_all(&sig_wr);
```

4.6.4.4 `ixp_mem_write_ind()`

This function writes count longwords to the memory type specified at the specified address from data structure addressed by `wr_reg`. `ixp_mem_write_ind( )` writes the data from SRAM transfer register, DRAM transfer register and General Purpose register depending on type of memory specified. The count argument can be constant or variable. It must be in the range of 0 to 15, where 0 means 1 LW and 15 means 16 LW. The `max_nn` variable must be constant and must indicate the maximum number of LWs possible. The `sig_ptr` argument should be the address of the user signal variable passed by direct reference. When memory type is DRAM, `sig_ptr` variable should point to signal pair.

Function Syntax

```
ixp_mem_write_ind(
    MEM_TYPE mem_type,
    void *wr_reg,
    uint32_t address,
    uint32_t max_nn,
    uint32_t count,
    sync_t sync,
    void *sig_ptr)
```

Input

`mem_type`	Memory type to read data. Check below for detail.
`wr_reg`	Address of the data buffer to write.
`address`	Address to write to.
`max_nn`	Maximum number of longwords to write.
`count`	Number of longwords to read. Value starts from 0, so if count is n, pass n-1.
`sync`	Type of synchronization to use. This argument must be a constant.
`sig_ptr`	Signal to raise upon completion.

Memory Type

The following memory types, as shown in Table 4-7, are valid inputs to this function:

- MEM_DRAM
- MEM_DRAM_S
- MEM_SRAM
- MEM_SRAM_D
- MEM_SCRATCH
- MEM_SCRATCH_D
- MEM_LM

Note: Signal declaration and read register declaration should use the same memory type as passed to this function.

Operation Type

This API doesn't take operation type as parameter. But signal declaration macro for this API call needs to pass 'RW' as operation type.

Example

```
#define MyMem1MEM_SRAM
#define MyMem2MEM_SCRATCH

DS_WRITE_REG(MyMem1) uint32_t data[10];
DS_WRITE_REG(MyMem2) uint32_t data1[12];

SIG_COMB(MyMem1, RW) sig_wr;
SIG_COMB(MyMem2, RW) sig_wr1;

uint32_t address, address1, count, count1;
```

```
address = 0;
address1 = 123;
count = 10;
count 1= 12;

// Write information in variable data and data1.

ixp_mem_write_ind(MyMem1, (void *)data, address, 10, count, sig_done, (void *)&sig_wr);
ixp_mem_write_ind(MyMem2, (void *)data1, address1, 12, count1, ctx_swap, (void *)&sig_wr1);

wait_for_all(&sig_wr);
```

4.6.4.5 `ixp_mem_atomic()`

This function performs atomic operation based on memory type and operation type specified as input. In this API, operation type keeps the information about the type of operation needs to be performed. All the arguments are not needed for all the memory types. The `rd_reg` and `wr_reg` arguments are used to store information for writing and reading. The `sig_ptr` argument should be the address of the user signal variable passed by direct reference.

Function Syntax

```
ixp_mem_atomic(
    MEM_TYPE mem_type,
    OP_TYPE op_type,
    void *rd_reg,
    void *wr_reg,
    uint32_t address,
    sync_t sync,
    void *sig_ptr)
```

Input

`mem_type`	Memory type for data. Check below for detail.
`op_type`	Operation type. Check below for detail.
`rd_reg`	Address of data buffer to read.
`wr_reg`	Address of data buffer to write from.
`address`	Address to write to.
`sync`	Type of synchronization to use. This argument must be a constant.
`sig_ptr`	Signal to raise upon completion.

Memory Type

The following memory types, as shown in Table 4-7, are valid inputs to this function:

- MEM_SRAM

- MEM_SRAM_D
- MEM_SCRATCH
- MEM_SCRATCH_D
- MEM_LM

Note: Signal declaration and read register declaration should use the same memory type as passed to this function.

Operation Type

The following operation types, as shown in Table 4-8, are valid inputs to this function:

- SWAP
- SET
- CLR
- INCR
- DECR
- ADD
- SUB
- TEST_AND_SET
- TEST_AND_CLR
- TEST_AND_INCR
- TEST_AND_DECR
- TEST_AND_ADD
- TEST_AND_SUB

Note: Signal declaration and read register declaration should use the same operation type as passed to this function.

Example

```
#define MyMem1MEM_SCRATCH
#define MyMem2MEM_SRAM

DS_WRITE_REG(MyMem1) uint32_t data;
SIG_COMB(MyMem1, ADD) sig_wr;

uint32_t address, address1;
address = 0;
address1 = 123;
// Write information in variable data and data1.

ixp_mem_atomic(MyMem1, ADD, 0, (void *)&data, address, 0, 0);
ixp_mem_atomic(MyMem2, INCR, 0, 0, address1, 0, 0);
```

Protocol Support for the Microengine 5

The Protocol libraries provide support for various layer two and layer three protocols. This section describes this protocol support.

5.1 TCP/IP Protocol Interfaces

The TCP/IP Library provides functions supporting development of TCP/IP applications. This library is separate from the core functions in Level 1 and Level 0 of Point Reyes. Use of the TCP/IP Library is optional.

5.1.1 `ether`—Microengine

The `ether` API provides access to Ethernet protocol headers.

Table 5-1. `ether` API

Name	Description
`ether_addr_verify()`	Verifies the destination address, according to STD802.1D.
`ether_is_bpdu()`	Checks the Ethernet destination address for a Bridge Protocol Data Unit.
`ether_is_bcast()`	Checks the Ethernet destination address for broadcast.
`ether_is_mcast()`	Checks an Ethernet destination address verifying that it is a valid target for multicast.

5.1.1.1 API Functions

5.1.1.1.1 `ether_addr_verify()`

Verifies the destination address, according to STD802.1D. This function also checks for reserved addresses as well as verifies that the source address is neither a broadcast nor a multicast address.

Microengine Assembler Syntax

```
ether_addr_verify (out_exception, in_data, IN_HDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ether_addr_verify (void *in_data, IMMED IN_HDR_START_BYTE,
    mem_t PKT_MEM_TYPE);
```

Input

`in_data`	The buffer containing an Ethernet header.
`IN_HDR_START_BYTE`	The constant byte offset from start of `in_data` corresponding to the beginning of the Ethernet header.
`PKT_MEM_TYPE`	For *Microengine C only*: This is the source memory type for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GP_REG`, or `LOCALMEM`}. See Section 2.6.7, "`mem_t`."

Output/Returns

`out_exception`

Indicates completion status for the function. Returns one of:

- `0`, `ETHER_NO_EXCEPTION`—the address is valid
- `0x90`, `ETHER_INVALID_SRC_ADDRESS`—the packet is not destined to this forwarding engine
- `0x91`, `ETHER_RESERVED_ADDRESS`—the destination address is reserved per *RFC1812*

NOTE: The user can also pass in zero for `out_exception` and check the condition code Z in application code for the following values:

- `1`—the address is valid
- `0`—the address is invalid

Estimated Size

Ten to twenty instructions.

Microengine Assembler Example Usage One

```
.reg in_src, result
xbuf_alloc(in_src, 6, read_write);
immed32(in_src[0], 0x01020304);
immed32(in_src[1], 0xffff01ff);
immed32(in_src[2], 0xffffffff);
immed32(in_src[3], 0x0000efab);
immed32(in_src[4], 0x12340000);
ether_addr_verify(0, in_src, 0);
br=0[fail64#]
put_nv('F', case_num);
br[end64#]
fail64#:
//Failure
end64#:
xbuf_free(in_src);
```

Microengine Assembler Example Usage Two

```
.reg in_src, result
    xbuf_alloc(in_src, 6, read_write);
    immed32(in_src[0], 0x01020304);
    immed32(in_src[1], 0xffff01ff);
    immed32(in_src[2], 0xffffffff);
    immed32(in_src[3], 0x0000efab);
    immed32(in_src[4], 0x12340000);
    ether_addr_verify(0, in_src, 0);
    br!=0[fail64#]
        // success
    br[end64#]
    fail64#:
        // failure
    end64#:
    xbuf_free(in_src);
```

Microengine C Example Usage

```
uint32_t result, m_addr;
__declspec(sram_write_reg) uint32_t s_wr_reg[4];
__declspec(sram_read_reg) uint32_t s_rd_reg[4];
SIGNAL sig1;

s_wr_reg[0] = 0x02030405;
s_wr_reg[1] = 0x06070809;
s_wr_reg[2] = 0x0a0b0c0d;
s_wr_reg[3] = 0x0;
m_addr = 0x200;
sram_write(s_wr_reg, (__declspec (sram) uint32_t *)m_addr, 4, ctx_swap,
&sig1);
sram_read(s_rd_reg, (__declspec (sram) uint32_t *)m_addr, 4, ctx_swap, &sig1);
result = ixp_ether_addr_verify((void *)s_rd_reg, 0, SRAM_RD_REG);
if (result == ETHER_SUCCESS) {
    /* Success */
    } else {
    /* Failure */
    }
```

5.1.1.1.2 `ether_is_bpdu()`

Checks the Ethernet header type returning `ETHER_SUCCESS` for a Bridge Protocol Data Unit (BPDU).

Microengine Assembler Syntax

```
ether_is_bpdu (out_status, in_data, IN_HDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ether_is_bpdu (void *in_data, IMMED IN_HDR_START_BYTE,
    mem_t PKT_MEM_TYPE);
```

Input

`in_data`	For *Microengine C only*: This is the source memory type for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GP_REG`, or `LOCALMEM`}. See Section 2.6.7, "`mem_t`."
`IN_HDR_START_BYTE`	The constant byte offset from the start of `in_data` corresponding to the beginning of the ethernet header.
`PKT_MEM_TYPE`	For *Microengine C only*: This is the source memory type for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GP_REG`, or `LOCALMEM`}. See Section 2.6.7, "`mem_t`."

Output/Returns

`out_status`	Indicates completion status for the function. Returns one of: • `0`, `ETHER_SUCCESS`—the header indicates the destination is a Bridge Protocol Data Unit • `0x94`, `ETHER_BPDU_FAILURE`—the header indicates the destination is not a Bridge Protocol Data Unit **NOTE:** The user can also pass in zero for `out_stat` and check the condition code `Z` in application code. The value of `Z` can be: • `1`—the destination is a Bridge Protocol Data Unit • `0`—the destination is not a Bridge Protocol Data Unit

Estimated Size

Three to eight instructions.

Microengine Assembler Example Usage One

```
.reg in_src, result
//
xbuf_alloc(in_src, 6, read_write);
//
immed32(in_src[0], 0x01020304);
immed32(in_src[1], 0x05060a0b);
immed32(in_src[2], 0x0c0d0e0f);
immed32(in_src[3], 0xabcdefab);
ether_is_bpdu(result, in_src, 6);
xbuf_free(in_src);
```

Microengine Assembler Example Usage Two

```
.reg in_src, result
    xbuf_alloc(in_src, 6, read_write);
    immed32(in_src[0], 0xffffffff);
    immed32(in_src[1], 0xffff0102);
    immed32(in_src[2], 0xffffffff);
    immed32(in_src[3], 0xabcdefab);
    immed32(in_src[4], 0xabcdefab);
    ether_is_bpdu(0, in_src, 3);
    br!=0[fail7#]
    // Success
    br[end7#]
    fail7#:
    //Failure
    end7#:
    xbuf_free(in_src);
```

Microengine C Example Usage

```
uint32_t result, m_addr;
__declspec(sram_write_reg) uint32_t s_wr_reg[5];
__declspec(sram_read_reg) uint32_t s_rd_reg[5];
//
SIGNAL sig1;
s_wr_reg[0] = 0x02030405;
s_wr_reg[1] = 0x06070809;
s_wr_reg[2] = 0x0a0b0c0d;
s_wr_reg[3] = 0x00000000;
s_wr_reg[4] = 0x0;
m_addr = 0x100;
sram_write(s_wr_reg, (__declspec (sram) uint32_t *)m_addr, 5, ctx_swap,
        &sig1);
sram_read(s_rd_reg, (__declspec (sram) uint32_t *)m_addr, 5, ctx_swap, &sig1);
result = ixp_ether_is_bpdu(s_rd_reg, 0, SRAM_RD_REG);
if (result == ETHER_SUCCESS) {
    /* success */
    } else {
    /* failure */
    }
```

5.1.1.1.3 `ether_is_bcast()`

Checks the Ethernet destination address for broadcast.

Microengine Assembler Syntax

```
ether_is_bcast (out_status, in_data, IN_HDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ether_is_bcast (void *in_data, IMMED IN_HDR_START_BYTE,
    mem_t PKT_MEM_TYPE);
```

Input

`in_data`	The buffer with an Ethernet header.
`IN_HDR_START_BYTE`	The constant byte offset from the start of `in_data` corresponding to the beginning of the Ethernet header.
`PKT_MEM_TYPE`	For *Microengine C only*: This is the source memory type for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GP_REG`, or `LOCALMEM`}. See Section 2.6.7, "`mem_t`."

Output/Return

`out_status` Indicates completion status for the function. Returns one of:

- `0, ETHER_SUCCESS`—the address is an address for broadcast
- `0x92, ETHER_BCAST_FAILURE`—the address is *not* an address for broadcast

NOTE: The condition code is also set. The user can pass zero in to `out_status` and check condition code `Z`. The value of `Z` is one of:

- `1`—the address is a broadcast address
- `0`—the address is not a broadcast address

Estimated Size

- Microengine Assembler—4 to 10 instructions
- Microengine C—8 to 15 instructions

Microengine Assembler Example Usage One

```
.reg in_src, result
xbuf_alloc(in_src, 4, read_write);

immed32(in_src[0], 0xffffffff);
immed32(in_src[1], 0xffff0506);
immed32(in_src[2], 0x01020304);
immed32(in_src[3], 0xabcdefab);
ether_is_bcast(result, in_src, 0);
CHECK_VAL_EQ(result, ETHER_SUCCESS, 0);
xbuf_free(in_src);
```

Microengine Assembler Example Usage Two

```
.reg in_src, result
xbuf_alloc(in_src, 4, read_write);
immed32(in_src[0], 0xAAAAAAff);
immed32(in_src[1], 0xffff01ff);
immed32(in_src[2], 0x01020304);
immed32(in_src[3], 0xabcdefab);
ether_is_bcast(0, in_src, 3);
br!=0[passfail1#]
put_nv('F', case_num);// Success
//
br[end1#]
failpass1#:
//Failure
put_nv('P', case_num);
end1#:
xbuf_free(in_src);
```

Microengine C Example Usage

```
uint32_t result, m_addr;
__declspec(sram_write_reg) uint32_t s_wr_reg[5];
__declspec(sram_read_reg) uint32_t s_rd_reg[5];
SIGNAL sig1;
/* Initialize Ethernet header */
s_wr_reg[0] = 0xffffffff;
s_wr_reg[1] = 0xffffffff;
s_wr_reg[2] = 0xffffffff;
s_wr_reg[3] = 0x456789ab;
s_wr_reg[4] = 0x56789abc;
m_addr = 0x100;
sram_write(s_wr_reg, (__declspec(sram) uint32_t *)m_addr, 5, ctx_swap, &sig1);
sram_read(s_rd_reg, (__declspec(sram) uint32_t *)m_addr, 5, ctx_swap, &sig1);
result = ixp_ether_is_bcast(s_rd_reg, 0, SRAM_RD_REG);
if (result == 0) {
    /* Success */
    } else {
    /* Failure */
    }
```

5.1.1.1.4 `ether_is_mcast()`

Checks an Ethernet destination address verifying that it is a valid target for multicast.

Microengine Assembler Syntax

```
ether_is_mcast (out_status, in_data, IN_HDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ether_is_mcast (void *in_data, IMMED IN_HDR_START_BYTE,
    mem_t PKT_MEM_TYPE);
```

Input

`in_data`	The buffer with an Ethernet header.
`IN_HDR_START_BYTE`	The constant byte offset from the start of `in_data` corresponding to the beginning of the Ethernet header.
`PKT_MEM_TYPE`	For *Microengine C only*: This is the source memory type for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GP_REG`, or `LOCALMEM`}. See Section 2.6.7, "`mem_t`."

Output/Returns

`out_status` Indicates completion status for the function. Returns one of:

- `0`, `ETHER_SUCCESS`—the address is a valid multicast target
- `0x93`, `ETHER_MCAST_FAILURE`—the address is *not* a valid multicast target

NOTE: The condition code is also set. The user can pass zero in to `out_status` and check the condition code `Z`. The return values for `Z` are:

- `1`—the address is a valid multicast target
- `0`—the address is *not* a valid multicast target

Estimated Size

Three to eight instructions.

Microengine Assembler Example Usage One

```
.reg in_src, result

xbuf_alloc(in_src, 6, read_write);

immed32(in_src[0], 0x01000203);
immed32(in_src[1], 0xffff0106);
immed32(in_src[2], 0x01020304);
immed32(in_src[3], 0xabcdefab);
ether_is_mcast(result, in_src, 0);
CHECK_VAL_EQ(result, ETHER_SUCCESS, case_num);
xbuf_free(in_src);
```

Microengine Assembler Example Usage Two

```
.reg in_src, result
xbuf_alloc(in_src, 4, read_write);
immed32(in_src[0], 0xAAAAAA01);
immed32(in_src[1], 0xffff01ff);
immed32(in_src[2], 0x01020304);
immed32(in_src[3], 0xabcdefab);
ether_is_mcast(0, in_src, 3);
br!=0[fail1#]
    // Success
br[end1#]
fail1#:
    // failure
end1#:
xbuf_free(in_src);
```

Microengine C Example Usage

```
uint32_t result, m_addr;
__declspec(sram_write_reg) uint32_t s_wr_reg[6];
__declspec(sram_read_reg) uint32_t s_rd_reg[6];
SIGNAL sig1;
s_wr_reg[0] = 0x01010101;
s_wr_reg[1] = 0xffffffff;
s_wr_reg[2] = 0xffffffff;
s_wr_reg[3] = 0x456789ab;
s_wr_reg[4] = 0x56789abc;
s_wr_reg[5] = 0xfff0ff0f;
m_addr = 0x100;
sram_write(s_wr_reg, (__declspec (sram) uint32_t *)m_addr, 6, ctx_swap,
    &sig1);
sram_read(s_rd_reg, (__declspec (sram) uint32_t *)m_addr, 6, ctx_swap, &sig1);
result = ixp_ether_is_mcast(s_rd_reg, 0, SRAM_RD_REG);
if (result == ETHER_SUCCESS) {
    /* Success */
    } else {
     /* failure */
     }
```

5.1.2 ipv4—Microengine

The `ipv4` interface defines the following functions for processing the IP header. Refer to [RFC791], [RFC1812], and other associated internet documents for a description of internet protocol fields and functionality performed by routers.

Using Microengine Assembler macros, `xbuf_extract()` can be used to get any of the IP fields. Using Microengine C language, the packet is cast as an `ipv4_header` struct, and fields can be accessed with C notation for the struct member variables.

Table 5-2. ipv4 API

Name	Description
`ipv4_addr_verify()`	Checks for invalid router addresses according to RFC1812: • Network 0 • 127—loopback address • 240—greater than Class D
`ipv4_cksum_verify()`	Verifies an IPv4 header checksum.
`ipv4_ is_dbcast()`	Checks an IPv4 destination address for directed broadcast.
`ipv4_is_snmp()`	Checks whether the protocol is SNMP, checking for UDP or TCP, and a port number of 160 or 161.
`ipv4_modify()`	Modifies an IPv4 header—the header's time-to-live (TTL) and checksum fields.
`ipv4_proc()`	Verifies an IPv4 header—that is, the addresses, checksum, header length, and TTL. Modifies TTL and checksum.
`ipv4_route_lookup()`	Performs lookup of a route entry using a longest prefix match.
`ipv4_verify()`	Verifies an IPv4 header—that is, the addresses, checksum, header length, and TTL.
`ipv4_total_len_verify()`	Verifies that `total_len` is at least 20.
`ipv4_ttl_verify()`	Verifies that TTL is greater than one.

5.1.2.1 Defined Types, Enumerations, and Data Structures

5.1.2.1.1 IPv4 Header Data Structure

The `ipv4` interface defines the following struct for access to the IP header. This provides read and write access to the header fields.

Microengine C Syntax

```
typedef struct ixp_ipv4_header {
    unsigned int version: 4;
    unsigned int length: 4;
    unsigned int type_of_service: 8;
    unsigned int total_length: 16;
    unsigned int id: 16;
    unsigned int flags: 3;
    unsigned int fragment: 13;
    unsigned int time_to_live: 8;
    unsigned int protocol: 8;
    unsigned int checksum: 16;
    unsigned int source_addr: 32;
    unsigned int dest_addr: 32;
};
```

5.1.2.2 API Functions

5.1.2.2.1 `ipv4_addr_verify()`

Checks for invalid router addresses according to RFC1812:

- Network 0
- 127—loopback address
- 240—greater than Class D

Microengine Assembler Syntax

```
ipv4_addr_verify (out_exception, in_addr);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_addr_verify (uint32_t in_addr);
```

Input

`in_addr`	The IP source or destination address.

Output/Returns

`out_exception` *or* Return Value	Sets the `Z` flag in condition code. If `Z` equals: • `0`—means address is well-formed • `1`—means the address is bad

Estimated Size

Seven instructions.

Microengine Assembler Example Usage

```
ipv4_addr_verify(exception, source_addr);
```

Microengine C Example Usage

```
exception = ixp_ipv4_addr_verify(ip.source_addr);
```

5.1.2.2.2 `ipv4_cksum_verify()`

Verifies an IPv4 header checksum. The `ip_header` start byte is at:
`in_data + in_iphdr_start_byte`

Microengine Assembler Syntax

```
ipv4_cksum_verify (in_data, in_iphdr_start_byte);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_cksum_verify (
    void *in_data,
    uint32_t in_iphdr_start_byte,
    mem_t in_data_mem_type);
```

Input

`in_data`	The buffer with IP header whose checksum is to be verified.
`in_iphdr_start_byte`	The constant byte offset from the start of `in_data` to the first byte of the IP header.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}

Output/Returns

Return Value	For Microengine C, the function return value provides the outcome of the verify operation. • `0`—no exception • `IP_BAD_CHECKSUM`—incorrect checksum, per [RFC791] For Microengine Assembler sets condition code flag `Z`: • 0—incorrect checksum, per [RFC791] • 1—the checksum is good

Estimated Size

- 13 instructions for alignment zero
- 17 instructions for alignment two
- 24 instructions for alignment one or three

Microengine Assembler Example Usage

```
ipv4_cksum_verify($packet, 14);
br>0[discard#]
```

Microengine C Example Usage

```
exception = ixp_ipv4_cksum_verify(&packet, 14, SRAM_RD_REG);
```

5.1.2.2.3 `ipv4_ is_dbcast()`

Determines if an IPv4 destination address is a directed broadcast destination.

Microengine Assembler Syntax

```
ixp_ipv4_is_dbcast (out_status, in_dest_addr, SRAM_DBCAST_BASE, \
    SRAM_READ_BLOCK_COUNT);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_is_dbcast (
    uint32_t in_dest_addr,
    volatile __declspec(sram) void *SRAM_DBCAST_BASE,
    uint32_t SRAM_READ_BLOCK_COUNT);
```

Input

`in_dest_addr`	The IP source or destination address.
`SRAM_DBCAST_BASE`	The constant base address of a dbcast address table.
`SRAM_READ_BLOCK_COUNT`	The constant to specify the SRAM read block count in longwords. This dictates the number of longwords per SRAM read. The value of `SRAM_READ_BLOCK_COUNT` must be divisible by two and within the range of two to 16, inclusive.

Output/Returns

`out_status`	The result of checking the directed broadcast status of the address `in_dest_addr`. • `TRUE`—the address is a directed broadcast destination • `FALSE`—the address is *not* a directed broadcast destination

Estimated Size

Sixteen to 21 instructions per lookup.

Microengine Assembler Example Usage

```
ipv4_is_dbcast (status, dest_addr, 0x1000, 8);
```

Microengine C Example Usage

```
status = ixp_ipv4_ is_dbcast (ip.dest_addr, 0x2000, 4);
```

5.1.2.2.4 `ipv4_is_dbcast_lm()`

Checks IPv4 destination address for directed broadcast.

Directed Broadcast Table description:

Local memory implementation Directed Broadcast will have two tables,

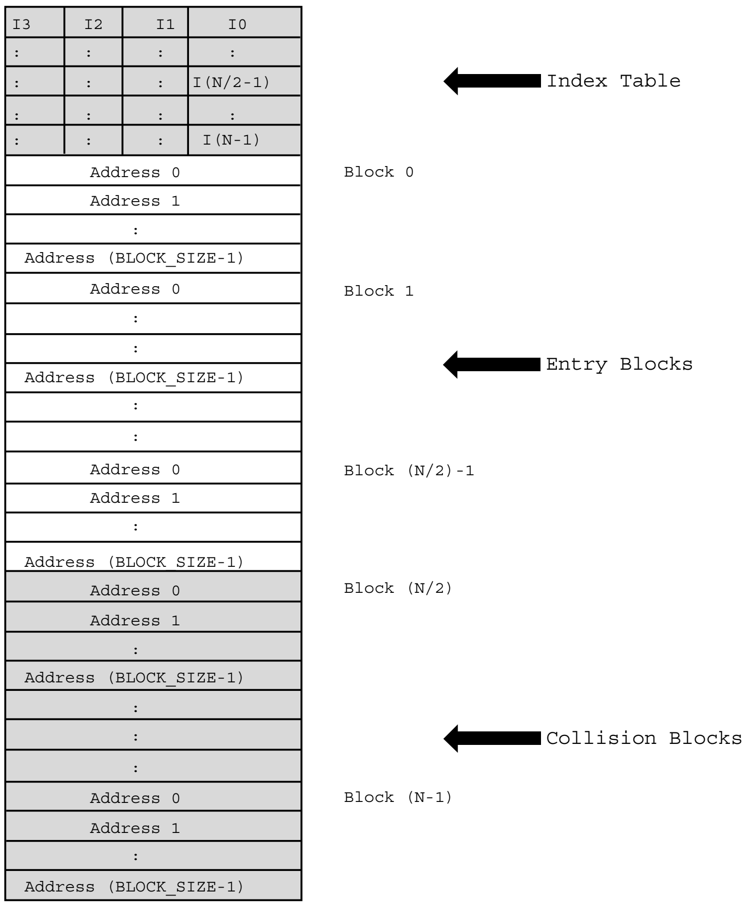
1. Entry table

   The entry table is divided into blocks of size `LM_DBCAST_BLOCK_SIZE`. The number of blocks (`LM_DBCAST_BLOCK_COUNT`), is further divided into two sets—the first set is for direct entries block and another set is for collision blocks.

2. Index table

   The index table is used to store the *next block pointer* for respective blocks. Each entry in the index table is 8 bits wide and corresponds to the respective block. Therefore each local memory entry (32 bit) can hold four *next block pointers*. Size of the index table is equal to `N` bytes—or `N/4` longwords, where `N` is equal to the `LM_DBCAST_BLOCK_COUNT`

Figure 5-1. Memory Map for `ipv4_is_dbcast_lm()`



Algorithm

1. Check passed parameters for correctness.
2. Divide number of blocks into two sets.
   a. `LM_DBCAST_BLOCK_COUNT` divided by two for the number of blocks used to store entries
   b. `LM_DBCAST_BLOCK_COUNT` divided by two for the number of blocks used to store collision entries
3. Calculate the sizes and create the two tables
   a. An entry table

      The table to hold DBCAST entries

   b. An index table

      The table to store the next indexes for the entry table blocks. Each byte in this block will correspond to the block number and will store the *Next Index* for that block. The size of this table will be `LM_DBCAST_BLOCK_SIZE` bytes or `LM_DBCAST_BLOCK_SIZE` divided by four longwords.
4. Compute hash on the destination address. Given an IP address of form `A.B.C.D` produce a hash result of the form `A^(A.B)^(A.B.C)^(A.B.C.D)`
5. Based on the number of direct entry blocks extract the last `N` bits of hash result to get the `BLOCK_NUMBER` where `N` can be derived from the relation:

   (LM_DBCAST_BLOCK_SIZE / 2) = 2 ^ N

6. Set the search starting index according to this assignment:
   ```
   *l$Index = LM_DBCAST_ENTRY_TABLE_BASE +
           (BLOCK_NUMBER *  LM_DBCAST_BLOCK_SIZE )
   ```
   That is, start the search from this block onwards.
7. Perform a read of longwords at local memory location indicated by `*l$Index`.
8. Compare each local memory entry with the destination IP address with the following results:
   a. If the entry matches with the destination IP address return success.
   b. If a zero entry is found return failure.
   c. If end-of-block is found then go to the index table and retrieve the *Next Block* pointer and start the search from step 6

Microengine Assembler Syntax

```
ipv4_is_dbcast_lm ( out_status, in_dest_addr, LM_DBCAST_BASE, \
    LM_DBCAST_ BLOCK_COUNT, LM_DBCAST_BLOCK_SIZE, LM_INDEX);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_is_dbcast_lm (
    uint32_t in_dest_addr,
    IMMED LM_DBCAST_BASE,
    IMMED LM_DBCAST_BLOCK_COUNT,
    IMMED LM_DBCAST_BLOCK_SIZE,
    IMMED LM_INDEX );
```

Input

`in_dest_addr`	The register containing the IPv4 source or destination address.
`LM_DBCAST_BASE`	The constant local-memory base address of the dbcast address table.
`LM_DBCAST_BLOCK_COUNT`	A constant to specify the number of blocks. The value of `LM_DBCAST_CACHE_SIZE` must be a power of two.
`LM_DBCAST_BLOCK_SIZE`	A constant to specify the size of the block in terms of the number of longword entries per block—this constant must be a power of two.
`LM_INDEX`	A constant specifying the local memory index—this value is either zero or one.

Output/Returns

`out_status` *or* Return Value	The returned directed broadcast status of the destination address. This value is one of: • `FALSE`—this is not a directed broadcast address • `TRUE`—a directed broadcast address

Notes on Inputs

1. Internally `LM_DBCAST_INDEX_TABLE_SIZE` will be calculated as `LM_DBCAST_BLOCK_COUNT` divided by four longwords.
2. Therefore the total local memory longword entries required, starting at `LM_DBCAST_BASE`, will be,

   ```
   (LM_DBCAST_INDEX_TABLE_SIZE  +  (LM_DBCAST_BLOCK_COUNT *
          LM_DBCAST_BLOCK_SIZE))
   ```

   and the total number must not exceed 640 `LM` entries
3. `LM_DBCAST_BLOCK_COUNT` cannot exceed 256—this is because each index table entry is 1 byte long and can hold maximum value of `0xFF`.

Estimated Size

- 35 instructions for Microengine Assembler
- 47 instructions for Microengine C

Microengine C Example Usage

```
status = ixp_ipv4_ is_dbcast (ip.dest_addr, 0, 64, 8, 0 );
```

Microengine Assembler Example Usage

```
ipv4_is_dbcast (status, dest_addr, 0, 64, 8, 0);
br=0[next_location#:]
```

5.1.2.2.5 `ipv4_is_snmp()`

Determines if a protocol is SNMP; checks for UDP or TCP and checks if port number is 160 or 161.

Microengine Assembler Syntax

```
ipv4_is_snmp (out_status, in_data, in_iphdr_start_byte);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_is_snmp (
    void *in_data,
    uint32_t in_iphdr_start_byte,
    mem_t in_data_mem_type);
```

Input

`in_data`	The buffer with IP header whose protocol is in question.
`in_iphdr_start_byte`	The constant byte offset from the start of `in_data` to the first byte of the IP header.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output/Returns

`out_status`	The result of checking the identity of the protocol initiating the packet in the buffer specified by `in_data`. • `TRUE`—the protocol is SNMP • `FALSE`—the protocol is *not* SNMP

Estimated Size

Seven instructions.

Microengine Assembler Example Usage

```
ipv4_is_snmp(status, $$packet, 14);
```

Microengine C Example Usage

```
status = ixp_ipv4_is_snmp(&packet, 14, SRAM_RD_REG);
```

5.1.2.2.6 `ipv4_modify()`

Processes an IP header that starts at a fixed-byte offset from the start of `in_data`. Modifies the time-to-live and the checksum and writes the new header at `out_data`. If IP header is not on a longword boundary, merges `in_prepend` with the first word. Writes the modified IP header to the specified `out_data` buffer.

For Microengine Assembler, the read and write start bytes must be the same.

Note: IP options are not supported.

Microengine Assembler Syntax

```
ipv4_modify(out_data, IN_IPHDR_WR_START_BYTE, in_prepend, in_data, \
    IN_IPHDR__RD_START_BYTE);
```

Microengine C Syntax

```
INLINE void ixp_ipv4_modify(
    void *out_data,
    IMMED IN_IPHDR_WR_START_BYTE,
    mem_t OUT_DATA_MEM_TYPE,
    uint32_t in_prepend,
    void *in_data,
    IMMED IN_IPHDR_RD_START_BYTE,
    mem_t IN_DATA_MEM_TYPE);
```

Input

`IN_IPHDR_WR_START_BYTE`	The constant byte offset from the start of `out_data` to the beginning of the modified IP header.
`OUT_DATA_MEM_TYPE`	The memory type for the `out_data` parameter. This value is one of {`SRAM_WR_REG`, `DRAM_WR_REG`, `LOCALMEM`, `GP_REG`}.
`in_prepend`	If the byte alignment within the first IP longword is non-zero this is the longword containing the byte or bytes to be merged with the first word. The bytes to be merged must be at the exact byte locations that they occupy in the output header. All other bytes must be zero.
`in_data`	The buffer containing the IP header to modify.
`IN_IPHDR_RD_START_BYTE`	The constant byte offset from the start of `in_data` to the first byte of the IP header to be modified.
`IN_DATA_MEM_TYPE`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output

`out_data`	The new IP header with modified fields.

Estimated Size

- 14 instructions for alignment 0
- 15 instructions for alignment 2
- 20 to 21 instructions for alignment 1 or 3

Microengine Assembler Example Usage

```
move(in_prepend, 0xb0b1b200)
ipv4_modify($$modified_packet, 3, in_prepend, $packet, 3);
```

Microengine C Example Usage

```
move(in_prepend, 0xb0b10000);
ixp_ipv4_modify(
    &modified_packet,
    2,
    SRAM_WR_REG,
    in_prepend,
    &packet,
    2,
    SRAM_RD_REG);
```

5.1.2.2.7 `ipv4_proc()`

Processes an IP header as follows:

1. Verifies the IPv4 header's addresses, checksum, and header length. See RFC791 and RFC1812. The compile time define `RFC1812_SHOULD` produces code compliant with those requirements identified in the RFCs by *MUST* and *SHOULD*. In absence of `RCFC1812_SHOULD` compiler directive the code is conditionally compliant, meeting only those requirements identified by `MUST`.
2. Modifies the time-to-live and the checksum and writes the new header at `out_data`. See RFC791.

If the IP header is positioned at a longword boundary, `in_prepend` is merged with the first word.

Source `ip_header` start byte is at:
`in_data + iphdr_wr_start_byte`

Modified `ip_header` start byte is at:
`out_data + iphdr_rd_start_byte`

Microengine Assembler Syntax

```
ipv4_proc(out_exception, out_data, IPHDR_WR_START_BYTE, in_prepend, \
in_data, IPHDR_RD_START_BYTE)
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_proc(
    void *out_data,
    uint32_t iphdr_wr_start_byte,
    mem_t out_data_mem_type,
    uint32_t in_prepend,
    void *in_data,
    uint32_t iphdr_rd_start_byte,
    mem_t in_data_mem_type);
```

Input

`iphdr_wr_start_byte`	The constant byte offset from the start of `out_data` to the byte at which to start writing the modified IP header.
`out_data_mem_type`	The memory type for the `out_data` parameter. This value is one of {`SRAM_WR_REG`, `DRAM_WR_REG`, `LOCALMEM`, `GP_REG`}.
`in_prepend`	If the byte alignment within the first IP longword is non-zero this is the longword containing the byte or bytes to be merged with the first word. The bytes to be merged must be at the exact byte locations that they occupy in the output header. All other bytes must be zero.

Input (Continued)

`in_data`	The buffer containing the IP header.
`iphdr_rd_start_byte`	The constant byte offset from the start of `in_data`.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output/Returns

`out_exception`	Reports the return status of this operation. • `0`—no exception occurred, the modification was successful • `IP_BAD_PACKET_LENGTH`—the packet length is less than 20 bytes, per [RFC1812] • `IP_BAD_TOTAL_LENGTH`—the IP header is malformed with a length less than 20 bytes, per [RFC791] • `IP_BAD_CHECKSUM`—the checksum is incorrect, per [RFC791] • `IP_INVALID_ADDR`—illegal IP address, per [RFC1812]
`out_data`	The new IP header with modified fields.

Estimated Size

A minimum of eight instructions if the address is invalid.

If there is no exception—that is, if the address is valid:

- 48 instructions for alignment 0
- 53 instructions for alignment 2
- 60 to 63 instructions for alignment 1 or 3

Microengine Assembler Example Usage

```
move(in_prepend, 0xb0b1b200);
ipv4_proc(exception, $$modified_packet, 15, in_prepend, $packet, 15);
```

Microengine C Example Usage

```
move(in_prepend, 0xb0000000);
exception = ixp_ipv4_proc(
    &modified_packet,
    9,
    SRAM_WR_REG,
    in_prepend,
    &packet,
    9,
    SRAM_RD_REG);
```

5.1.2.2.8 `ipv4_route_lookup()`

Performs an IPv4 longest-prefix match lookup. Uses tables written by the Intel® XScale™ core Route Table Manager.

Microengine Assembler Syntax

```
ipv4_route_lookup (out_rt_ptr, in_ip_da, in_trie_addr, TRIE_DEPTH);
```

Microengine C Syntax

```
uint32_t ixp_ipv4_route_lookup(
    uint32_t in_ip_da,
    volatile __declspec(sram) void* trie_addr,
    trie_t TRIE_DEPTH);
```

Input

`in_ip_da`	The IP destination address.
`trie_addr`	The address of the trie table.
`TRIE_DEPTH`	The depth of trie—currently only `TRIE5` is supported. • `TRIE5`—uses a dual lookup of a high-64K entry table and trie block lookups consuming a maximum of five SRAM reference signals See Section 2.6.6, "`trie_t`."

Output/Returns

`out_rt_ptr`	A general-purpose register containing the resulting route pointer, an index to the route entry. The value is zero if there is no route entry.

Estimated Size

A minimum size of:

- 16 instructions to perform a match after the first lookup in a long path table
- 23 instructions to perform a match after the first lookup in a short path table

A maximum size of 83 instructions.

Microengine Assembler Example Usage

```
ipv4_route_lookup(route_entry_index, ip_da, trie_addr, TRIE5);
```

Microengine C Example Usage

```
route_entry_index = ixp_ipv4_route_lookup(ip_da, trie_addr, TRIE5);
```

5.1.2.2.9 `ipv4_verify()`

Verifies the IPv4 header's addresses, checksum, and header length. See RFC791 and RFC1812. The compile time define `RFC1812_SHOULD` produces code compliant with those requirements identified in the RFCs by *MUST* and *SHOULD*. In the absence of the `RCFC1812_SHOULD` compiler directive the code is conditionally compliant, meeting only those requirements identified by `MUST`.

Optionally, if `SKIP_DEST_ADDR_CHECK` is defined then the destination IP address check is skipped.

Microengine Assembler Syntax

```
ipv4_verify (out_exception, in_data, in_iphdr_start_byte);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_verify (
    void *in_data,
    uint32_t in_iphdr_start_byte,
    mem_t in_data_mem_type);
```

Input

`in_data`	The buffer containing the IPv4 header to be verified.
`in_iphdr_start_byte`	The constant byte offset from the start of `in_data` to the IP header to verify.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output/Returns

`out_exception`	The exception: • `0`—the IPv4 header was successfully verified and raised no exceptions • `IP_BAD_TOTAL_LENGTH`—verification failed due to an IP header length of less than 20 bytes, per [RFC791] • `IP_BAD_CHECKSUM`—incorrect checksum, per [RFC791] • `IP_INVALID_ADDR`—illegal IP address, per [RFC1812]

Estimated Size

A minimum size of eight instructions if there is an address exception.

If there is no address exception:

- 34 instructions for alignment 0
- 38 instructions for alignment 2
- 40 to 42 instructions for alignment 1 or 3

Microengine Assembler Example Usage

```
ipv4_verify(exception, $$packet, 14);
```

Microengine C Example Usage

```
exception = ixp_ipv4_verify(&packet, 14, SRAM_RD_REG);
```

5.1.2.2.10 `ipv4_total_len_verify()`

Verifies that the total-length field in an IPv4 header is at least 20.

Microengine Assembler Syntax

```
ipv4_total_len_verify (in_data, in_iphdr_start_byte);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_total_len_verify (
    void *in_data,
    uint32_t in_iphdr_start_byte,
    mem_t in_data_mem_type);
```

Input

`in_data`	The buffer with the IP header of interest.
`in_iphdr_start_byte`	A constant byte offset from the start of `in_data` to the IP header.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output/Returns

Return Value	For Microengine C returns one of the following values: • 0—if no exception • `IP_BAD_TOTAL_LENGTH`—if total length is less than 20 bytes For Microengine Assembler sets condition code flags—one of: • greater than zero—the total length is bad • zero—the total length is valid

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
ipv4_total_len_verify($$packet, 14);
```

Microengine C Example Usage

```
exception = ixp_ipv4_total_length_verify(&packet, 14);
```

Note: The C function name is `ixp_ipv4_total_length_verify` for this release only and is changing to `ixp_ipv4_total_len_verify` in the next release.

5.1.2.2.11 `ipv4_ttl_verify()`

Verifies that the time-to-live field in an IPv4 header is at least two.

Microengine Assembler Syntax

```
ipv4_ttl_verify (in_data, in_iphdr_start_byte);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv4_ttl_verify (
    void *in_data,
    uint32_t in_iphdr_start_byte,
    mem_t in_data_mem_type);
```

Input

`in_data`	The buffer with the IP header of interest.
`in_iphdr_start_byte`	A constant byte offset from the start of `in_data` to the IP header.
`in_data_mem_type`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, `GP_REG`}.

Output/Returns

Return Value

For Microengine C returns one of the following values:

- 0—if no exception
- `IP_BAD_TTL`—if total length is less than 20 bytes

For Microengine Assembler sets condition code flags with one of the following values:

- greater than zero—the time-to-live is bad
- zero—the time-to-live is valid

Estimated Size

One instruction.

Microengine Assembler Example Usage

```
ipv4_ttl_verify($$packet, 1);
```

Microengine C Usage

```
exception = ixp_ipv4_ttl_verify(&packet, 0, SRAM_RD_REG);
```

5.1.2.2.12 `ipv4_n_tuple_extract()`

Extract an n-tuple value from an IP header.

Microengine Assembler Syntax

```
ipv4_n_tuple_extract(out_ntuple, in_data, IN_IPHDR_START_BYTE, TUPLE_TYPE);
```

Microengine C Syntax

```
INLINE void ixp_ipv4_n_tuple_extract(
    void *out_ntuple,
    void *in_data,
    IMMED IN_IPHDR_START_BYTE,
    tuple_t TUPLE_TYPE,
    mem_t IN_DATA_MEM_TYPE);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`TUPLE_TYPE`	The n-tuple type. This value is one of: • `FIVE_TUPLE` For `FIVE_TUPLE` table lookup this operation uses source port and address, destination port and address, and protocol values. • `SIX_TUPLE` For `SIX_TUPLE` table lookup this operation uses source port and address, destination port and address, protocol, and DSCP values.
`IN_DATA_MEM_TYPE`	The memory type for the `in_data` argument. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

Output/Returns

`out_ntuple`	An n-tuple value. The returned n-tuple value uses the `tuple_t` data structure.

Estimated Size

Five to eight instructions.

Microengine Assembler Example Usage

```
.sig sig1
.reg base
xbuf_alloc ($in_header, 7, READ)
xbuf_alloc ($out_header, 7, WRITE)
xbuf_alloc($n_tuple, 5, WRITE)
//initialize ip header in sram
immed32($out_header[0], 0x4510006e)
immed32($out_header[1], 0xabcd0000)
immed32($out_header[2], 0x10115270)
immed32($out_header[3], 0x0a031380)
immed32($out_header[4], 0x18877638)
immed32($out_header[5], 0x10000023)
immed[base, 0x1000]
sram_write($out_header[0], base, 0, 6, sig1, sig1, ___)
sram_read($in_header[0], base, 0, 6, sig1, sig1, ___)
//extract FIVE_TUPLE values in $n_tuple
ipv4_n_tuple_extract($n_tuple, $in_header, 0, FIVE_TUPLE)
//now n_tuple contains five tuple values
xbuf_free($in_header)
xbuf_free($out_header)
xbuf_free($n_tuple)
```

Microengine C Example Usage

```
__declspec(sram_read_reg) uint32_t s_in_header[8];
__declspec(sram_write_reg) n_tuple_t out_tuple;
__declspec(sram_write_reg) uint32_t tcpIpHdr[12];
SIGNAL sig1;
//Initialize TCP/IP header with 0 offset
tcpIpHdr[0] = 0x4500006e;
tcpIpHdr[1] = 0xabcd0000;
tcpIpHdr[2] = 0x10115270;
tcpIpHdr[3] = 0x0a031380;
tcpIpHdr[4] = 0x18877638;
tcpIpHdr[5] = 0x08000400;
tcpIpHdr[6] = 0x00000000;
base = 0x1000;
sram_write(tcpIpHdr, (__declspec(sram) U32 *)0x1000, 7, ctx_swap, &sig1);
sram_read(s_in_header, (__declspec(sram) U32 *) base, 7, ctx_swap, &sig1);
ixp_ipv4_n_tuple_extract((void*)&out_tuple,SRAM_WR_REG,(void*)s_in_header,\
    0,FIVE_TUPLE, SRAM_RD_REG);
```

5.1.2.2.13 `ipv4_n_tuple_lookup()`

Looks up a route entry using an n-tuple index of IPv4 source port and address, destination port and address, protocol and DSCP values. Currently this operation supports two types of n-tuple lookups:

- `FIVE_TUPLE`

  For `FIVE_TUPLE` lookup the operation uses source port and address, destination port and address, and protocol values.
- `SIX_TUPLE`

  For `SIX_TUPLE` lookup the operation uses source port and address, destination port and address, protocol, and DSCP values.

For both `FIVE_TUPLE` or `SIX_TUPLE` lookup the operation uses tables written by the Hash Table Database Manager.

See section Section 4.3.1, "Hash Algorithm" for a description of the hash trie algorithm.

Microengine Assembler Syntax

```
ipv4_n_tuple_lookup (out_index, in_data, IN_IPHDR_START_BYTE, \
    in_trie_base_addr, KEY_DATA_SD_BASE, TRIE_TYPE, TUPLE_TYPE);
```

Microengine C Syntax

```
uint32_t ixp_ipv4_n_tuple_lookup(
    void *in_data,
    IMMED IN_IPHDR_START_BYTE,
    mem_t IN_DATA_MEM_TYPE,
    tuple_t TUPLE_TYPE,
    volatile __declspec(sram) void* in_trie_base_addr,
    uint32_t KEY_DATA_SD_BASE,
    uint32_t TRIE_TYPE);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`IN_DATA_MEM_TYPE`	The memory type for the `in_data` argument. This is the source memory for packet data. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `GPR_BUF`, or `LOCALMEM`}.
`TUPLE_TYPE`	The n-tuple type. This value is one of: • `FIVE_TUPLE` For `FIVE_TUPLE` table lookup this operation uses source port and address, destination port and address, and protocol values. • `SIX_TUPLE` For `SIX_TUPLE` table lookup this operation uses source port and address, destination port and address, protocol, and DSCP values.

Input

`in_trie_base_addr`	The address of the trie table.
`TRIE_TYPE`	The index bits used to address the hash trie: This value is one of: • `HASH_16_4`—first lookup uses 16 bits of index, subsequent lookups uses four bits of index • `HASH_FOLD_16`—first lookup XORs the initial index to reduce it by half, then performs the trie lookup using 16 bits of half-index, with subsequent chain search until there is no collision
`KEY_DATA_SD_BASE`	The base address of the key database located in SDRAM.

Output/Returns

`out_index`	The index of the route entry if the lookup is successful and zero if the lookup fails—that is, if no entry is found.

Estimated Size

Five to eight instructions plus the size of a hash lookup.

Microengine Assembler Example Usage

```
.reg out_index base
xbuf_alloc($ip_buffer, SIZE, read); // SIZE = size of IP packet
xbuf_alloc($out_header, 6, WRITE)
//initialize ip header in sram
immed32($out_header[0], 0x4510006e)
immed32($out_header[1], 0xabcd0000)
immed32($out_header[2], 0x10115270)
immed32($out_header[3], 0x0a031380)
immed32($out_header[4], 0x18877638)
immed32($out_header[5], 0x10000023)
immed[base, 0x1000]
sram_write($out_header[0], base, 0, 6, sig1, sig1, ___)
//Read tcp/ip header
sram_read($ip_buffer[0], base, 0, 6, sig1, sig1, ___)
//look up for key
ipv4_n_tuple_lookup(out_index, $ip_buffer, 0, \
    0x100, 0x10, HASH_16_4, FIVE_TUPLE);
```

Microengine C Example Usage

```
__declspec(local_mem) uint32_t tcpIpHdr[8];
__declspec(gp_reg) uint32_t out_index;
__declspec(gp_reg) uint32_t triAddrBase, keyDataSdBase;
triAddrBase = 64;
keyDataSdBase = 512;
//Intialize TCP/IP header
tcpIpHdr[0] = 0x4500006e;
tcpIpHdr[1] = 0xabcd0000;
tcpIpHdr[2] = 0x10115270;
tcpIpHdr[3] = 0x0a031380;
tcpIpHdr[4] = 0x18877638;
tcpIpHdr[5] = 0x08000400;
tcpIpHdr[6] = 0x00000000;
out_index = ixp_ipv4_n_tuple_lookup(
        tcpIpHdr,
        0,
        LOCALMEM,
        FIVE_TUPLE,
        (__declspec(sram) void *) triAddrBase,
        keyDataSdBase,
        HASH_16_4);
```

5.1.2.2.14 ipv4_tos_modify()

Writes the TOS field in the IP header and updates the checksum.

Microengine Assembler Syntax

```
ipv4_tos_modify(out_data,IN_IPHDR_WR_START_BYTE, in_tos_value, in_data, \
    IN_IPHDR_RD_START_BYTE, in_prepend);
```

Microengine C Syntax

```
INLINE void ixp_ipv4_tos_modify(
    void *out_data,
    IMMED IN_IPHDR_WR_START_BYTE,
    mem_t OUT_DATA_MEM_TYPE,
    uint32_t in_tos_value,
    void *in_data,
    IMMED IN_IPHDR_RD_START_BYTE,
    mem_t IN_DATA_MEM_TYPE,
    uint32_t in_prepend);
```

Input

`IN_IPHDR_WR_START_BYTE`	The constant byte offset from the start of `in_data`.
`OUT_DATA_MEM_TYPE`	The memory type for the `out_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`in_data`	The buffer containing an IP header.
`IN_IPHDR_RD_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_tos_value`	The new TOS value to be included in `out_data`.
`IN_DATA_MEM_TYPE`	The memory type for the `in_data` parameter. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`in_prepend`	The longword containing bytes to be merged with the first word—prepended if the byte alignment within the first IP longword is not zero. Each byte to be merged must be at the exact location it will occupy in the output header.

Output/Returns

`out_data`	The new IP header with modified fields.

Estimated Size

- Microengine C—15 to 24 instructions
- Microengine Assembler—11 to 24 instructions

Microengine Assembler Example Usage

```
.begin
.reg tos_value in_prepend
.sig sig1
xbuf_alloc ($in_header, 7, READ)
xbuf_alloc ($out_header, 7, WRITE)
//initialize ipv4 header offset 1
immed32($out_header[0], 0x00450000)
immed32($out_header[1], 0x6eabcd00)
immed32($out_header[2], 0x00101152)
immed32($out_header[3], 0x700a0313)
immed32($out_header[4], 0x80188776)
immed32($out_header[5], 0x38100000)
immed[base, 0x1000]
// Write header in sram
sram_write($out_header[0], base, 0, 6, sig1, sig1, ___)
//Read ipv4 header
sram_read($in_header[0], base, 0, 6, sig1, sig1, ___)
move(tos_value, 0x12)
// modify tos in in_header & write new packet in out_header
ipv4_tos_modify($out_header, 2, tos_value, $in_header, 1, 0)
//now $out_header contains new ipv4 header
xbuf_free($in_header)
xbuf_free($out_header)
.end
```

Microengine C Example Usage

```
__declspec(sram_read_reg) uint32_t s_in_header[6];
__declspec(sram_write_reg) uint32_t s_out_header[6], tcpIpHdr[8];
SIGNAL sig1;
// Initialize IPHdr alignment = 0
tcpIpHdr[0] = 0x4500006e;
tcpIpHdr[1] = 0xabcd0000;
tcpIpHdr[2] = 0x10115270;
tcpIpHdr[3] = 0x0a031380;
tcpIpHdr[4] = 0x18877638;
tcpIpHdr[5] = 0x08000400;
tcpIpHdr[6] = 0x00000000;
addr = 0x1000;
sram_write(tcpIpHdr, (__declspec(sram) U32 *)addr, 7, ctx_swap, &sig1);
sram_read(s_in_header, (__declspec(sram) U32 *) addr, 5, ctx_swap, &sig1);
ixp_ipv4_tos_modify((void *)s_out_header, 0, SRAM_WR_REG, 10, s_in_header, 0,
       SRAM_RD_REG, 0);
// Write back modified header in new location
sram_write(s_out_header, (__declspec(sram) uint32_t *) 0x1400, 5, ctx_swap,
       &sig1);
```

5.1.3 ipv6—Microengine

The ixp_ipv6 interface defines the following functions for processing IP headers. Please refer to [RFC2373], [RFC2460], [IPV6LLD] and other associated internet documents for a description of internet protocol fields and functionality performed by routers.

Table 5-3. ipv6 API

Name	Description
ipv6_version_verify()	Verifies that the version field contains 6 for IPv6.
ipv6_addr_verify()	Checks for invalid router addresses.
ipv6_hop_limit_verify()	Verifies that the hop limit value is greater than one.
ipv6_next_hdr_verify()	Checks for invalid Next Header values.
ipv6_verify()	Performs IPv6 header validation.
ipv6_modify()	Decrements the hop limit by one and writes back the result.
ipv6_proc()	Processes an IPv6 header that starts at the read *xbuf* byte pointer.
ipv6_5tuple_lookup()	Looks up a table entry using five fields—source and destination addresses, source and destination ports, and next header.
ipv6_route_lookup()	Performs a longest prefix match lookup on a 128-bit destination address and returns a 32-bit next hop ID.
ipv6_route_cache_lookup()	Performs a quick lookup in cache.

5.1.3.1 API Functions

5.1.3.1.1 `ipv6_version_verify()`

Verifies that the version field contains `6` for IPv6.

Microengine Assembler Syntax

```
ipv6_version_verify (in_data, IN_IPHDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_version_verify (void *in_data,
    uint32_t IN_IPHDR_START_BYTE, mem_t in_mem_type);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

Output/Returns

Return Value

- Microengine C
  Zero, `IPV6_SUCCESS`—for correct version—that is, IPv6
  Non-Zero, `IPV6_BAD_VERSION`—otherwise
- Microengine Assembler
  Condition Code `Z=1` for correct version
  Condition Code `Z=0` otherwise

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
ipv6_version_verify($$packet, 14);
bne[exception_handling#]
```

Microengine C Example Usage

```
result = ixp_ipv6_version_verify(&packet, 14, DRAM_RD_REG);
if (result)
    goto exception_handling;
```

5.1.3.1.2 `ipv6_addr_verify()`

Checks for invalid router addresses according to these rules:

- If the destination address is `0x0` or `0x1` the packet is dropped
- If the destination address is a link-local address the packet is processed as an exception
- If the destination address is a multicast address the packet is processed as an exception
- If the source address is a multicast address the packet is dropped

Microengine Assembler Syntax

```
ipv6_addr_verify (out_result, in_data, IN_IPHDR_START_BYTE, in_addr_origin);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_addr_verify (void *in_data,
    uint32_t IN_IPHDR_START_BYTE, addr_origin_t in_addr_origin,
    mem_t in_mem_type);
```

Input[1]

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_addr_origin`	One of {`IP_SOURCE` or `IP_DESTINATION`}.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

1. Buffer and offset are used instead of `uint32_t` because an IPv6 address is 128 bits long.

Output/Returns

Return Value	• For Microengine C Zero, `IPV6_SUCCESS`—for success Non-Zero—exception and drop codes otherwise • For Microengine Assembler, condition code `Z` value is one of: Zero—for success Non-Zero—for exception and drop codes
`out_result`	• Zero, `IPV6_SUCCESS`—for success • Non-Zero—exception and drop codes otherwise

Estimated Size

Four to 35 instructions.

Microengine Assembler Example Usage

```
ipv6_addr_verify(out_result, $$packet, 14, IP_DESTINATION, LOCALMEM);
beq[exception_drop_handling#]
//...
exception_drop_handling#:
       // examine out_result
```

Microengine C Example Usage

```
result = ixp_ipv6_addr_verify(&packet, 14, IP_SOURCE, SRAM_RD_REG);
if (result)
goto exception_drop_handling;
```

5.1.3.1.3 `ipv6_hop_limit_verify()`

Verifies that the hop limit value is greater than one.

Microengine Assembler Syntax

```
ipv6_hop_limit_verify (in_data, IN_IPHDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_hop_limit_verify (
    void *in_data,
    uint32_t IN_IPHDR_START_BYTE,
    mem_t in_mem_type);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

Output/Returns

Return Value	For Microengine C: • Zero, `IPV6_SUCCESS`—for success • Non-Zero, `IPV6_BAD_HOP_LIMIT`—otherwise For Microengine Assembler: condition code flag value is one of: • Zero—for success • Greater than Zero—the hop limit was bad, an exception was signaled

Estimated Size

Two to four instructions.

Microengine Assembler Example Usage

```
ipv6_hop_limit_verify($$packet, 14);
br>0[exception_drop_handling#]
```

Microengine C Example Usage

```
result = ixp_ipv6_hop_limit_verify(&packet, 14, LOCALMEM);
if (result)
goto exception_drop_handling;
```

5.1.3.1.4 `ipv6_next_hdr_verify()`

Checks the following invalid *Next Header* values:

- 0—hop-by-hop options header
- 43—source routing header
- 60—destination options header

Microengine Assembler Syntax

```
ipv6_next_hdr_verify (in_data, IN_IPHDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_next_hdr_verify (
    void *in_data,
    uint32_t IN_IPHDR_START_BYTE,
    mem_t in_mem_type);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

Output/Returns

Return Value	For Microengine C: • Zero, `IPV6_SUCCESS`—for valid next header values • Non-Zero—the next header values were invalid For Microengine Assembler—condition code `Z` is set to one of the following: • One—for an exception or drop • Zero—for valid next header values

Estimated Size

Four to thirteen instructions.

Microengine Assembler Example Usage

```
ipv6_next_hdr_verify($$packet, 14);
beq[exception_drop_handling#]
```

Microengine C Example Usage

```
result = ixp_ipv6_next_hdr_verify(&packet, 14, SRAM_RD_REG);
if (result)
    goto exception_drop_handling;
```

5.1.3.1.5 `ipv6_verify()`

Performs IPv6 header validation for the following fields:

- Version
- Source and Destination addresses
- Hop Limit
- Next Header

Microengine Assembler Syntax

```
ipv6_verify (out_result, in_data, IN_IPHDR_START_BYTE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_verify (
   void *in_data,
   uint32_t IN_IPHDR_START_BYTE,
   mem_t in_mem_type);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.

Output/Returns

`out_result` *or* Return Value	Microengine C: • Zero, `IPV6_SUCCESS`—for success • Non-Zero—exception and drop codes otherwise Microengine Assembler: condition code `Z` value is one of: • One—for exception and drop code • Zero—for success

Estimated Size

Four to 99 instructions.

Microengine Assembler Example Usage

```
ipv6_verify(out_result, $$packet, 14);
beq[exception_drop_handling#]
//...
exception_drop_handling#:
       // examine out_result
```

Microengine C Example Usage

```
result = ixp_ipv6_verify(&packet, 14, SRAM_RD_REG);
if (result)
goto exception_drop_handling;
```

5.1.3.1.6 `ipv6_modify()`

Decrements the hop limit by one and writes back the result. Merges `in_prepend` data to the first word if the IP header offset is not longword aligned. This function assumes that the hop limit has already been verified as valid—that is, its value is greater than one.

Microengine Assembler Syntax

```
ipv6_modify (out_data, IPHDR_WR_START_BYTE, in_data, \
    IN_IPHDR_RD_START_BYTE, in_prepend);
```

Microengine C Syntax

```
INLINE void ixp_ipv6_modify (
    void *out_data,
    uint32_t IN_IPHDR_WR_START_BYTE,
    mem_t out_mem_type,
    void *in_data,
    uint32_t IN_IPHDR_RD_START_BYTE,
    mem_t in_mem_type,
    uint32_t in_prepend);
```

Input

`IPHDR_WR_START_BYTE`	The constant byte offset from the start of `out_data`.
`out_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `out_data`—the value is one of {`SRAM_WR_REG`, `DRAM_WR_REG`, `LOCALMEM`, or `GP_REG`}.
`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`in_prepend`	The longword containing bytes to be merged with the first word—to be prepended if byte alignment within the first IP longword is not zero. The bytes to be merged must be at the exact byte locations that they occupy in the output header. All other bytes must be zero. If `out_mem_type` is `DRAM`, `in_prepend` only applies to the longword that contains the start of IP header.

Output/Returns

`out_data`[1]	The output buffer containing the modified fields.

1. Not available from Microengine C code.

Estimated Size

Ten to 16 instructions.

Microengine Assembler Example Usage

```
move(in_prepend, 0xb0b1b200);
ipv6_modify(out_modified_pkt, 14, $$packet, 14, in_prepend);
```

Microengine C Example Usage

```
move(in_prepend, 0xb0b10000);
ixp_ipv6_modify(&modified_pkt, 2, DRAM_WR_REG, &pkt, 2, SRAM_RD_REG,
    in_prepend);
```

5.1.3.1.7 `ipv6_proc()`

Processes an IPv6 header that starts at the read `xbuf` byte pointer. This function calls the following:

- `ipv6_verify()` to validate header
- `ipv6_modify()` to decrement hop limit value by one

Microengine Assembler Syntax

```
ipv6_proc (out_result, out_data, IPHDR_WR_START_BYTE, in_data, \
   IN_IPHDR_RD_START_BYTE, in_prepend);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_proc (
   void *out_data,
   uint32_t IN_IPHDR_WR_START_BYTE,
   mem_t out_mem_type,
   void *in_data,
   uint32_t IN_IPHDR_RD_START_BYTE,
   mem_t in_mem_type,
   uint32_t in_prepend);
```

Input

`IPHDR_WR_START_BYTE`	The constant byte offset from the start of `out_data`.
`out_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `out_data`—the value is one of {`SRAM_WR_REG`, `DRAM_WR_REG`, `LOCALMEM`, or `GP_REG`}.
`in_data`	The buffer with an IP header.
`IN_IPHDR_RD_START_BYTE`	The constant byte offset from the start of `in_data`.
`in_mem_type`	For *Microengine C only*: this argument defines the type of memory used for `in_data`—the value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`in_prepend`	The longword containing bytes to be merged with the first word—to be prepended if byte alignment within the first IP longword is not zero. The bytes to be merged must be at the exact byte locations that they occupy in the output header. All other bytes must be zero. If `out_mem_type` is `DRAM`, `in_prepend` only applies to the longword that contains the start of IP header.

Output/Returns

`out_result` *or* Return Value	The results of the IPv6 verification and hop-limit decrement: • Zero, `IPV6_SUCCESS`—for success • Non-Zero—for exception and drop codes otherwise
`out_data`[1]	The output buffer containing modified fields.

1. Available from *Microengine Assembler only*.

Estimated Size

The number of instructions equals the estimated size of the equivalent `ipv6_verify()` and `ipv6_modify()` calls combined.

Microengine Assembler Example Usage

```
move(in_prepend, 0xb0b1b200);
ipv6_modify(out_result, out_modified_pkt, 14, $$packet, 14, in_prepend);
bne[exception_drop_handling#]
    // use out_modified_pkt
exception_drop_handling#:
    // examine out_result
```

Microengine C Example Usage

```
move(in_prepend, 0xb0b10000);
result = ixp_ipv6_proc(&modified_pkt, 2, DRAM_WR_REG, &pkt, 2, SRAM_RD_REG,
    in_prepend);
if (result) goto exception_drop_handling;
```

5.1.3.1.8 `ipv6_5tuple_lookup()`

Looks up a table entry using five fields—source and destination addresses, source and destination ports, and next header. This function does not handle packets that contain IPv6 extension headers.

Microengine Assembler Syntax

```
ipv6_5tuple_lookup (out_index, in_data, IN_IPHDR_START_BYTE, trie_addr,\
    KEY_DATA_SD_BASE, TRIE_TYPE);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_5tuple_lookup (
    void *in_data,
    IMMED IN_IPHDR_START_BYTE,
    mem_t IN_MEM_TYPE,
    uint32_t trie_addr,
    uint32_t TRIE_TYPE);
```

Input

`in_data`	The buffer with an IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from start of `in_data`.
`IN_MEM_TYPE`	For *Microengine C only*, this value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`trie_addr`	The address of the trie table.
`KEY_DATA_SD_BASE`	The base address of the key database located in SDRAM.
`TRIE_TYPE`	The index bits used to address the hash trie. This value is one of: • `HASH_16_4`—first lookup uses 16 bits of index, subsequent lookups use four bits of index • `HASH_FOLD_16`—first lookup XORs the initial index to reduce it by half, then performs a table lookup using 16 bits of half-index, with subsequent chain search until there is no collision

Output/Returns

`out_index` *or* Return Value	Returns the index of the found entry if successful or zero if the lookup failed—that is, there no entry was found.

Estimated Size

- If the value of `TRIE_TYPE` is `HASH_16_4` the number of instructions equals:
  `29 + sizeof(_hash_read_prim_table) + sizeof(_hash_resolve_16_4)`
- If the value of `TRIE_TYPE` is `HASH_FOLD_16` the number of instructions equals:
  `29 + sizeof(_hash_resolve_fold_16)`

Microengine Assembler Example Usage

```
.reg out_index
xbuf_alloc($$packet, 12, read)
// read the IP packet from memory into $$packet buffer
ipv6_5tuple_lookup(out_index, $$packet, 14, trie_addr, KEY_DATA_SD_BASE,\
   TRIE_TYPE);
```

Microengine C Example Usage

```
uint32_t entry_id;
__declspec(DRAM_RD_REG) uint32_t packet[20];
// read the IP packet from memory into packet[] buffer
entry_id = ixp_ipv6_5tuple_lookup(
       &packet,
       14,
       DRAM_RD_REG,
       trie_addr,
       KEY_DATA_SD_BASE,
       TRIE_TYPE);
```

5.1.3.1.9 ipv6_route_lookup()

Performs a longest prefix match lookup on a 128-bit destination address and returns a 32-bit next hop ID. Uses the table set up by the core component Route Table Manager.

Including the `_IPV6_ROUTE_LOOKUP_IN_CACHE` compiler preprocessor directive enables the calling application to perform cache-based route lookup through an application-provided callback function. This allows the application to maintain its own cache entries and enables the invocation of the application-provided callback function from within the `ixp_ipv6_route_lookup()` function. See `ipv6_route_cache_lookup()` for details.

Microengine Assembler Syntax

```
ipv6_route_lookup (out_rt_ptr, in_data, IN_IPHDR_START_BYTE,\
    in_trie_addr, TRIE_DEPTH);
```

Microengine C Syntax

```
INLINE uint32_t ixp_ipv6_route_lookup (
    void *in_data,
    IMMED IN_IPHDR_START_BYTE,
    mem_t IN_MEM_TYPE,
    uint32_t trie_addr,
    trie_t TRIE_DEPTH);
```

Input

`in_data`	The buffer with IP header.
`IN_IPHDR_START_BYTE`	The constant byte offset from start of in_data
`IN_MEM_TYPE`	For Microengine C only this argument specifies the type of memory used. This value is one of {`SRAM_RD_REG`, `DRAM_RD_REG`, `LOCALMEM`, or `GP_REG`}.
`trie_addr`	The address of the trie table.
`TRIE_DEPTH`	The depth of trie. At this time only the value of `TRIE15` is supported.

Output/Returns

`out_rt_ptr` ***or*** Return Value	A route pointer—that is, an index—to the route entry. If this value is zero, no route entry was found. For *Microengine Assembler only* this is a general-purpose register.

Estimated Size

- For Microengine Assembler, if `_IPV6_ROUTE_LOOKUP_IN_CACHE` is defined the estimated size ranges from:
  - A minimum number of instructions equal to `8 + sizeof (ipv6_route_cache_lookup)`
  - To a maximum number of instructions equal to the minimum plus 118 instructions—that is, the worst case adds fifteen SRAM accesses
- For Microengine Assembler, if `_IPV6_ROUTE_LOOKUP_IN_CACHE` is not defined the estimated size is 13 to 125 instructions—the worst case adds 15 SRAM accesses

Microengine Assembler Example Usage

```
.reg route_entry_index
xbuf_alloc($$pkt, 12, read)
// read the IP packet from memory into pkt[] buffer
ipv6_route_lookup(route_entry_index, $$pkt, 14, DRAM, trie_addr, TRIE15);
```

Microengine C Example Usage

```
uint32_t route_entry_index;
__declspec(local_mem) uint32_t pkt[20];
// read the IP packet from memory into pkt[] buffer
route_entry_index = ixp_ipv6_route_lookup(
       &pkt,
       14,
       LOCALMEM,
       trie_addr,
       TRIE15);
```

5.1.3.1.10 `ipv6_route_cache_lookup()`

This is the function prototype for a calling-application implemented callback function. This callback performs a quick lookup in cache.

Microengine Assembler Syntax

```
ipv6_route_cache_lookup(out_rt_ptr, trie_ptr, da_127_96, da_95_64,\
    da_63_32, da_31_0)
```

Microengine C Syntax

```
ipv6_route_cache_lookup(
    uint32_t *rt_ptr,
    unit32_t trie_ptr,
    uint32_t da_127_96,
    uint32_t da_95_64,
    uint32_t da_63_32,
    uint32_t da_31_0)
```

Input

`trie_ptr`	The index of the next trie entry.
`da_127_96`	Bits 127 through 96 of the 128-bit IPv6 destination address.
`da_95_64`	Bits 95 through 64 of the 128-bit IPv6 destination address.
`da_63_32`	Bits 63 through 32 of the 128-bit IPv6 destination address.
`da_31_0`	Bits 31 through zero of the 128-bit IPv6 destination address.

Output/Returns

`rt_ptr`	The resulting route pointer, an index to route entry or zero if no entry.

5.2 ATM Protocol Interfaces

5.2.1 ATM Header Access

ATM Header access is implemented through a structure definition. This structure is used to extract the required fields of an ATM header. The calling application must appropriately use either the big endian header structure definition or the little endian header structure definition.

Figure 5-2 shows the structure of a UNI cell header. The corresponding data structures are defined in Section 2.6.16, "`atm_uni_header_le_t`" and Section 2.6.18, "`atm_uni_header_be_t`."

Figure 5-2. UNI Cell Header

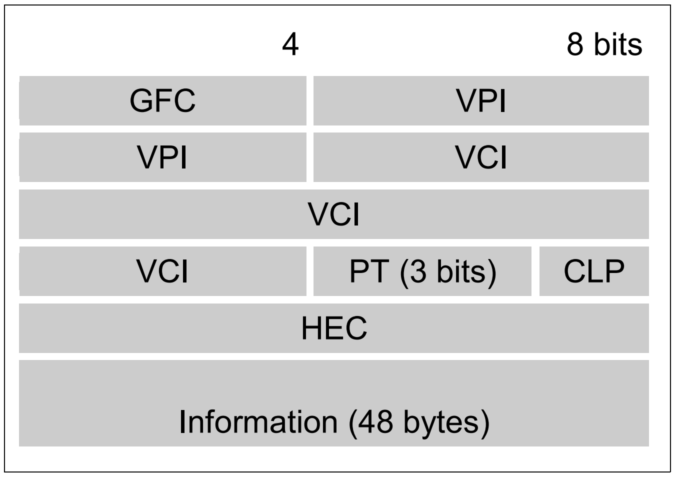
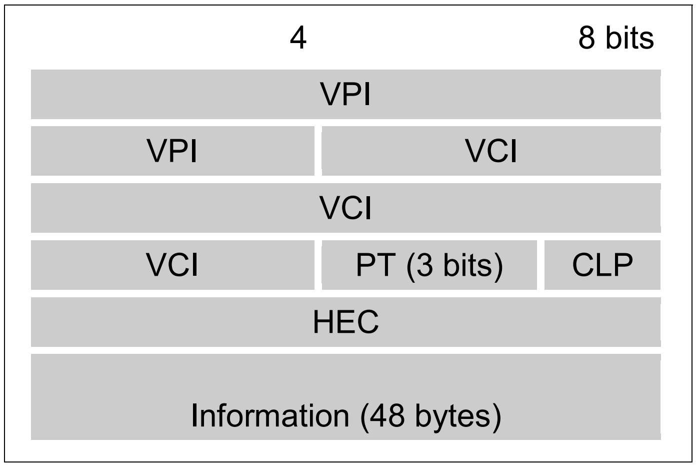


Figure 5-3 shows the structure of an NNI cell header. The corresponding data structures are defined in Section 2.6.17, "`atm_nni_header_le_t`" and Section 2.6.19, "`atm_nni_header_be_t`."

Figure 5-3. NNI Cell Header



Summary of the ATM Protocol API

Name	Description
`atm_compare_vpi_vci()`	Compares an optimized VPI VCI pair.
`atm_compare_vpi_vci_pti()`	Compares an optimized VPI VCI PTI combination.

5.2.1.1 `atm_compare_vpi_vci()`

Compares the passed VPI and VCI values with VPI and VCI values in the ATM cell header. For *Microengine C only* the memory type is also specified.

Microengine Assembler Syntax

```
atm_compare_vpi_vci(in_headerPtr, ATM_HEADER_START_BYTE, vpi, vci,\
    IN_ATM_HDR_TYPE, ENDIAN)
```

Microengine C Syntax

```
INLINE uint32_t ixp_atm_compare_vpi_vci (
    void   *in_atmHeaderPtr,
    IMMED ATM_HEADER_START_BYTE,
    MEM_TYPE memory_type,
    uint32_t vpi,
    uint32_t vci,
    atm_header_type ATM_HDR_TYPE,
    endian_t ENDIAN)
```

Input

`in_atmHeaderPtr`	Specifies the pointer to the ATM cell.
`ATM_HEADER_START_BYTE`	Specifies the start byte value from the ATM cell longword pointer.
`IN_MEM_TYPE`	For *Microengine C only* this is memory type for the address. This value is one of {`LOCAL_MEM`, `GP_REG`, `SRAM_RD_REG`, `DRAM_RD_REG`}.
`vpi`	The VPI value.
`vci`	The VCI value.
`IN_ATM_HDR_TYPE`	The ATM header type which is either `ATM_UNI` or `ATM_NNI`.
`ENDIAN`	Specifies the byte order which is either `LITTLE_ENDIAN` or `BIG_ENDIAN`.

Output/Returns

Return Value *or* Condition Codes	Returns `EQUAL` or Condition Code `Z` equal to one for correct version and `NOT_EQUAL` or Condition Code `Z` equal to zero otherwise.

Estimated Size

Four to five instructions.

Microengine Assembler Example Usage

```
atm_compare_vpi_vci (in_atmHdr, 10, 11, ATM_UNI, big_endian)
beq[equal#]
```

Microengine C Example Usage

```
If (ixp_atm_compare_vpi_vci (
          &atm_hdr_lm, 0, LOCALMEM, 10, 11, ATM_UNI, big_endian) {
       // If VPI = 10 and VCI = 11 do some action.
    } else {
       // Other code.
}
```

5.2.1.2 `atm_compare_vpi_vci_pti()`

This compares the passed vpi, vci and pti values with values in the ATM cell header. The memory type is specified in case of micro C API.

Microengine Assembler Syntax

```
atm_compare_vpi_vci_pti(in_headerPtr, vpi, vci, pti, \
    ATM_HEADER_START_BYTE, IN_ATM_HDR_TYPE, ENDIAN)
```

Microengine C Syntax

```
INLINE uint32_t ixp_atm_compare_vpi_vci_pti(
    void *in_atmHeaderPtr,
    IMMED ATM_HEADER_START_BYTE,
    mem_t IN_MEM_TYPE,
    uint32_t vpi,
    uint32_t vci,
    uint32_t pti,
    atm_header_t ATM_HDR_TYPE,
    endian_t ENDIAN)
```

Input

`in_atmHeaderPtr`	Specifies the pointer to the ATM cell.
`ATM_HEADER_START_BYTE`	Specifies the start byte value from the ATM cell longword pointer.
`IN_MEM_TYPE`	For *Microengine C only* this is memory type for the address. This value is one of {`LOCAL_MEM`, `GP_REG`, `SRAM_RD_REG`, `DRAM_RD_REG`}.
`vpi`	The VPI value.
`vci`	The VCI value.
`pti`	The PTI value.
`IN_ATM_HDR_TYPE`	The ATM header type which is either `ATM_UNI` or `ATM_NNI`.
`ENDIAN`	Specifies the byte order which is either `LITTLE_ENDIAN` or `BIG_ENDIAN`.

Output/Returns

Return Value *or* Condition Codes	Returns `EQUAL` or Condition Code `Z` equal to one for correct version and `NOT_EQUAL` or Condition Code `Z` equal to zero otherwise.

Estimated Size

Four to five instructions.

Microengine Assembler Example Usage

```
atm_compare_vpi_vci_pti (in_atmHdr, 10, 11, 5, ATM_UNI, big_endian)
beq[equal#]
```

Microengine C Example Usage

```
If (ixp_atm_compare_vpi_vci_pti (
           &atm_hdr_lm, 0, LOCALMEM, 10, 11, 5, ATM_UNI, big_endian) {
   // If VPI = 10 and VCI = 11 pti = 5 do some action.
   } else {
// Other code.
}
```

5.2.2 OAM Cell Access

OAM cell access is implemented through Portability Framework data structure definitions—these structures are used to extract required fields for an OAM cell. Figure 5-4 shows the structure definition for an OAM cell. The calling application must appropriately use either the big endian header structure definition or the little endian header structure definition—see Section 2.6.21, "`oam_cell_le_t`" and Section 2.6.22, "`oam_cell_be_t`." See also Section 2.6.23, "`oam_cell_t`" for an enumeration listing the supported OAM cell types.

Figure 5-4. OAM Cell Layout

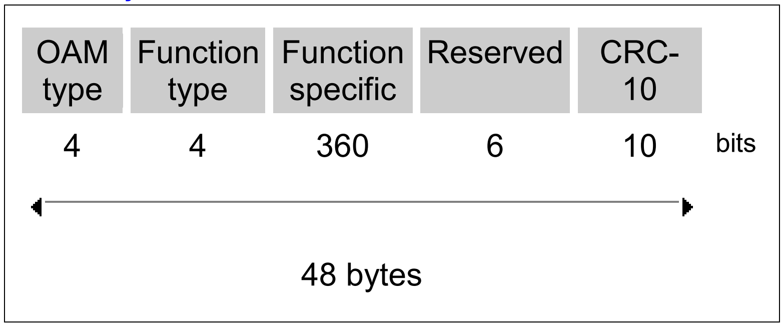


Table 5-4. OAM Cell Protocol API

Name	Description
`oam_gen_crc10()`	Generates CRC10 checksum for the OAM cell given.
`oam_validate_crc10()`	Validates if the CRC10 checksum in the cell is correct.

5.2.2.1 `oam_gen_crc10()`

Generates the CRC10 checksum for the OAM cell. The function accepts a pointer to the location where the OAM cell is located. For *Microengine C only* the memory type is also specified.

Note: The OAM cell must be passed word aligned—that is, with an offset of zero with respect to the OAM cell pointer.

Microengine Assembler Syntax

```
oam_gen_crc10(in_OamCell_ptr, ENDIAN)
```

Microengine C Syntax

```
INLINE void ixp_oam_gen_crc10(
    void *in_oamCell_ptr,
    mem_t IN_MEM_TYPE,
    endian_t ENDIAN)
```

Input

`in_oamCell_ptr`	Specifies the pointer to the OAM cell.
`IN_MEM_TYPE`	For *Microengine C only* this is the memory type for the address. This value is one of {`LOCAL_MEM`, `GP_REG`, `SRAM_RD_REG`, `DRAM_RD_REG`}.
`ENDIAN`	Specifies the byte order which is either `LITTLE_ENDIAN` or `BIG_ENDIAN`.

Output/Returns

None. The CRC10 checksum in written to the checksum field pointed to by `in_oamCell_ptr`.

Estimated Size

Approximately 900 plus 20 instructions for a total of 920 instructions.

Microengine Assembler Example Usage

```
          // calculate and insert CRC10 for the oam cell.
oam_gen_crc10 (inPtrOamCell, big_endian)
```

Microengine C Example Usage

```
          // insert crc10 checksum into the oam cell.
ixp_oam_gen_crc10 (inPtrOamCell, LOCALMEM, big_endian)
```

5.2.2.2 oam_validate_crc10()

Validates the CRC10 checksum for the OAM cell. This requires a pointer to the location where the PDU is located. For *Microengine C only* the memory type is also specified.

Note: The OAM cell must be passed word aligned—that is, with an offset of zero with respect to the OAM cell pointer.

Microengine Assembler Syntax

```
oam_validate_crc10 (in_OamCell_ptr, ENDIAN)
```

Microengine C Syntax

```
INLINE uint32_t ixp_oam_validate_crc10(void *in_oamCell_ptr,
    mem_t IN_MEM_TYPE, endian_t ENDIAN)
```

Input

`in_oamCell_ptr`	Specifies the pointer to the OAM cell.
`IN_MEM_TYPE`	For *Microengine C only* this is the memory type for the address. This value is one of {`LOCAL_MEM`, `GP_REG`, `SRAM_RD_REG`, `DRAM_RD_REG`}.
`ENDIAN`	Specifies the byte order which is either `LITTLE_ENDIAN` or `BIG_ENDIAN`.

Output/Returns

Return Value *or* Condition Codes	Returns `CRC_GOOD` or Condition Code `Z` equal to one for a correct CRC and `CRC_BAD` or Condition Code `Z` equal to zero otherwise.

Estimated Size

Approximately 900 plus 20 instructions for a total of 920 instructions.

Microengine Assembler Example Usage

```
oam_validate_crc10 (inPtrOamCell, big_endian)
beq[validChecksum#]
```

Microengine C Example Usage

```
If (ixp_oam_validate_crc10(inPtrOamCell, LOCALMEM, big_endian)) {
        //Valid checksum in the oam cell.
    } else {
        // Invalid Oam crc10 checksum.
}
```