Netronome Network Flow Processor 6xxx

NFP SDK version 6.1 Preview

Network Flow Assembler System User's Guide

- Proprietary and Confidential -

b3286.dr7273

Product code
030-00023-006

Netronome Network Flow Processor 6xxx: Network Flow Assembler System User's Guide

COPYRIGHT

WARRANTY

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

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

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

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

LIABILITY

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

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

Revision History

Date	Revision	Description
April 2016	006	Updated for NFP SDK 6.0 Beta 1
October 2014	005	Updated for NFP SDK 5.1
January 2014	004	Updated for NFP SDK 5.0 Beta
August 2013	003	Updated for NFP SDK 5.0 Alpha 2
6 June 2013	002	Updated for NFP SDK 5.0 Alpha 1
8 March 2013	001	Initial release

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Scope

This manual is a user's guide for the Network Flow Assembler System and is organized as follows:

- Chapter 2: Provides information on running the Assembler.
- Appendix A: Contains descriptions of the NFAS Assembler Warnings.
- Appendix B : Contains descriptions of the NFAS Error Messages.

1.2 Related Documents

Descriptive Name	Description
Netronome Network Flow Processor 6xxx: Databook	Contains detailed reference information on the Netronome Network Flow Processor NFP-6xxx.
Netronome Network Flow Processor 6xxx: Microengine Programmer's Reference Manual	Provides a reference for microcode programming of the Netronome Network Flow Processor NFP-6xxx.
Netronome Network Flow Processor 6xxx: Datasheet	Provides a functional overview of the Netronome Network Flow Processor NFP-6xxx's internal hardware, signals and electrical and mechanical specifications.
Netronome Network Flow Processor 6xxx: Development Tools User's Guide	Describes Programmer Studio and the development tools you can access through Programmer Studio.
Netronome Network Flow Processor 6xxx: Network Flow C Compiler User's Guide	Presents information, language structures and extensions to the language specific to the Netronome Network Flow C Compiler for Netronome NFP-6xxx.
Netronome Network Flow Processor 6xxx: Micro-C Standard Library Reference Manual	Specifies the subset and the extensions to the language as well as the intrinsic functions that support the unique features of the Netronome Network Flow Processor NFP-32xx product line.
Netronome Network Flow Processor 6xxx: Microcode Standard Library Reference Manual	Provides a reference to the Microcode Standard Library that supports the Netronome Network Flow Processor NFP-32xx product line.

1.3 Acronyms

Acronym	Description
ARM	ARM Holding plc
CAM	Content Addressable Memory

Acronym	Description
CPP	Command/Push/Pull bus
CPU	Central Processing Unit
CSR	Control and Status Register
DDR	Dual Data Rate
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSF	Distributed Switch Fabric
GPR	General-Purpose Register
LM	Local Memory
ME	Microengine
NFAS	Network Flow Assembler System
NFLD	Network Flow Linker
NFP	Network Flow Processor
NIC	Network Interface Card
PCIe	PCI Express
QDR	Quad Data Rate
VQDR	Virtual Quad Data Rate
Xfer	Transfer
IMB CPPAT	Island Master Bridge CPP Address Translation

2. Assembling, Linking and Running

This chapter provides information on running the Assembler. Background information on the Assembler functions, including the Network Flow Processor architecture appears in the *Netronome Network Flow Processor 6xxx Databook* for the network processor.

2.1 Command Line Arguments

The Assembler is invoked from the command line:

```
nfas [options] microcode_file [microcode_file ...]
```

where the options consist of:

-verbose	Print more information on successful build, such as a register usage summary.
-chip *chip_id*	Select target chip. This is the preferred method of targeting chip types. A chip definition also includes a set of chip revisions. For example, -chip nfp-6xxx will target all revisions of nfp-6xxx and -chip nfp-6xxx-a0 will only include revision A0. It is possible to define custom chips with custom chip IDs in nfp_sdk_user_chips.json. If no -chip is specified, the environment variable NFP_CHIP_ID is checked. The default chip is nfp-6xxx
-chip_list	List all defined chips as found in chip data and user definition files.
-nfp*TYPE*	Set processor type, where TYPE is 6xxx or 6xxxc. Letter 'c' indicates the presence of the crypto unit. It is preferred to use -chip nfp-6xxx instead of -nfp6xxx and -chip nfp-6xxxc instead of -nfp6xxxc.
-ctx_start *n*	Only use contexts starting with n.
-ctx_end *n*	Only use contexts up to and including n.
-pml	Enables protect macro local feature (see Section 2.8.5.1.1)
-indirect_ref_format_v1	Use the NFP-32xx (Legacy) indirect reference format.
-indirect_ref_format_v2	Use the NFP-6xxx/version2 indirect reference format.(Default)
-third_party_addressing_32_bit	Use 32-bit third party addressing mode.
-third_party_addressing_40_bit	Use 40-bit third party addressing mode (default).
-preproc32	Emulate 32 bit arithmetic in preprocessor expressions. It is best to use ELF32 for the output NFFW file with -preproc32 and ELF64 with -preproc64, with ELF32 then effectively being limited to 2GiB addressing space, because nfas expressions and constants are all signed integers.
-preproc64	Use 64 bit arithmetic in preprocessor expressions (default).
-preproc_c	Preprocess undefined macro to value 0.

-lm *start*	Define the *start* of local memory allocation in bytes. The starting and ending address of local memory must be aligned on 4-byte boundary.
-lm *start:size*	Define the *start* and *size* of local memory allocation in bytes. The starting and ending address of local memory must be aligned on 4-byte boundary.
-lr	Dumps the register liverange information into a file with a `.lri` extension. If the register allocation fails, the `.uci` file may not be created so the information will be available for viewing in the `.lri` file.
-o *file*	Use *file* as the generated list file. This is only valid if there is one microcode_file.
-O	Enables optimization.
-Of	Tries to automatically fix A/B Bank conflicts. Default is disabled.

Note

These two optimization options are independent of each other; in other words any combination can be specified. The default option, **disabled**, avoids having the Assembler **add** code that the programmer did not specify.

-Os	Tries to automatically spill GPRs into local memory. Default is disabled.
-spilling-no-code	Enables codeless spilling. That is spilling will not insert extra instructions to your code for use with spilling. See Section 2.5.11
-spill-def-ind IND	Specifies the default local memory index that is used with spilling. (See Section 2.5.11). Defaults to 1
-g	Adds debugging info to output file.
-v	Prints the version number of the Assembler.
-h	Prints a usage message (same as -?).
-?	Prints a usage message (same as -h).
-r	Register declarations are not required.
-R	Register declarations are required (this is the default).
-Wn	Set Warning Level (n=0-4).
-w	Disable warnings (same as -W0).
-WX	Report error on any warning.
-C	Enables case sensitive assembly.
-help or --help	Displays help.
-version or --version	Displays current nfas version.
-t	Enable "terse" output.
-keep_unreachable_code	Do not comment out unreachable code.

The following version arguments allow assembly to be targeted for a specific chip version or range of versions, overriding the default values. The predefined Preprocessor macros `-REVISION_MIN` and `-REVISION_MAX` will reflect the specified version range. In addition, the version range is also written to the `.list` file in a '.cpu_version' directive.

For the following arguments, *rev* is an upper or lower case letter (A-P) followed by a decimal number (0-15), for example `-REVISION_MIN=A2` or `-REVISION_MIN=B0`, or an eight-bit number where bits <7:4> indicate the major stepping and bits <3:0> indicate the minor stepping, for example, `-REVISION_MIN=1` or `-REVISION_MIN=0x10`

`-REVISION=rev`	Targets assembly to chip version *rev.* This is equivalent to setting options `'-REVISION_MIN=rev'` and `'-REVISION_MAX=rev'` with the same value.
`-REVISION_MIN=rev`	Targets assembly to the minimum chip version *rev.* (The default is 0.)
`-REVISION_MAX=rev`	Targets assembly to the maximum chip version *rev.* (The default is 15, no limit.)

The following options are passed to the preprocessor:

-P	Preprocess only into a file: `microcode_file.ucp`.
-E	Preprocess only into stdout.
-I*directory*	Add the *directory* to the end of the list of directories to search for included files.
-D*name*	Define *name* as if the contained "#define *name 1"*.
-D*name=def*	Define *name* as if the contained "*#define name def*".
-N	Disable the preprocessor.

The **microcode_file** names may contain an explicit suffix, or if the suffix is missing, `.uc` is assumed. Assembling several files in one command line is equivalent to assembling each individually; the files are not associated with each other in any way.

If NFAS is invoked with no command line arguments, a usage summary is printed.

In Windows environments, the Assembler may also be invoked through Programmer Studio. In Linux environments, the Assembler is available as a command-line executable called `nfas` and has the same functionality and options as `NFAS.exe`.

2.2 Assembler Steps

As shown in Figure 2.1, invoking the Assembler results in a two-step process composed of preprocessing and assembly steps. The preprocessor step takes a `.uc` file and transforms it in memory by expanding macros, replacing `#define` and `#define_eval` literals, and processing include files. This preprocessed image is written out to a `.ucp` file. The next step transforms the preprocessed memory image into the `.list` file. At the end of assembly, an auxiliary file with the file name extension of `.uci` is also created. The `.uci` file contains error information and register allocation map which are useful for debugging purposes. During the assembly step the following functions are performed:

- Checks instruction restrictions.
- Resolves symbolic register names to physical locations.
- Optimizes the code.
- Resolves label addresses.
- Translates symbolic opcodes into binary patterns.

The preprocessor is invoked from within the Assembler. Command line options are available when invoking NFAS.exe (or NFAS.dll via the workbench).

```
nfas [options] microcode_file [microcode_file ...]
```

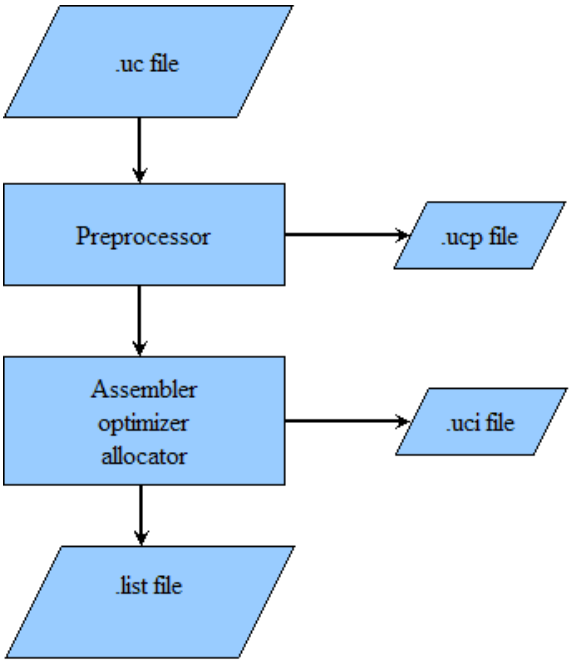


Figure 2.1. Assembly Process

The `.uc` file contains three types of elements: microwords, directives, and comments. Microwords consist of an opcode and arguments and generate a microword in the `.list` file. Directives pass information either to the preprocessor, Assembler, or to downstream components (e.g., the Linker) and generally do not generate microwords. Comments are ignored in the assembly process.

2.3 Assembler Preprocessor

The preprocessor is invoked automatically by the assembler to transform a program before actual assembly. The preprocessor provides six separate facilities that you can use as you see fit:

- **Inclusion of files:** These are files of declarations that can be substituted into your program.
- **Macro expansion:** You can define macros, which are abbreviations for arbitrary fragments of assembly code, and then the preprocessor will replace instances of the macros with their definitions throughout the program.
- **Conditional compilation:** Using special preprocessing directives, you can include or exclude parts of the program according to various conditions.
- **Line control:** If you use a program to combine or rearrange source files into an intermediate file which is then assembled, you can use line control to inform the assembler of where each source line originated from.

- **Structured Assembly:** You can organize the control flow of the Microengine instructions into structured blocks instead of several go to statements.
- **Token Replacement:** It causes instances of an identifier to be replaced with a token string.

2.3.1 Preprocessor Reserved Labels

The preprocessor generates labels during macro expansion and during conditional assembly. Avoid using labels with these prefixes to avoid confusion with those generated by the preprocessor and the possibility of multiple label definitions. In the following, *nnn* represents a three-digit decimal number.

- Mnnn_ Prefix used for macro references not preceded by a label.
- lnnn_ Prefix used for labels of structured assembly constructs.

2.3.2 Preprocessor Operation

The preprocessor is a simple macro processor that processes the source file before it is assembled. It is important to have a basic understanding of how the preprocessor operates, to understand how directives interact with one another. This section provides a brief overview.

During the initial reading of an input file, there are three occasions when the file is read but not processed:

- Within a #if … #endif clause, if the text is being skipped.
- Within a macro definition.
- Within the body of an assembly loop (e.g., #repeat).

In each of these cases, no processing of directives takes place, with the exception of the directive that ends that context. Constructs of a similar type may nest, however, so that if within a macro definition there is another macro definition, the first macro definition will not end until the second (i.e., the matching) .endm is reached. Within a particular context, other directives are ignored. For example, if a macro definition had a #if without a matching #endif, an error would not be reported until the macro was referenced (expanded). These constructs can thus be nested within each other, but they cannot be only partially contained within each other. It would be an error, for example, to put an unmatched #if within one macro and the matching #endif in another.

Lines that are being processed, have expandable tokens expanded. Then macro references are expanded. This means that an expandable token used as an argument in a macro reference is expanded at the time of the reference, not when it is used within the body of the macro. Table 2.1 summarizes the preprocessor directives.

Table 2.1. Summary of Preprocessor Directives

Directive	Arguments Expanded?	Description
#include	No	Start reading lines from another file.
#define	No	Define an expandable token.

Directive	Arguments Expanded?	Description
#define_eval	Yes	Define an expandable token as the result of evaluating a constant expression.
#undef	No	Undefine an expandable token.
#ifdef, #ifndef	No	Conditionally skip following lines.
#if, #elif	Yes, including “defined(name)”	Conditionally skip following lines based on a constant expression.
#else, #endif	N/A	Conditionally skip following lines.
#macro	No	Start defining a macro.
#endm	N/A	Finish defining a macro.
#repeat, #while	Yes	Repeat following lines based on a constant expression.
#for	No	Repeat following lines based on a constant expression.
#endloop	N/A	End repeated lines.
.if, .elif	Yes	Generate branch instructions.
.if_unsigned,.elif_unsigned	Yes	Generate branch instructions.
.else, .endif	N/A	Generate branch instructions.
.while	Yes	Generate branch instructions.
.while_unsigned	Yes	Generate branch instructions.
.endw	N/A	Generate branch instructions.
.repeat	N/A	Generate branch instructions.
.until	Yes	Generate branch instructions.
.until_unsigned	Yes	Generate branch instructions.
.break, .continue	N/A	Generate branch instructions.

2.3.2.1 Evaluation of Undefined Macro

This applies to evaluation of expressions in #if and #elif.

The normal behavior for the preprocessor when encountering undefined macros in #if or #elif is to stop with an error (Invalid constant expression). The user can make the preprocessor behave more like a C preprocessor in this case and let the undefined macro be evaluated as 0 by specifying `-preproc_c` on the command line. In this case no error will be produced and it becomes the responsibility of the user to check for the existence of macros using #ifdef where macros are required to exist, regardless of their value.

For example:

Example 2.1. Undefined Macro

```
#define AAA 1
#define BBB 0

#if (AAA == 1)
    #warning "AAA 1"
#else
    #warning "AAA 0"
#endif

#if (BBB == 1)
    #warning "BBB 1"
#else
    #warning "BBB 0"
#endif

#if (CCC == 1)
    #warning "CCC 1"
#else
    #warning "CCC 0"
#endif

The above will produce the following output when not using -preproc_c:
warning : AAA 1
warning : BBB 0
error : Invalid constant expression:  (CCC == 1)

The following output will be produced when using -preproc_c:
warning : AAA 1
warning : BBB 0
warning : CCC 0
```

2.3.3 Constant Expressions

Constant expressions (const-expr) are expressions that evaluate to a constant. Generally, after the assembler performs token substitution, the expression consists only of numeric constants and operators. The exception to this is a number of preprocessor functions that take identifiers as arguments. Within an instruction, wherever a constant is valid, you can use a constant expression that is surrounded by parenthesis. The parentheses are needed to differentiate expressions from tokens, such as “B-A”, which should not be evaluated. For some directives, the parenthesis may be omitted, but it is generally a good idea to use them for return identifiers or numeric constants as values. Wherever the term **const_expr** appears in this manual, it can be replaced with (**const_expr**), where const_expr is one of the following:

- (const)
- (const-expr bin-op const-expr)
- (unary-op const-expr)
- (const_expr ? const_expr : const_expr)
- function (token, token, ...)

2.3.3.1 32-bit or 64-bit Constants and Expressions

By default, constants are interpreted as signed 64-bit integers and expressions are evaluated in the signed 64-bit integer domain. This can be overridden by command line options `-preproc32` or `-ixp_compatible` to get 32-bit preprocessor behavior as for the IXA SDK or previous NFP SDK releases.

Some examples of expression results in both domains are shown below. More information on the functions and operators follow in the sections below. For convenience, hexadecimal digits are grouped.

Table 2.2. 32-bit and 64-bit Expression Examples

Expression	32-bit Result	64-bit Result
-1	0xFFFF FFFF (-1)	0xFFFF FFFF FFFF FFFF (-1)
0xFFFF FFFF	0xFFFF FFFF (-1)	0x0000 0000 FFFF FFFF (4294967295)
(1 << 31)	0x8000 0000 (-2147483648)	0x0000 0000 8000 0000 (2147483648)
((1 << 31) >> 31)	0xFFFF FFFF (-1)	0x0000 0000 0000 0001 (1)
((1 << 63) >> 63)	0x0000 0000 (0)	0xFFFF FFFF FFFF FFFF (-1)
~0	0xFFFF FFFF (-1)	0xFFFF FFFF FFFF FFFF (-1)
(0x8000 0000 / 1000000)	0xFFFF F79D (-2147)	0x0000 0000 0000 0863 (2147)
log2(-1, 1)	0x0000 0020 (32)	0x0000 0000 0000 0040 (64)

2.3.3.2 Preprocessor Binary & Unary Operators

Table 2.3 describes the binary and unary operators that are supported within constant expressions. Operator precedence is the same as that defined for the C programming language.

Table 2.3. Binary & Unary Operators

Type	Operator	Associativity	Comment
unary-ops	! ~ + - (unary)	Right to left	
bin-ops	* / %	Left to right	
	+ -	Left to right	
	<< >>	Left to right	
	< <= => >	Left to right	These relational operators assume signed 32-bit values.
	== !=	Left to right	
	&	Left to right	
	^	Left to right	
	|	Left to right	
	&&	Left to right	
	||	Left to right	
	?:	Right to left	
	,	Left to right	

2.3.3.3 Preprocessor: Functions

Table 2.4 describes the functions supported within constant expressions. These functions, with the exception of "defined", operate on the results of expanding tokens and evaluating expressions.

Note

In expanding `.if` and `.elif`, the defined(name) construct is replaced by a 0 or 1, as appropriate.

Table 2.4. Functions

Function	Description
is_nfptype(type), is_ixptype(type)	Returns non-zero if the targeted processor type is only of the type given by the parameter type and no other. The calculation performed is "!(__IXPTYPE & ~(type))". The result of IS_IXPTYPE(__IXP28XX) is controlled by assembler IXP mode settings. NOTE: The parameter type can be an expression such as "(type1 | type2)". This function should be used in conjunction with the predefined macros described in Section 2.3.8.2.
is_ct_const(token)	Returns 1 if the token expands to a numeric constant, otherwise it returns 0.
is_rt_const(token)	Returns 1 if the parameter expression is an expression that evaluates to a constant known at link or load time (e.g., a symbol reference).
isrestricted(token)	Returns 1 if the token can be used as a restricted operand (for example, in alu_shf). Otherwise it returns 0

Function	Description
isimport(token)	Returns 1 if the token begins with an "i$", which indicates that it is an import variable. Otherwise it returns 0. NOTE: The .import_var directive generates a warning if an import variable does not begin with "i$" and it was used in an isimport() call.
streq(token1, token2)	Returns 1 if both tokens are identifiers which match, or if both are numeric constants which match; otherwise it returns 0.
strstr(token1, token2)	Returns the index of the first occurrence of token2 in token1 (starting with 1). If token2 is not found in token1, it returns a value of 0. If either token is not an identifier, it returns a value of -1.
strlen(token)	Returns the number of characters in token. If the token is not an identifier, it returns -1.
strleft(token1, token2)	Returns an identifier consisting of the leftmost token2 characters of token1. If token1 is not an identifier or token2 is not numeric, the identifier "error" is returned.
strright(token1, token2)	Returns an identifier consisting of the rightmost token2 characters of token1. If token1 is not an identifier or token2 is not numeric, the identifier "error" is returned. NOTE: strright(token, -len) is essentially equivalent to strright(token,strlen(token)-len).
defined(token)	Evaluates to 1 if the token is a macro defined within the preprocessor, or 0 otherwise.
log2(arg, round) log2(arg)	Returns the log-based-2 of arg as an integer. The round argument is optional and if omitted, it defaults to 0. Refer to Section 2.3.3.5 for a detailed description of this function.
__nfp_idstr2meid("idstr")	Returns an MEID for the given string. For example, __nfp_idstr2meid("i32.me9") refers to microengine 9 in island 32. The format of the MEID returned by this function inside the assembler will be the same as the format of __MEID and depends on the chip family.
__nfp_meid(island, menum)	Returns an MEID for the given island and menum. For example, __nfp_meid(32, 9) refers to microengine 9 in island 32. The format of the MEID returned by this function inside the assembler will be the same as the format of __MEID and depends on the chip family.
__nfp_has_island(island)	Returns 1 if the selected chip contains the specified island otherwise 0. The island can be specified via an Island ID Alias or via an island number. For example: __nfp_has_island(26), __nfp_has_island("emem2"), __nfp_has_island("i26") are all equivalent ways of referring to the third external memory island.
is_csr_set("tgt:map.reg.field", value)	Returns true if the CSR (field) is set to "value". The value parameter may be omitted to perform a basic "is set" check to determine if a previous #pragma init_csr was used to set the CSR. If checking against a value, the function returns false if the CSR was not set or if the value does not match - there will be no warning or error message. Note that when referring to a whole register, all init_csr settings to fields of the register are combined to form the whole register value. A CSR field is deemed set for "const" and "required" init_csr settings.
mask(sig)	Evaluates to 1 if sig is a single signal and 3 if sig is a double signal. For a detailed explanation of this function, refer to Section 2.5.9.1 for ctx_arb[--]".

Function	Description
relbase(symbol)	The relbase function returns the linker allocated base address of the given symbol relative to the base of the memory resource, in other words, the address before IMB CPP Address Translation is applied to it. Evaluated at link time.
sizeof(symbol)	The sizeof function returns the byte size of the given symbol. Evaluated at link time.

The constant expression function `strright(token,len)` originally meant to take the *len* characters from the right-most position of *token*. Now, if *len* is <=0, it will mean to drop the leftmost –*len* characters. For example:

```
strright(abcdef, 2)   ef ; original behavior

strright(abcdef, -2)   cdef ; new behavior
```

2.3.3.4 STRING Operator

One of the limitations of the “string functions” within the preprocessor constant-expression parsing, is that they operate on identifiers, not true strings. This has practical implications. For example, a macro may want an ALU operation passed in, and it may want to do something different based on whether that operation allows a shift or not. The problem is that the operation cannot be compared with streq, because the name of some of the operations is not a valid identifier. To address this, there is a new operator defined by single quotes.

This operator will return a valid “identifier”, composed of the text enclosed by the quotes *after* token expansion, that is, after macro arguments are expanded. However, the identifier is created *before* any of the “arguments” are evaluated, based on normal expression rules. More precisely, the function will return an “identifier” formed by taking all of the (expanded) text between the single quotes, minus any leading and trailing white space. For example, you could write:

```
#macro test(arg)
#if (streq('arg', 'b-a'))
…
```

Note that the string operator was used both on the arg and in the comparison string. This points out that the argument to the string operator may be a constant rather than an expandable token. One detail to note: the leading and trailing white space is deleted, but interior white space is not.

In the context of constant-expressions, an identifier can also be created using double-quotes. This behaves the same as the single-quoted version defined above, except that leading and trailing white space is not removed. This would probably be used typically to construct an identifier consisting only of white space, e.g. `strstr('token',″ ″)`.

Note that text within double quotes is not token-expanded.

Here are some examples:

Example 2.2.

```
#define A 1
#define B 2
’ A + B ’ ; evaluates to “1 + 2”
’ 1 2 3 ’ ; evaluates to “1 2 3”
” A + B ” ; evaluates to “ A + B ”
” 1 2 3 ” ; evaluates to “ 1 2 3 ”
```

In this case, the string function would evaluate to “`1 + 2`”, not “`3`”. This illustrates the rule that arguments are expanded but expressions are not evaluated.

2.3.3.5 LOG2() Function

The function log2() can be used within constant expressions. It takes two arguments, the second of which is optional:

```
log2(arg,round)
```

or

```
log2(arg)
```

`arg`: Numeric value (taken as an unsigned value) whose log-2 value is desired.

`round`: Optional numeric value determining how arg is to be rounded.

The function returns the log-based-2 of arg as an integer. If the round argument is not supplied, then the default rounding is 0. If arg is a power of two, then the same value is returned regardless of round. If arg is not a power of two, the behavior depends on round, which is described in the following tables:

Condition	Results when arg is not a power of two
round < 0	Round result down to next smaller integer.
round = 0	Generate an error.
round > 0	Round result up to next larger integer.

Condition	Results when arg is 0
round < 0	-1
round = 0	Generate an error.
round > 0	0

Table 2.5 provides examples of the log2( ) function.

Table 2.5. Examples of log2( ) Function (64-bit mode)

arg	log2(arg) == log2(arg,0)	log2(arg, -1)	log2(arg,1)
0	error	-1	0
8	3	3	3
10	error	3	4
-1	error	63	64

2.3.3.6 Preprocessor Function Examples

The following examples show the usage of the functions.

The defined(token) constant expression evaluates to 1 if the token is a macro defined within the preprocessor, or 0 otherwise. Typical usage would be:

Example 2.3. Defined(token)

```
#if (defined(FOO) || defined(BAR))
```

Example 2.4. ISNUM

```
#macro assign[reg, val]
#if (isnum(val))
// value is a numeric constant
immed[reg, val]
#else
// assume arg is the name of a register
alu[reg, --, b, val]
#endif
...
```

Example 2.5. STREQ

```
#macro something[type]
#if (streq(type,sync))
...
/* This allows the application to specify type as the string
* "sync", assuming that the user has not #defined sync to be
* something else. So the application could call this macro as:
* something[sync]
* or
* something[async]
```

Example 2.6. STRSTR

```
#macro somethingelse[reg]
#if (strstr(reg,@) > 0)
/* reg is absolute */
#else
/* reg is relative */
#endif
```

2.3.4 Macros and Expansion Token Restriction

Macros and expansion tokens share the same name space; therefore, it is invalid to have a macro with the same name as a #define token.

2.3.5 Syntax for Argument and Token Lists

In the preprocessor, several places exist where commas are used in separating items in a list. For example:

```
#for identifier [arg1, arg2, ...]
```

and macro references:

```
macro[arg1, arg2, ...]
```

In both cases, commas can be included in the items list as long as they are enclosed in a matching set of parentheses or brackets. For example, the directive:

```
#for id[item1, foo(bar,bif), immed[reg,32]]
```

would be expanded with `id` taking three values:

```
item1
```

```
foo(bar,bif)
```

```
immed[reg,32]
```

The assembler does not interpret the “,” and take “bif)” or “immed[reg” as values.

2.3.6 Leading and Trailing Spaces in Macros

Leading and trailing spaces in macro arguments are automatically removed by the assembler. For example:

```
macro[ arg1, arg2 ]
```

would be translated by the assembler as

`macro[arg1,arg2]`

2.3.7 Environment Variables

The following environment variables are recognized by the assembler:

`NFAS_INCLUDE`: A list of directories (separated by semicolons) to be added to the include path:

dir1;dir2;dir3...

and is appended after the directories supplied on the command line.

`NFAS_NFPTYPE`: Used to set default processor type. See Section 2.3.8.2.

`NFP_CHIP_ID`: Similar to `NFAS_NFPTYPE`, used to set default processor type.

2.3.8 Predefined Macros

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

As the `IS_NFPTYPE` mechanism (described in the next section) can only be used with assembler (i.e. microcode), commencing with NFP SDK 4.6.2, an alternate mechanism to determine the processor type can be used. The new mechanism can be used in Micro-C source code, assembler source code. This mechanism relies on the fact that the NFP SDK will define the macro `__NFP_IS_6000` when compiling C code or assembling microcode with NFP-6xxx processors. Neither the Intel IXA SDK nor 4.x versions of the NFP SDK will define these macros, therefore code like the following can be compiled or assembled with the IXA SDK or the NFP SDK:

Example 2.7. Different Code Sequences for Netronome NFP Devices and Other (e.g. Intel IXP) Devices

```
#ifdef __NFP_IS_6000
   // Micro-C code or assembler code for NFP-6xxx processors
#elif __NFP_IS_3200
   // Micro-C code or assembler code for NFP-32xx processors
#else
   // Micro-C code or assembler code for other (e.g. Intel IXP) processors
#endif
```

2.3.8.2 IXA SDK Compatible Assembler Only Predefined Processor Type and Revision Macros

The preprocessor defines the macro __NFPTYPE, which identifies the target processor type(s) for the code being assembled. It is defined as some combination of the various processor type macros also defined by the preprocessor. Table 2.6 lists the processor type macros and their meanings. To enable code to be built with the Intel IXA SDK as well as with the NFP SDK, the preprocessor defines __IXPTYPE as alternate name for this macro.

The preprocessor also defines two macros __REVISION_MIN and __REVISION_MAX, which specify the target processor revision for the code being assembled. The values of these macros are defined according to the command line values or by the Programmer Studio assembler settings. The revision values consist of an 8-bit value. The upper four bits correspond to the major revision number (which is a letter, such as “A”, “B”, etc) while the lower four bits correspond to the minor revision number (such as 1, 2, 3, etc.)

Table 2.6. Processor Type Macros

Macro	Meaning	Type
__IXPTYPE	Value is determined by command line arguments. It takes on some combination of the following values.	Variable
__IXP2805	Intel IXP2805 processor.	1 bit set
__IXP28X5	Intel IXP2805 or IXP2855 processors.	n bits set
__IXP2XXX	All Intel IXP2xxx processors.	n bits set
__NFP3216	Netronome NFP-3216 processor.	1 bit set
__NFP3240	Netronome NFP-3240 processor.	1 bit set
__NFP3200	All Netronome NFP-32xx processors.	n bits set
__NFP6000	All Netronome NFP-6xxx processors.	n bits set
__NFP3800	All Netronome NFP-38xx processors.	n bits set

All of these macros, other than the first one, are pure constants that have only one bit set (for the processor-specific ones). The __IXPTYPE macro indicates the processor types for the code being generated. Its value can consist of a single processor type or a combination of types. The preprocessor also defines macros for the various revision values, as shown in Table 2.7.

Table 2.7. Revision Macros

Macro	Meaning	Value
__REVISION_MIN	Minimum processor revision for code being assembled.	Variable. Default is 0x00
__REVISION_MAX	Maximum processor revision for code being assembled.	Variable. Default is 0xff
__REVISION_A0	A0 revision.	0x00
__REVISION_A1	A1 revision.	0x01
__REVISION_B0	B0 revision.	0x10
__REVISION_B1	B1 revision.	0x11
__REVISION_C0	C0 revision.	0x20
etc...		

The leaf nodes represent values with a single bit set while non-leaf nodes represent the union (bitwise-OR) of the nodes below them.

A default value can also be set using the environment variable NFAS_NFPTYPE. For example:

```
set NFAS_NFPTYPE=nfp3216
```

These predefined macros should be used with the IS_NFPTYPE(type_expression) or IS_IXPTYPE(type_expression) constant expression function described in Section 2.3.3.3.

Example 2.8. Different Code Sequences for Intel IXP28xx and Netronome NFP Devices

```
#if (IS_IXPTYPE(__IXP28XX))
   // code for IXP2800
#elif (IS_IXPTYPE(__NFP3200))
   //code for NFP-32xx
#else
   // code for NFP processors
#endif
```

Example 2.9. Code Only Works on NFP-6xxx

```
#if (!IS_IXPTYPE(__NFP6000))
   #error
#endif
```

Any code targeted at a specific revision of the Netronome Network Flow Processor NFP-6000 should make use of the predefined __REVISION_MIN and __REVISION_MAX macros. For example, if the code is designed to exploit certain features found only in processor revisions A1 or higher, then the __REVISION_MIN macro can be used along with the #error directive to abort the assembly process.

Example 2.10.

```
Revision
Number           Hex Value
--------------------
A0:              0x00
A1:              0x01
A2:              0x02
B0:              0x10
B1:              0x11
B2:              0x12
C0:              0x20
etc.
```

The default values for __REVISION_MIN and __REVISION_MAX are 0x00 and 0xff, respectively.

In addition to the __REVISION_MIN and __REVISION_MAX macros, the assembler also predefines macros of the form:

```
#define __REVISION_A0 (0x00)
#define __REVISION_A1 (0x01)
 ...
#define __REVISION_B0 (0x10)
#define __REVISION_B1 (0x11)
...
etc.
```

Example 2.11. Revision Macro Use

```
#if (__REVISION_MIN < __REVISION_A1)
#error "This feature is not supported on revision(s) prior to A1"
#else
; version specific code here
#endif
```

Example:

For more information on setting the processor type or revision, please refer to the *Netronome Network Flow Processor NFP-6xxx Development Tools User's Guide*.

2.3.8.3 Additional Predefined Macros

Table 2.8. Additional Predefined Macros

Macro	Description
__NFAS_VERSION	This is defined to a string indicating the version of the assembler. It is in the form M.m.b where M, m, and b are integers and M is the major version, m is the minor version, and b is the build number.
__NFAS_MAJOR_VERSION	This is defined to an integer indicating the major version of the assembler.

Macro	Description
__NFAS_MINOR_VERSION	This is defined to an integer indicating the minor version of the assembler.
__NFAS_BUILD_NUM	This is defined to an integer indicating the build number of the assembler.
__NFP_TOOL_NFAS	This is defined when code is being processed by NFAS, and not another tool. It can be used to for example permit code to be either preprocessed by a standalone preprocessor or assembled with NFAS, with `#ifdef` directives being used to ensure that the appropriate code is emitted in each case. The value of this macro will be the version of the SDK, as returned by `NFP_SW_VERSION(major, minor, sub-minor, sub-sub-minor)` , for example NFP_SW_VERSION(5, 0, 0, 0). (NFP SDK tools other than NFAS will define similar macros, e.g. `__NFP_TOOL_NFCC` are defined by NFCC)
__NFP_LANG_ASM	This is defined when the file being processed should be written in the assembler language, e.g. is a `.uc` file, or when an include file, e.g. a `.h` file, should conform to assembler language syntax and conventions. It can be used to for example ensure that appropriate constructs are used in files that are included by Micro-C, assembler. The value of this macro will be the version of the SDK, as returned by NFP_SW_VERSION(major, minor, sub-minor, sub-sub-minor), for example NFP_SW_VERSION(5, 0, 0, 0). (NFP SDK tools other than NFAS will define similar macros, e.g. `__NFP_LANG_MICROC` are defined by NFCC)
__NFP_INDIRECT_REF_FORMAT_V1 and __NFP_INDIRECT_REF_FORMAT_V2	The former or the latter of these is defined, depending on whether the code being assembled needs to use NFP-32xx compatible (Version1) or NFP-6xxx (Version2) enhanced indirect formats. The latter is defined if the `-indirect_ref_format_v2` command line option or when assembling code with default settings. The former is defined if the `-indirect_ref_format_v1` command line options are used or if equivalent GUI checkboxes are activated.
__PREPROC32 and __PREPROC64	The former or the latter of these is defined, depending on whether the assembler preprocessor is operating in 32-bit or 64-bit mode. The latter is defined if the `-preproc64` command line option or corresponding GUI checkbox is activated, or when assembling code with default settings, as NFP enhanced settings are the default. The former is defined if the `-preproc32` command line options are used (provided `-preproc64` was not also specified), or if equivalent GUI checkboxes are activated.
__DATE__	ANSI standard macro defined to a string indicating the date that the assembler was invoked. It is in the form: Mmm dd yyyy, where Mmm is the first three letters of the month name, e.g. “Jan”, dd is the date with a leading zero, and yyyy is the year.

Macro	Description
__TIME__	ANSI standard macro defined to a string indicating the time that the assembler was invoked. It is in the form hh:mm:ss, where hh is the hour, mm is the minutes, and ss is the seconds.
NFP_SW_VERSION(major, minor, sub-minor, sub-sub-minor)	This predefined preprocessor macro builds a number from the separate version numbers which can then be compared to __NFP_TOOL_NFAS, for example `(__NFP_TOOL_NFAS > NFP_SW_VERSION(5,0,0,0))` means the assembler version is greater than version 5.0.0.0. The `sub-sub-minor` number is used to differentiate between interim releases, if any, and will not necessarily step in single increments. It is not related to the build number.

2.3.9 Predefined Import Variables

The Predefined Import variables are detailed in Table 2.9. For more information on import variables, refer to Section 2.8.9.

Table 2.9. Predefined Import Variables

Name	Definition
__MEID, __UENGINE_ID	The format depends on the processor type: • NFP-32xx: __UENGINE_ID = (cluster_num << 4) | (me_num) NFP-32xx: __MEID = (cluster_num << 4) | (me_num) | 0x8 • NFP-6xxx: __MEID = (island_id << 4) | (me_num + 4) On microengines configured for shared control store bit 0 of __MEID cannot be used due to the fact that it evaluates to a different result based on whether the instruction in which it is used is located at an odd or an even code address (which are loaded to the odd and even microengine code stores). For NFFW files, this is evaluated at link time.
__ISLAND	The value depends on the processor type: • NFP-32xx: __ISLAND = (cluster_num) • NFP-6xxx: __ISLAND = (island_id)
__MENUM	The value is the number of the microengine within the island, starting at 0 for the first microengine.
__ADDR_<ID>	This is a series of predefined import variables that are resolved by the linker using the link-time IMB CPP Address Translation tables to get the base address of the given memory resource or target from a given microengine. The possible values for <ID> are the same as the memory types used for '.alloc_mem name mem_type scope size', except upper case and dots are replaced with underscores, as well as some additions for non-memory targets. For example, __ADDR_I32_CLS is address 0 of CLS in island 32. When this is used from island 32 it is zero, when used from other islands it will be a 40-bit address with island ID bits set. Both __ADDR_I24_EMEM and __ADDR_EMEM0 refer to address 0 of emem0. __ADDR_CTM refers to the local CTM and __ADDR_CLS to the local CLS. The local CLS and CTM addressing is intended to be simple and a 40-bit address of 0 should target the local resource, but these __ADDR variables are provided for completeness. The non-memory targets like PCIe and NBI can also be addressed like this. __ADDR_NBI0 and __ADDR_I8_NBI will both resolve to address 0 of NBI 0, the address used with nbi[] instructions. __ADDR_PCIEn can be used to target PCIE n when using pcie[] instructios (also __ADDR_Ix_PCIE for PCIE island x). __ADDR_ILA0 and __ADDR_I48_ILA with ila[] instructions and __ADDR_ARM with arm[] instructions. Finaly, __ADDR_CRYPTO0 and __ADDR_I12_CRYPTO with crypto[] instructions.

Name	Definition
__MU_LOCALITY_LSB	This will resolve to the lower bit in the address where the MU locality value is located, from the island the list file is assigned to (based on that island's IMB CPP Address Translation setup).
__MU_LOCALITY_HIGH	This will resolve to a 40-bit address that contains only the MU locality value for High Locality, no island ID, no base address. This can be bit-wise ORed with a memory symbol to control locality if necessary. It also takes into account the IMB CPP Address Translation from the originating island. Note that IMEM and CTM memory symbols already resolve to addresses using Direct Access locality, so these are mostly useful for EMEM access.
__MU_LOCALITY_LOW	Similar to above, but for Low Locality.
__MU_LOCALITY_DISCARD	Similar to above, but for Discard After Read.
__MU_LOCALITY_DIRECT	Similar to above, but for Direct Access.

2.4 Case Sensitivity

The command line arguments are case sensitive. If the "-C" command line argument is not specified, the microcode file is case insensitive – all text in the file, with the exception of comments, is converted to lower case.

If case sensitivity is enabled (the "-C" argument was specified), the Assembler does not convert input file text to lower case. Predefined macros and predefined import variables, as well as user-defined registers, signals, and import variables will all be case sensitive. For compatibility with earlier releases, Assembler keywords will be handled in a case insensitive manner (converted to lower case). These include the following:

- Instruction and directive names (e.g. alu and .reg)
- Operand Keywords (e.g. alu operators and CSR names)
- Built-in preprocessor function names.

2.5 Registers and Signals

The Assembler resolves symbolic register names into physical register addresses. The following sections describe the details of registers and signals.

2.5.1 Register Naming Conventions

Registers are specified symbolically using a string of alphanumeric characters (including “_”). The first character of a register name cannot be numeric. The register type (such as GPR and Transfer) and the addressing mode (context-relative or absolute) is determined by prefixing the register name with the reserved characters “@” and “$”. Absolute addressing is only supported for GPR registers; it is not supported for transfer registers.

Register types are defined by prefixes applied to the register names. Registers have a type based on the name. The table below shows the prefix (in bold) that is applied to register names to specify the type. The word “reg” is the user specified name of the register. Local memory is shared by all contexts using an index register and does not support relative or absolute names. Transfer and neighbor registers can also be accessed globally using index registers.

Register Type	Relative Name	Absolute Name
GPR	reg	@reg
Transfer	**$**reg	not available
Next Neighbor	**n$**reg[a]	not available

[a]Named next neighbor registers are not supported in restricted addressing mode.

2.5.1.1 Indexed Registers

It is possible to access some of the register types by using an index register. An index register is a local CSR that points to an address in the related register file. The actual register that is addressed, can be accessed in a manner similar to that of “normal” registers.

Generally, the index is set using the local_csr_wr instruction. The exception is the index register used for writes to the neighbor register. In this case, the CSR is located in the neighbor Microengine and is not visible to the writer (although the neighbor can access it). Typically, it is initialized by the neighbor and then the neighbor registers function as a FIFO, with the read/write index registers never being directly set again (they are always advanced indirectly via postincrement).

Local memory can only be accessed through indexed registers; it does not support the use of “named” registers. In the case of local memory, there are four independent index registers (0..3). For each of these index registers, the contexts in a Microengine can either share the same register, or each context can reference its own copy. For all other indices, each context within a Microengine shares the same index.

Neighbor registers can be accessed by name or by an index, but you will not typically access the registers using both names and index in the same program[1]. Transfer registers can also be accessed by name or by an index; however, it is more likely that you will access the registers using both names and index in the same program.

Each of the indices supports a number of additional features, including:

- Post-increment
- Post-decrement
- Offsetting

The feature set of the different indices are shown in the following table:

Register Type	Register Name	Postincrement	Postdecrement	Offsetting	Local CSR Name for Index Register1
Local Memory	*l$index0 *l$index1	*l$index0++ *l$index1++	*l$index0-- *l$index1--	*l$index0[n] *l$index1[n]	ACTIVE_LM_ADDR_0 ACTIVE_LM_ADDR_1

[1]The index implements a FIFO that will eventually overwrite any named register.

Register Type	Register Name	Postincrement	Postdecrement	Offsetting	Local CSR Name for Index Register1
	*l$index2 *l$index3	*l$index2++ *l$index3++[a]	*l$index2-- *l$index3--[a]	*l$index2[n] *l$index3[n][b]	ACTIVE_LM_ADDR_2 ACTIVE_LM_ADDR_3
Next Neighbor Fifo	*n$index	*n$index++[c]	N/A	N/A	NN_PUT NN_GET
Transfer	*$index	*$index++	*$index--	N/A	T_INDEX

[a]Post increment and Post decrement are supported for Absolute/Relative mode.

[b]For offsetting, n = 0 to 15 for Context/Relative mode.

[c]Optional when used as a source and required when used as a destination.

The use of an indexed register reference is indicated by the leading asterisk ("*") in the register name. After that follows the normal type prefix and then the keyword "index". In the case of local memory, the keyword "index" is followed by a 0 or 1 to identify which Local Memory CSR index register to use. Post-increment or post-decrement is indicated by appending "++" or "--" to the register name. For example, "`*n$index++`".

Offsetting refers to addressing a word (4-bytes) at a fixed offset from the word addressed by the Index local CSR. The offset is a constant in the range from 0 to 15 words. When an offset is used, the offset is bit-wise ORed into the address. This means that the address register must be aligned on an appropriate boundary for offsetting to work[2]. Offsetting is indicated by appending a numeric constant surrounded by square brackets to the register name. An example would be "`*l$index2[3]`".

Offsetting cannot be used with post-increment or post-decrement.

An index register is the *only* way that the local memory can be accessed. Local memory does not support the use of "named" registers as do the other register files. In the case of local memory, there are two independent index registers for each context and one set for the active context (the context that is currently executing).

In the case of local memory, the FOUR index registers are referenced within the instructions using the keywords *l$index0, *l$index1, *l$index2, or *l$index3. These keywords always refer to the active set. The Microengine can be put into a mode where all contexts share the same two registers (the active set), or each context uses its own set. This is specified using the Assembler directives local_mem0_mode, and local_mem1_mode (refer to Section 2.8.14 for more information on these directives). When a Microengine is set up to have each context use their own four index registers and a context begins executing, its set of registers is loaded into the active set and when the context goes to sleep, the active set is saved back to context's set. In the mode where all contexts share the same four registers, the active set is only set that is used.

The local memory index registers are loaded using the local_csr_wr instruction. The following register names can be used to access the four independent index registers for the active context.

ACTIVE_LM_ADDR_0	ACTIVE_LM_ADDR_1
ACTIVE_LM_ADDR_0_BYTE_INDEX	ACTIVE_LM_ADDR_1_BYTE_INDEX
ACTIVE_LM_ADDR_2	ACTIVE_LM_ADDR_3
ACTIVE_LM_ADDR_2_BYTE_INDEX	ACTIVE_LM_ADDR_3_BYTE_INDEX

[2]Alignment needs to be maintained by the programmer. The Assembler cannot check for proper alignment.

For the contexts that are not active, you can access their local registers by first setting CSR_CTX_POINTER to the correct context number. Then, the following register names can be used to access the four independent index registers of the specified context.

INDIRECT_LM_ADDR_0	INDIRECT_LM_ADDR_1
INDIRECT_LM_ADDR_0_BYTE_INDEX	INDIRECT_LM_ADDR_1_BYTE_INDEX
INDIRECT_LM_ADDR_2	INDIRECT_LM_ADDR_3
INDIRECT_LM_ADDR_2_BYTE_INDEX	INDIRECT_LM_ADDR_3_BYTE_INDEX

Refer to the "Local CSRs" section in the *Netronome Network Flow Processor 6XXX: Databook* for more information on all these registers.

The Next Neighbor registers can use the index register in support of Next Neighbor rings. When the destination register is specified as *n$index++, the NN_PUT index register is used to perform a put operation. When the source register is specified as *n$index++, the NN_GET index register is used to perform a get operation.

The entire transfer register set can be accessed by a context using the Transfer Register Index Registers (T_INDEX). A register number (as shown in Table 2.10) is written to the T_INDEX register using the local_csr_wr instruction. Then a transfer register is read or written using the notation shown in Table 2.10 to specify either a source (for a write) or a destination (for a read).

Table 2.10. Registers Used By Contexts in Context-Relative Addressing Mode

Number of Active Contexts	Active Context Number	GPR Absolute Register Numbers		Neighbor Index Number	Transfer Register Number
		A Port	B port		
8	0	0-15	0-15	0-15	0-31
	1	16-31	16-31	16-31	32-63
	2	32-47	32-47	32-47	64-95
	3	48-63	48-63	48-63	96-127
	4	64-79	64-79	64-79	128-159
	5	80-95	80-95	80-95	160-191
	6	96-111	96-111	96-111	192-223
	7	112-127	112-127	112-127	224-255
4	0	0-31	0-31	0-31	0-63
	2	32-63	32-63	32-63	64-127
	4	64-95	64-95	64-95	128-191
	6	96-127	96-127	96-127	192-255

Two registers of the same type, other than GPR, cannot appear as source operands in a single instruction, but two registers of the same type can appear, with one being a source and one being a destination. This raises the question of what happens if in this case one wishes to apply an increment/decrement operator to that register. The rule that the Assembler uses, is that when the same index register is used as both a source and destination, any

increment/decrement operator must be applied to the destination usage. Usage on the source or on both will result in an error.[3]

Thus:

```
alu[*l$index0++, 1, +, *l$index0 ]
```

would be valid, but

```
alu[*l$index0 , 1, +, *l$index0++]
```

```
alu[*l$index0++, 1, +, *l$index0++]
```

would not. This is to prevent confusion in people who are looking at the code and who might think that the first bad case is writing to the next register after the one being read, and who might think that the second bad case is incrementing the index register twice.

Neighbor registers have different read and write pointers, so both of the following are valid:

```
alu[*n$index++, 1, +, *n$index]
```

```
alu[*n$index++, 1, +, *n$index++]
```

It is allowed that the same local memory index can be offset differently as source and destination in the same instruction. Thus, the following is valid:

```
alu[*l$index0[3], 1, +, *l$index0[4] ]
```

It is not valid, however, to use a post-modify on the destination and an offset on the same index as source.

2.5.1.2 Mixing Indexed and Named Register Usage

Use of index registers does not result in any register allocation. Conceptually, this is similar to the C language behavior that "`int *p;`" does not allocate an integer.

Local memory can be allocated and managed manually. You can allocate and choose specific addresses for local memory use through the `#define` statements; however, the preferred method is to use the memory allocation directives described in Section 2.8.12 to allocate blocks of local memory.

Caution

In the case of neighbor registers, the two methods (indexed and named) are conceptually exclusive. When the indexed method is being used, one would not be using the named method, and since the index defines a FIFO covering the entire register array, allocation is not relevant.

[3]Conceptually, the hardware samples the value of the index register and uses that for both the source and destination references. Meanwhile, the index register is modified. So it makes no sense to think about incrementing the register after the read operation but before the write operation or incrementing it twice.

For transfer registers, however, indexed and named usage may be mixed. This is partially a result that I/O references work on named transfer registers and not indexed transfer registers. To cause the register allocation to occur, all of the registers need to have names and to be part of the .xfer_order instruction, regardless of whether you will actually reference them by name. Additionally, you need to use .set or .use directives to indicate when these registers are being used.

2.5.1.3 Transfer Registers

The Transfer Registers (`xfer`) parameter specifies a Transfer register. Transfer registers are always specified with one $ character as a prefix. Read and write transfer registers are specified by how they are used. For example, reading `$xfer` reads a Transfer Read register, while writing the register writes an Transfer Write register.

When an I/O instruction specify a reference count (`ref_cnt`) greater than 1, the data from multiple transfers are read or written from a contiguous set of Transfer registers. In this case, the `xfer` parameter specifies the first register in the contiguous set. Because the instruction only specifies the first register in the contiguous set of registers, the Assembler requires that you indicate the names of registers that you would like to use for the other contiguous registers. This is specified using the `.xfer_order` Assembler directive.

When the "indirect_ref" token is used with one of the command instructions, it is possible to denote a transfer register with a single dollar sign as shown in the example below:

```
mem[read, $, base, offset, max_8], indirect_ref, sig_done[s]
```

In this case it is assumed that the transfer register will be overridden by the ALU output of the previous instruction.

2.5.2 Register Declarations

The main purpose of register declarations is to assist you with catching bugs; primarily those resulting from typing the name of a register incorrectly. The use of register declarations may be made optional or required depending on the command-line switches. If declarations are not required, then a register that is used without being declared is implicitly declared with a global scope and an automatic lifetime.

2.5.2.1 Preferred Register Declaration Syntax

Registers are declared[4]using the .reg directive. Registers can have four different attributes. These are specified as described later in this section by keywords. The default attribute values (i.e., the attributes specified with no keywords) are underlined.

Scope This determines what part of the source code can reference this register. Conceptually, this can have one of three values (if the declaration occurs within a block, then block is the default scope; otherwise, module is the default):

- Block : The virtual register can only be referenced from its declaration to the end of the enclosing block. This is similar to a variable declared within the body of a C-function.

[4]Note that index registers do not need to be declared. Since local memory can only be accessed by use of index registers, it is never declared.

- Module: The Assembler currently has no notion of “module”. This attribute is reserved for possible future use. At this release, the module attribute is effectively the same as having it at global scope except that the name is prefixed in the list file. This is similar to a top-level nonstatic variable in C.
- Global: The virtual register can be referenced from its declaration to the end of the module, or from within other modules (via extern declarations). This is similar to a top-level non-static variable in C.

Lifetime This determines how the register allocator allocates this register. Conceptually, this can have one of two values, Automatic and Volatile:

- Automatic: The allocator will determine those parts of the code where the register contains a meaningful value. This is called the live-range of the virtual register. Two registers whose live-ranges do not overlap, may be safely allocated to the same physical register. This is the normal situation.
- Volatile: The virtual register is allocated to a dedicated physical register; i.e. no other virtual register will be allocated to the same physical register. This guarantees that a reference to a different virtual register will never modify values in this virtual register. This would be needed if either the allocator’s algorithms compute the wrong live-range, or if the register were to be accessed asynchronously (e.g. a transfer register that was to the be target of a reflector operation).

Direction In the case of transfer registers, this indicates which of the read/write transfer registers are being allocated. This can take three values:

- Read: Only a read transfer register is declared.
- Write: Only a write transfer register is declared.
- Both: Both a read and a write transfer register (at the same address) is declared. This is needed for operations that do both a read and a write at the same time, e.g. test-and-set.

A transfer register that is not explicitly declared “read” or “write” (that is, it is implicitly declared as read/write or “both”) is considered as two separate but linked physical registers. That is, the live-range is computed for each of the pair separately.

This means that declaring a transfer register with the “read” or “write” keywords has no effect on register allocation. This is because if it is declared as “both”, but only used as a “read”, then the write “half” of the register will have an empty live range, and so it will not conflict with anything else. The same holds true if it is declared as “both” but only used as a “write” transfer register.

The only advantage to declaring a register as “read” or “write” is that attempts to use that register incorrectly (for example, making a read transfer register the destination of an ALU instruction) results in an error. If the register was declared as “both”, then such an attempt would not generate an error, although it might generate a warning.

Visibility In the case of transfer registers, this indicates whether the register is visible to other microengines via the reflector. Neighbor registers are always visible. Conceptually, this can take one of two values:

- Visible: Other microengines can read/write this register.
- Invisible: Other microengines cannot read/write this register.

Note

It would be rare to have an absolute register declared with an automatic (non-volatile) lifetime. Since absolute registers are generally accessed by multiple contexts, it should generally have a volatile lifetime.

If an absolute register is declared with an automatic lifetime, the Assembler may choose to treat it as if its lifetime were volatile.

A visible transfer register would automatically be considered volatile. The syntax of the register declaration is:

```
.reg [keywords]* name1 name2 …
```

where

`keywords : Zero or more keywords as described below.`

`names : One or more register names. You cannot declare a register whose name matches one of the keywords.`

The keywords define the attributes of the registers being declared, as defined by the following table:

Keyword	Meaning
volatile	If the volatile keyword is present, the lifetime of the declared registers is set to volatile. Otherwise, the lifetime is automatic. Note that in some cases (e.g. named neighbor registers), the lifetime is always volatile, regardless of the absence or presence of this keyword.
global	If the global keyword is present, the scope of the declared registers is set to global. Otherwise, if the declaration is within a block, the scope is set to block. If it is not within a block, then the scope is set to module.
visible	If the visible keyword is present, the visibility of the declared transfer registers is set to visible. Neighbor registers are always visible.
read/write	If either the read or write keywords are present, the direction for transfer and neighbor registers is set accordingly. These attributes have no effect on GPRs. If neither keyword is given, or if both keywords are given, then the direction is set to both.
extern	If the extern keyword is present, the named registers are declared elsewhere (either in this module or another that will be linked in). This is similar to the C-language construct “extern type name”.
remote	If the remote keyword is present, the named transfer or neighbor registers must be declared in a different microengine and will presumably be the target of a neighbor write or a reflector reference. These are resolved by nfld. The remote register must be declared as visible in the remote microengine to be seen by this microengine.

Usage of these keywords is summarized in the following tables:

	GPR	Xfer	Neighbor
volatile	valid	valid	implied[a]
global	valid	valid	implied

	GPR	Xfer	Neighbor
visible	error	valid	implied
read/write	error	valid	error
extern	valid	valid	valid
remote	error	valid	valid

[a]Implied means that it may be specified or not. In either case, the program behaves as if it was specified.

Keyword compatibility				
	volatile global visible	read/write	extern	remote
volatile global visible	x	OK	error	error
read/write	OK	x	OK	error
extern	error	OK	x	error
remote	error	error	error	x

Rules:

1. Remote cannot be used with any other keywords.
2. Extern can only be used with read or write.
3. Other keywords may be freely mixed.

It is valid to declare the same register as ".reg extern" and ".reg global". It is also valid to declare a remote register multiple times. This would occur when a macro which contains a remote register declaration, is used multiple times.

It is valid to declare the same register name as remote and non-remote. In this case, context will determine which register is referenced. This usage is confusing and should be avoided, but there may be strange cases where this is required (such as two microengines running identical code and which want to access each other's transfer registers).

To declare registers with a "module" scope, they should be declared outside of any .begin/.end blocks and without the GLOBAL keyword. (See Section 2.5.1.2.)

The attribute implications are summarized as follows:

neighbor $\Rightarrow$ global, volatile, visible
visible $\Rightarrow$ global, volatile
remote $\Rightarrow$ global
extern $\Rightarrow$ global

The response to declaring a register with the same name as a previously declared register, is summarized in the following table. Note that for the "first register/Block" column, the assumption is that the second register is declared within the same block.

		First Register				
		Global	**Module**	**Block**	**Extern**	**Remote**
Second Register	Global	Error	Error	Warn	OK[a]	OK
	Module	Error	Error	N/A[b]	Error	OK
	Block	Warn[c]	Warn[c]	Error	Warn[c]	OK
	Extern	OK[a]	Error	Warn	OK[a]	OK
	Remote	OK	OK	OK	OK	OK

[a]First and second registers refer to same register.

[b]A module-scoped register cannot be declared within a block.

[c]Generates a high-level (level-4) warning.

In such cases, more than one register will exist with the same name, and which of them is referenced, depends on the context. The following example shows that a local register declaration will mask a global variable declaration of the same name within the scope of the local block:

```
.begin
.reg foo             // defines a local variable named foo
.reg global foo      // defines a global variable named foo
foo…                // reference to foo is to the local variable
.end
foo …               // reference to foo is to global variable
```

The directive `.xfer_order` does not declare registers. The arguments to `.xfer_order` need to be declared before they are used in the `.xfer_order`. If you want to avoid writing out the variable list twice, you can use a macro similar to:

```
#macro reg_order[type, regs]
.reg type regs
.xfer_order regs
#endm
reg_order[global volatile, $1 $2 $3]
reg_order[,$4 $5 $6] Note, the null type field
```

2.5.2.2 Details of Volatile and Visible

Registers can be declared as volatile without being declared global. This means that the virtual register is allocated to a dedicated physical register within the defining block. If the code flow leaves that block, then the register could be reallocated. The typical use for this feature would be microcode where different code blocks correspond to different independent threads. In this case, the defining block for the volatile registers would consist of all of the code for a given thread. For example, the fact that context-0 had a particular volatile register should not affect the allocation of the registers local to context-1.

Since volatile attribute does not imply global, it is possible to have non-global visible registers. In this case, there is the added restriction that any particular Microengine could only declare one visible register with a given name. If two were declared, then there would be no way to distinguish them. For example, the following code would be invalid:

```
.begin
.reg visible foo
…
.end
.begin
.reg visible foo         // Not allowed because there are two visible foo's
```

Note

This would have been legal if either or both declarations had omitted the visible keyword.

2.5.2.3 Dealing with Self-Write Neighbor Registers

There is an ambiguity when writing to named neighbor registers. Normally, doing so writes to a register in the neighboring Microengine. However, a CSR bit can be set that causes writes to go to the same Microengine. This might be used, for example, to store certain constants or pseudoconstants for later use. The Assembler cannot determine the setting of this CSR bit, so you must indicate whether a write to a named neighbor register is going to the neighbor Microengine or to the self Microengine.

Usually, the destination register is declared via a `.reg` directive, and it is assumed to be local to this Microengine. If it is declared via a `.reg remote` directive, then it is assumed to be in the neighbor. The one ambiguous situation is if the register is declared with both a `.reg` and a `.reg remote` directive. In this case, it will be assumed that the actual destination of the write is the register in the remote Microengine. While this usage may be needed in rare occasions, it is confusing and should be avoided.

Note

You must ensure that the destination of the neighbor write matches the current setting of the CSR bit; otherwise, a random register in the wrong Microengine will be modified.

2.5.3 Register Arrays

The Assembler supports the notion of an array of registers. This is called an “aggregate”. These are declared by giving a bracketed size following the name in “.reg”. For example:

```
.reg $x[3]
```

declares an aggregate called “$x” consisting of three registers. The maximum size for an aggregate is 128 registers. In the case of remote registers, the size can be left off, e.g. “.reg remote $r[]”. You cannot have an aggregate with the same name as a non-aggregate; the following would be invalid:

```
.reg $a[3]
.reg $a ; illegal because of $a[3] above
```

Aggregates cannot be implicitly declared, they must be declared explicitly.

The .xfer_order directive takes entire aggregates, not aggregate elements. For example:

```
.xfer_order $a $x $b
.xfer_order $a $x[0] $b ;; ERROR, can't use indices
```

This would result in the registers being ordered as “$a $x[0] $x[1] $x[2] $b”.

In instructions and most directives (exclusive of .reg and .xfer_order), elements of an array are referenced with a bracketed index. Continuing the example above:

```
immed[$x[0], 1]
alu[$x[1], $x[2], +, 1]
alu[--,--,b,$x] ;; ERROR: missing index
immed[$x[3], 0] ;; ERROR: index out of range
```

2.5.4 Doubled Signal References

A doubled signal is another type of “aggregate”. For some I/O instructions (i.e. some DRAM references) both signals are needed to indicate that the I/O has completed. For other I/O instructions that generate a doubled signal, one “half” indicates that the write-transfer-register has been consumed, and the other “half” that the read-transfer-register has been filled.

From a syntactic point of view, references to a doubled signal name by itself (e.g. “sig”) will refer to essentially the entire pair. To reference the low half, either “[0]” or “[write]” will be appended to the signal (e.g. “sig[0]” or “sig[write]”). To reference the high half, one would append either “[1]” or “[read]” to the signal (e.g. “sig[1]” or “sig[read]”). References to “[0]” versus “[write]” (or “[1]” versus “[read]”) are equivalent; i.e. either can be used interchangeably.

Note

Doubled signals appear very similar to an array of signals (of length 2), except that in this context, “write” is a synonym for “0”, and “read' is a synonym for “1” (as an aid in readability). Doubled signals can be referenced three ways: via the address operator, via a br_*signal, and via a ctx_arb.

The address operator can be applied to the unqualified name (e.g. “sig”), or to the qualified names (“sig[0]” or “sig[read]”). The address of “X” would be the address of “X[0]”. Examples would include:

```
immed[x, &s] ; references low/write half
immed[y, (1+&remote(s))] ; references high/read half
```

The br_*signal (br_signal, br_!signal) instructions always reference qualified names. The following would be valid:

```
l1#: br_!signal[dram_sig[0], l1#] ; references low half
l2#: br_!signal[dram_sig[1], l2#] ; references high half
l3#: br_!signal[sram_sig[write], l3#] ; references low half
l4#: br_!signal[sram_sig[read], l4#] ; references high half
```

The br_*signal functions should not reference the unqualified names.

The ctx_arb instruction can reference qualified or unqualified names. If the unqualified name is used, then it will be automatically doubled by the assembler, for example:

```
dram[read,Xfer,base,offset,  sig_done[s]
ctx_arb[s] ; equivalent to ctx_arb[s[0], s[1]]
```

For advanced users, the individual halves of the signal can be referenced by using the qualified names, for example:

```
sram[swap, $x, addr,0,--], sig_done[s]
ctx_arb[s[read]] ; not automatically doubled due to "[read]"
...; it is now safe to access $x as a read xfer reg
ctx_arb[s[write]] ; not automatically doubled due to "[write]"
...; it is now safe to access $x as a write xfer reg
; or to reuse "s"
```

2.5.4.1 Optional error signals

Some I/O instructions may only return a second signal in the case of an error (For example mem[get, ...]). For this case, one should wait on the first signal (signame[0]) and only once that signal has returned should one check for the optional error signal (signame[1]). For example:

```
mem[get, $x, addr, 0, 1], sig_done[s[0]]
ctx_arb[s[0]] ; Wait for first signal
br_signal[s[1], handle_error#]  ; Branch to error code if signal is set
```

2.5.5 Transfer Order

The `.xfer_order` directive describes an ordering of transfer registers. In the case of transfer requiring multiple 32-bit data transfers, it is necessary to describe to the Assembler those 32-bit register names that must be contiguously ordered in the transfer register address space. This is so that the intended data can be accessed predictably by symbolic specification to individual registers. The `.xfer_order` reserved name is followed by the ordered list of register names (order increases from left to right).

Instruction format

```
.xfer_order reg1 reg2 ...
```

All registers that are related via `.xfer_order` must have the same scope. For example, the following would be invalid:

Example: Incorrect Usage of Xfer_order (1)

```
.begin
    .reg $x1 $x2
    .begin
    .reg $x3 $x4
    .xfer_order $x1 $x2 $x3 $x4 // invalid
```

Similarly, the following would also be invalid:

Example: Incorrect Usage of Xfer_order (2)

```
.reg $x1
.reg visible $x2
.xfer_order $x1 $x2 // invalid
```

This is because $x1 is declared with a module scope, and $x2 is declared with a global scope (since visible implies global).

2.5.6 Signal Declarations

Signals are declared in a manner similar to registers:

`.sig [keywords]*` *`name1 name2`* `...`

where

`keywords: Zero or more keywords as described below.`

`names: One or more signal names. You cannot declare a signal whose name matches one of the keywords.`

The keywords define the attributes of the signals being declared, as defined by the following table:

Keyword	Meaning
volatile	If the volatile keyword is present, the lifetime of the declared signals is set to volatile. Otherwise, the lifetime is automatic.
global	If the global keyword is present, the scope of the declared signals is set to global. Otherwise, if the declaration is within a block, the scope is set to block. If it is not within a block, then the scope is set to module.
visible	If the visible keyword is present, the visibility of the declared signals is set to visible.
extern	If the extern keyword is present, the named signal needs to be defined elsewhere (either in this module or another that will be linked in).

Keyword	Meaning
remote	If the remote keyword is present, the named signals must be defined in a different microengine and will presumably be the target of a remote reference (e.g. sending an inter-thread signal). These are resolved by nfld. The remote signal must be declared as visible in the remote microengine to be seen by this microengine.

The namespace for signals is the same as for GPRs, i.e. an alphabetic character followed by zero or more alphanumeric characters. Most particularly, there is no type prefix. This is because there is no reasonable scenario where a macro argument could represent a register or a signal, and the macro has to determine which it is.

The **volatile** keyword would in general be needed if the signal were being generated other than in response to some action within this thread. This might be needed, for example, for an inter-thread signal.

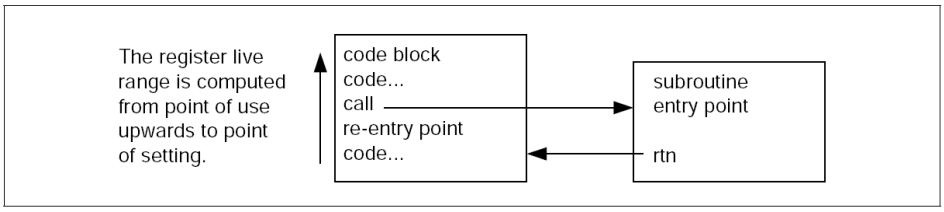
The keyword restrictions are the same as for registers as described in Section 2.5.2.1.

When a signal is consumed via a ctx_arb, all of the signals will be checked for number. For any of these signals, if the source along any path generates a doubled signal[5], then all sources must generate a doubled signal. Each doubled signal will implicitly include the next higher signal in the ctx_arb. For more information on doubled signals, refer to Section 2.5.4.

2.5.7 Register Lifetime Details

For non-volatile registers, the lifetime is computed automatically. Basically, the register is considered “live” everywhere it is used as a source of an operation. This “liveness” is then propagated backwards, up the flow graph until it is terminated by an operation that “sets” or assigns a value to that register. If the register is not volatile and has a block scope, then the live-range is truncated when it leaves the register’s defining block. The exception to this is when it makes a subroutine call/return.

More particularly, when going up the flow graph, the live-range is truncated when the current microword is inside the register’s defining block, the next microword in the graph is not, and the branch was not caused by a RTN. This is illustrated in the figure below. If the current microword is the one labeled “re-entry point”, then the next-microword is not in the code block, but since the next-microword is a RTN, the live-range is not truncated and continues up through the subroutine and back into the code block.



A similar situation exists when extending the live range of a transfer register. (It is extended going down the flow graph from the I/O operation to the completion of the I/O operation, typically a ctx_arb.) In this case (going down

[5]Some I/O operations generate two signals; i.e., the signal specified for the I/O operation must be even, and that signal and the next higher odd signal is also returned. Such a signal is called a “doubled signal” in this document.

the flow graph), the live-range is truncated when the current microword is in the register's defining block, the next microword is not, and the next microword is either not in a subroutine or is in the same subroutine as the current microword. That is, a "subroutine call" is defined as a branch into a subroutine block from someplace not in that subroutine block, and the live-range going down the flow graph is truncated when it leaves the defining block, unless it is a "subroutine call" as defined above.

Consider the case of a block-scoped register that wants to maintain its value outside of its block. This would be similar to a variable in C defined within a function with the static qualifier.

This cannot really be done, since with the exception of subroutine calls, the live-range of a block's coped register is truncated at the block boundaries. The best solution is to make the register global in scope. However, this would fail if it were being done inside a macro and the macro was invoked more than once. In that case, about the best you can do, is to pass in to the macro a unique name for that global register.

2.5.8 Use of REMOTE Keyword

Registers are defined by the code for the microengine in which they are located. Other microengines (e.g., those doing a neighbor-write or reflector operation) reference these via the "remote" keyword.

Similarly, signals are defined by the user or recipient of the signal (e.g., the microengine doing the ctx_arb). Other microengines that wish to send a signal (e.g., via a CSR write) would declare these signals with the "remote" keyword. The only valid operation on a remote signal is taking the address of it. It is never valid to ctx_arb on a remote signal, since that signal does not exist in the current microengine.

Note also that to successfully resolve the remote register or signal, it must be declared as "visible" in the remote microengine.

This is illustrated in the following examples:

Example 2.12. ME0 does a named neighbor write to ME1

ME0 code	ME1 code
`.reg remote n$name` `alu[n$name, …]`	`.reg visible n$name` `alu[xxx, --, b, n$name]`

Note

In this example the neighbor registers are located in the ME1 that reads them and not in the ME0 that writes them.

Example 2.13. ME1 does a reflector read from ME3

ME1 code	ME3 code
`.reg remote $name` `.reg $my_xfer` `// ME 3 is Master ID 7 (3+4)` `immed[addr, ((3 + 4) << 10 \| &remote($name,(__nfp_meid(32, 3))) << 2)]` `// Set the destination island, even if local island` `immed_w1[addr, (32 << 8)]` `ct[reflect_read_sig_init, $my_xfer, addr, 0, 1], ctx_swap[sig]`	`.reg visible $name` `immed[$name, …]`

Note

In this example a register $name is located in Island32 ME3, and it is accessed using normal source/destination references. In ME1, it is declared as “remote” and referenced via the “ct” command.

Example 2.14. ME4 sends a signal to ME2 (thread 1)

ME4 code	ME2 code
`.sig remote rsig` `immed[addr, ((2 + 4) << 9 | 0 <<6| &remote(rsig,(__nfp_meid(32, 2)))<<2)]` `immed_w1[addr, (32 <<8)]` `ct[interthread_signal, --, addr, 0, 1]`	`.sig visible rsig` `ctx_arb[rsig]`

Note

In this example the signal is declared and used normally in ME2 (except that it is declared “visible”), and it is referenced via the address operator in ME4.

2.5.9 Address Operator

There is a need to be able to take the address of transfer registers and signals (the “address” of a signal is the signal number associated with it). This is needed for registers and signals defined locally (i.e. non-remotely) and for ones declared remotely.

The address of a register is a “long-word address” (or equivalently a “register number”). In other words, all low-order bits of the address are significant, and the addresses of the first few registers are 0, 1, 2, etc. (not 0, 4, 8). The “address” of a signal is its signal number, in the range of 1…15.

When the address operator is used with a transfer register, the context relative address is returned. To get the absolute register address, the context offset must be added to the relative address. Note that in IXP indirect reference mode, assuming only $s and $$d are declared transfer registers for a specific context, &$s and &$$d are 0, whereas for NFP indirect reference format mode these two transfer registers will be in sequence and one of them will have an index (&$x) of 1. The code segment below illustrates loading the absolute register address into the T_INDEX

CSR for 8 context mode in NFP indirect reference mode. The T_INDEX and T_INDEX_BYTE_INDEX registers operate with the absolute register address.

```
;Get relative transfer register address & shift left 2 for location of register
alu[rel_reg_address, --, B, (&$txd_p0<<2)]

;read current ctx number (bits 0..2)
local_csr_rd[active_ctx_sts]
immed[active_ctx, 0]
;Mask out the remaining bits
alu[active_ctx, active_ctx, AND, 0x7]
;shift left 5 for 32 regs in each context, shift left 2 for location of register
;index in T_INDEX.
;For IXP compatible indirect reference format mode, this should be 4 for 16 regs
;per context and also shift left 2 for T_INDEX value.
alu_shf[abs_reg_address, rel_reg_address, OR, active_ctx, <<7]
local_csr_wr[T_INDEX, abs_reg_address]
```

When registers and signals are defined locally, this is done by prepending an “&” to the register/ signal name; e.g., `&$xfer, &sig_name`.

When registers and signals are declared remotely, the reference is defined with the following pseudo-function:

```
&remote(name, MEID, ...)

name: Name of remote register/signal
MEID: Number of remote microengine
```

Note that MEID must be a numeric constant. To produce valid MEIDs, use the preprocessor functions __nfp_meid and __nfp_idstr2meid. For example, &remote(name, __nfp_meid(35, 11)) will refer to a register in microengine 11 in island 35.

In the case where a register/signal is declared in a different block (typically corresponding to a different context), it is also considered “remote”. It is referenced with the “&remote(name)” construct as described above, but with no MEID specified.

If more than one MEID is specified, a check will be made to make sure that the specified register/signal has the same address in all of the listed microengines.

In all of these cases, the reference is usable where a constant would be used. Depending upon the usage, you may or may not need to declare registers or signals that have their address taken volatile.

These operators (as well as imported variables) can also be used in constant expressions.

Constant expressions have enclosing “()”. In order to use the address operator in a constant expression, the register or signal must first be declared. Outside of constant expressions, the address operator will implicitly declare the register or signal (if register declarations are not required).

In the context of constant expressions, the address of a remote (or local) neighbor register can be taken. When taking the address of a remote neighbor register, the microengine number of the neighbor does not need to be

specified, and so the &remote() function is not needed. That is, to take the address of a remote neighbor register, it is sufficient to say “&n$reg”.

Note also that the address of a local neighbor register can be taken within a constant expression but not outside of a constant expression. Constant expressions have enclosing “()”, so the following is valid:

```
.reg n$local gpr
immed[gpr, (&n$local)]
```

but the following is not:

```
.reg n$local gpr
immed[gpr, &n$local] ; Error: must be within constant expr
```

2.5.9.1 Accumulating Results for ctx_arb[--]

To use the `ctx_arb[--]` feature, you need to be able to accumulate a subset of certain signals into a register. To do this, you need a way to get the size of the signal and way to shift it to the correct position. To support this, there is a built-in function for constant expressions

`mask(sig)`

which will expand to 1 for normal signals and 3 for doubled signals. This would typically be used to generate a bit-mask to be written to active_ctx_wakeup_events as illustrated in the example below.

The mask() function behaves differently based on whether the specified signal is active at the time of use (e.g., if the use of mask() is between the I/O and the ctx_arb). If the signal is active, the value of the mask function will be based on whether the active signal is doubled or not. This is because another, unrelated use of the signal may use it in the other sense. However, if the signal is not active at the time that mask() is used, but the signal is only used in a doubled or single sense, the mask() function succeeds. If the signal is not active where mask() is used, and the signal is used in both a single and doubled manner, then an error results.

Additionally, the alu (alu_shf) instruction will support syntax for shifting based on the address of a signal. In particular, the shift token can take any of the following forms:

```
<<&sig_or_reg
<<&remote(sig_or_reg)
<<&remote(sig_or_reg, meid)
<<(constant_expression)
```

This would be used as indicated in the following example (for an explanation of “.io_completed”, see Section 2.5.10):

```
alu[sig_mask, --, b, (mask(sig1)), <<&sig1]
.if (…)
alu[sig_mask, sig_mask, or, (mask(sig2)), <<&sig2]
.endif
.if (…)
alu[sig_mask, sig_mask, or, (mask(sig3)), <<&sig3]
.endif
```

```
local_csr_wr[active_ctx_wakeup_events, sig_mask]
ctx_arb[--]
.io_completed sig1 sig2 sig3
```

If the mask() function is invoked where a signal is live and single, but then that value is later used when the signal is doubled, (or vice versa), the code will fail with no warnings or errors. For example:

Example 2.15. Incorrect usage of mask()

```
sram[read, $x, a,0, 1], sig_done[sig1] ; sig1 is single
alu[sig1_mask, --, b, (mask(sig1)), <<&sig1] ; mask() is 1
ctx_arb[sig1]
…
.if (…)
sram[swap, $x, a,0], sig_done[sig1] ; sig1 is now doubled
alu[tot_mask, tot_mask, or, sig1_mask] ; sig1_mask is 1
; ERROR: tot_mask is now incorrect. It is 1, but should be 3
.endif
local_csr_wr[active_ctx_wakeup_events, tot_mask]
ctx_arb[--]
.io_completed sig1 …
```

2.5.9.2 Examples of Address Operator and Visible/Volatile Signals

The first example is one context (context-0) signaling another (context-1) in the same microengine:

Example 2.16.

```
.if (ctx() == 0)
    .sig remote s
    local_csr_wr[…, (…&remote(s)…) ]
…
.elif (ctx() == 1)
    .begin // begin block for context-1
        .sig visible s
        .while(1)
        …
        .endw
    .end
.elif …
```

In this case, context-0 is using a remote declaration and the “&remote(name)” construct to reference a signal declared within the same microengine, but which is local to a different block.

Note

If context-1 were in a different microengine, the picture would look almost identical, except that the reference would be “&remote(name, MEID)”.

In either case, there is a problem if you want to use the same visible signal in different, independent contexts. There are several ways in which this could be addressed:

1. Use three different names, e.g., “s1”, “s2”, etc.
2. Use a .begin/.end block that included multiple contexts (but preferably not the unnecessary contexts, as this would defeat the purpose of making the visible signal non-volatile).
3. Make the signal “s” volatile and global, and just live with wasting a signal in the other contexts.

Another example where this mechanism would be useful, is in the case where one microengine (the master) is initializing some data structure and wants to hold off the other microengines (the slaves) until the structure is initialized. A typical way to do this, is to have the master microengine send an inter-thread signal to the slave microengines. This signal would have to be visible, but it would only be used during initialization. This could be handled in slaves as:

```
.if (ctx() == 0)
    .begin
    .sig visible wakeup
    lab#: br_!signal[wakeup, lab#]
    .end
.endif
```

In this case, after the begin/end block was exited, the physical signal allocated to “wakeup” would be available for reuse.

The master would do something like:

```
.reg remote wakeup
immed[addr, ((0 + 4) << 9 | 0 <<6 | &remote(wakeup,(__nfp_meid(32, 0)))<<2)]
immed_w1[addr, (32 <<8)]
ct[interthread_signal, --, addr, 0, 1]
immed_w0[addr, ((1 + 4) << 9 | 0 <<6 | &remote(wakeup,(__nfp_meid(32, 1)))<<2)]
ct[interthread_signal, --, addr, 0, 1]
…
```

2.5.10 Register Allocator Directives

There are several areas where limitations in the assembler cause it to make overly pessimistic assumptions. There are two ways that this can be dealt with:

- The simple approach is to do nothing. The assembler should always “do the right thing” or do the safe thing. The only two problems are that there may be excessive warnings and the register usage may not be optimal. The problems with the warnings can be handled via the warning mechanisms. If the register usage is a problem, then the following approach can be used.
- There is a series of directives (as described in this section) that allows the advanced user to more carefully tune the register allocation process and possibly achieve better register utilization. The downside is that use of these directives will be less-obvious to a casual user and would require a greater understanding of the register allocation process.

There are four areas where limitations of the register allocator may cause spurious warnings or non-optimal register allocation. These are described in the following four sections:

- Register Used Before Being Set (Section 2.5.10.1)
- Determining when I/O Operations Complete (Section 2.5.10.2)
- Using Registers Indirectly (Section 2.5.10.3)
- Use of `.set` and `.use` with Transfer Registers (Section 2.5.10.4)

2.5.10.1 Register Used Before Being Set

If there is no path that sets a register before using it, a low-level warning is issued (“Used before set”). If there were some paths that set it and some that do not, then a high-level warning is issued (“Maybe used before set”).

A typical example of this second kind would be:

Example 2.17.

```
.if (condition_1)
    immed[reg, 0]
.endif
…
.if (condition_1)
    alu[…,reg]
.endif
```

The assembler does not know that the second conditional is taken only if the first is already taken. It assumes that the first conditional can be skipped and that the second can be taken. This results in the register being used before being set.

To address this in the case that the warning level is set to include this warning and you have verified that there actually is no problem (i.e. in the above example, if the two conditions were different, there would be a real problem), you could use the .set directive.

```
.set reg1 reg2 …
reg1 reg2 …:One or more registers to be “set”.
```

This directive looks like an assignment to the register allocator (thus getting rid of the “used before set” warning), but it does not generate actual code.

Correct behavior can always be achieved by placing the .set directive at the start of the affected block. More optimized usage would place the directive immediately before the conditional causing the problem. Using the above example, the following would always be valid:

```
.begin
.reg reg
.set reg         // .set directive at start of block
…
.if (condition_1)
    immed[reg, 0]
```

```
.endif
…
.if (condition_1)
    alu[…,reg]
.endif
```

But the allocator may use the registers more effectively if it is placed just before the conditional:

```
.begin
.reg reg
…
.set reg         // .set directive before conditional
.if (condition_1)
    immed[reg, 0]
.endif
…
.if (condition_1)
    alu[…,reg]
.endif
```

It would be an error to place it after that conditional (i.e. between the two conditionals). In this case, you would be "assigning" a value to the register after having done so with an actual assignment. In this case, it would appear to the register allocator that the register was set at the .set directive and then used, and hence the setting of the register using the immed instruction was irrelevant (i.e. that value was never used). It would therefore be free to clobber the value of reg between the immed and the .set directive.

This directive would also be required if the setting of transfer registers were done using the index register. In this case, the assembler would not know which registers were actually being set. Presumably you know and can use the .set directive to tell the assembler.

Note also that this directive should not be placed after register declarations as a matter of course, because this may mask real bugs. This directive should only be used after you have determined that the warning is caused by a limitation of the assembler and not due to a potential problem with the code.

A similar directive, .set_sig can be used with signals:

```
.set_sig sig1 sig2 …
sig1 sig2 …: One or more signals to be "set".
```

This would be used in the case where the signal is being generated other than by an I/O operation.

Refer to Section 2.5.10.4 for additional information on using the .set directive.

2.5.10.2 Determining when I/O Operations Complete

The assembler has to determine when I/O operations, particularly write operations complete. The assembler attempts to determine when a write I/O operation completes, but it may not be able to do so in all circumstances. If it cannot, it will report an error.

To indicate that an I/O operation is completed, you would use one of the .io_completed directives:

```
.io_completed name1 name2 …
.io_completed_type type

name1 name2 …:One or more signals or transfer registers whose I/O operations
  have been completed.
type: Memory type (listed below) SRAM, MEM, ILLA, ARM, CRYPTO, NBI, PCI
```

The directive ends the live-ranges of referenced I/O operations. An operation is referenced if at the directive it is not completed and any of:

- The directive lists a signal used by the operation (or on a following operation using the same queue).
- The directive names the type of the operation.
- The directive lists any of the registers involved with the operation.

Ending the operation ends the live-ranges of all parts of the referenced operation (e.g., the transfer registers and the signal).

```
Ending the operation in this way can have have unintended results. The following example
   code is confusing and should be avoided:

.reg $X $y
.xfer_order $x $y
mem[read, $x, a, 0, 2]. sig_done[s]
. . .
.io_completed $y
```

One common case where you need to indicate where an I/O operation is completed is the same scenario as described in Section 2.5.10.1.

For example, consider:

```
.if (condition_1)
    mem[write, $xfer, …], sig_done[sig]
.endif
…
.if (condition_1)
    ctx_arb[sig]
.endif
```

The assembler does not know that after the I/O operation has been issued, the ctx_arb must be executed. It assumes that the ctx_arb could be skipped. The solution is to place after this code:

```
.io_completed sig
```

Note also that there would have to be a “.set_sig sig” before the first conditional. Otherwise, you would get a warning that the signal might be used before it was set.

2.5.10.3 Using Registers Indirectly

A similar but slightly different problem occurs when read transfer registers are referenced via the index register. The issue is that the assembler does not know which registers are addressed by the index register, so it does not know when the read values are actually used. You need to tell the assembler that the transfer registers in question are being used up to the point when they are not. This can be done with the .use directive:

```
.use reg1 reg2 …
reg1 reg2 …: One or more registers whose use has been completed.
```

Similar to the .set directive, this appears as a use of the named registers, but without generating any microwords.

Example 2.18. Indirect Usage

```
.xfer_order $1 $2 $3
mem[read, $1, addr1, addr2, 3], ctx_swap[sig]
// set up transfer register index register (not shown)
alu[…, *$index++] // uses $1
alu[…, *$index++] // uses $2
alu[…, *$index++] // uses $3
.use $1 $2 $3
```

Since those three registers are “used” by the .use directive, the lifetime of those registers will extend from the I/O operation to the .use directive, encompassing the uses from the index register.

Refer to the next section for additional information on using the .use directive with transfer registers.

2.5.10.4 Use of .set and .use with Transfer Registers

The .set directive can be used with both “read” and “write” transfer registers, but “.use” can only be used with “read” transfer registers (but “.use_wr” can be used with “write” transfer registers, see below for details).

More particularly, if .set is applied to a R/W (both) transfer register, it is considered as “setting” both the read register and the write register. If .use is applied to a R/W (both) transfer register, it is considered as a use only of the read register. If .use is applied to a register declared as “.reg write”, then it generates an error. Variants on .set and .use exist:

```
.set_rd reg1 reg2 …
.set_wr reg1 reg2 …
.use_rd reg1 reg2 …
.use_wr reg1 reg2 …
```

These variants can only be used with transfer registers of the appropriate type (e.g., “.set_rd” cannot be used with registers that are not transfer registers or which have been declared as “write”). An example of when you would use these is:

```
sram[read, $X, a,0, 1], ctx_arb[s] ; reads (sets) $X.read
… ; area-1
```

```
…
… ; set t_index to point to $X
immed[*$index, 0] ; sets $X.write
.set_wr $X
sram[write, $X, a,0, 1], ctx_arb[s] ; writes (uses) $X.write
…
alu[--,--,b,$X] ; use $X.read
```

The problem is that if one uses “.set” rather than “.set_wr”, it will “set” both the read and write transfer registers, so it will appear that the read of $x is useless, and the assembler is free to clobber $X within area-1. This is avoided by using “.set_wr”.

2.5.11 GPR Spilling

The basic approach for spilling GPRs to local memory, is to map all of the virtual registers into physical registers. If the number of physical registers needed exceeds the available number, then spilling is required. Then, some number of the physical registers are selected to be located in local memory rather than in an actual GPR.

The steps that need to be taken are:

1. Determine which “physical registers” are able to be located in local memory.
2. Select a sufficient number of these that the remaining ones can fit into the actual physical registers.
3. Assign the spilled registers to local memory addresses.
4. Modify the source code to use local memory.

If relative GPRs are spilled, then a non-spillable relative GPR is used to store the address of a local memory block to be used by that context. Absolute GPRs are not spilled.

Spilling will for each instruction that requires spilling select a local memory index to use. By default spilling will first try to use the local memory index *l$index1. If *l$index1 is in use by user code the assembler will then fall back to *l$index0. The option -spill-def-ind=0 can be used to change this default behaviour so that the assembler tries *l$index0 first and then falls back to *l$index1. For example if the developer will reserve *l$index0 for spilling and only use *l$index1 in their code, using the -spill-def-ind=0 option will allow the assembler to find a more optimal spilling solution.

Due to certain implementation restrictions, the assembler may not be able to use spilling to always resolve a “too many GPR” condition. Also, you can use the #pragma optimize mechanism (see Section 2.8.3) to further mark portions of the code as unmodifiable.

While the assembler attempts to be efficient in modifying the code, it does not understand the program nearly as well as you. While having the assembler handle spilling is convenient, it is likely that you could use a similar approach to resolve the issue with less execution overhead.

By default this option is disabled. This is so that the user has to make an explicit request before the assembler starts inserting instructions into the user's program.

2.5.11.1 Codeless GPR Spilling

In the codeless spilling mode (specified by the *-spilling-no-code* option) spilling will not add any instructions to your code for spilling. Some instructions will be modified to work with the local memory (spilled) registers instead of actual gprs. The advantage of this spilling mode is that it allows you to access additional registers without affecting the performance of your code. The drawback is that since the local memory pointer is never moved for spilling (which would require code to be added), this mode of spilling is limited to 16 registers (the amount of registers that can be accessed without moving the localmemory pointer). Note that in many cases when there are more than 8 registers that need to be spilled, spilling could still fail. This is due to the fact that some registers may be restricted and that only 8 restricted registers can be accessed via the lm index.

2.5.12 Lifetime Out-Of-Register Errors

An issue that a user may encounter is that if their source code requires too many registers, it can be difficult to figure out where the problem is occurring, and hence where changes should be made to reduce the number of required registers.

To help address this issue, there is a command-line flag “-lr” which dumps the register lifetime information into a new file with a “.lri” extension. The reason that a new file is being used rather than using the .uci file is that if register allocation fails, a .uci file is not produced.

The `.lri` file basically consists of the raw source (post preprocessor) along with embedded “directives” giving the registers live at each line.

The directives are:

`.%live_regs cnt` *`addr gpr sr_xfer sw_xfer dr_xfer dw_xfer sig`*

This gives the count of live registers of the different types for the associated uword. The *addr* field gives the corresponding uword address (if the source file assembles successfully and the optimizer is not used). The next six numbers then give the counts in terms of relative registers. The count of GPRs can be non-integral due to absolute GPRs (since an absolute GPR occupies the space of either 1/8 or 1/4 that of a relative register, depending on how many contexts there are). “*sr_xfer*” refers to SRAM-read transfer registers, “dw_xfer” refers to DRAM-write-transfer registers, etc.

```
;%live_regs cnt addr gpr $R $W $$R $$W sig
```

This is a “comment” line that indicates what the different numbers from the “.%live_regs cnt” directive mean. It is particularly useful (as described later) when the counts are imported into a spreadsheet.

`.%live_regs type` *`addr name1 name2 name3`* `...`

This gives the actual list of live registers. The type field contains the same labels as from the “;%live_regs cnt” line; i.e. “gpr”, “$R”, “$W”, “$$R”, “$$W”, and “sig”. There is also a “@gpr” type, whose line lists all of the absolute GPR registers. Since absolute registers are essentially always live, rather than repeating this list throughout the file, the absolute GPRs are just listed once, at the beginning.

The *addr* field is as described above for ";%live_regs cnt". The *names* are the names of the live registers. If the list is empty for a particular type, then that line is suppressed.

Example 2.19. A `.lri` File

```
;--------------------------------------
This is a separator between lines to make the file easier to read.
An example output might be:
;%live_regs cnt addr gpr $R $W $$R $$W sig
.%live_regs @gpr 0 @g1
; lri_regs.uc: test of lri live range values
.reg tmp ; module
.reg global g
;------------------------------------------------------------
.%live_regs cnt 0 0.125 2 1 0 0 0
.%live_regs $R 0 $z $i2
.%live_regs $W 0 $z
nop
;------------------------------------------------------------
.%live_regs cnt 1 0.125 2 1 0 0 0
.%live_regs $R 1 $z $i2
.%live_regs $W 1 $z
immed[tmp, 0]
```

Some miscellaneous comments about the listing:

- The number of registers listed in ".%live_regs xxx" lines may not match the count in the ".%live_regs cnt" lines for several reasons: for GPRs, the count also includes absolute GPRs, listed once at the beginning. Doubled signals increment the count by 2, but are listed once.
- Due to implementation reasons, there may be a block of text with a separator and ".%live_regs cnt", but which does not contain a real uword. For example:

Example 2.20. A .lri File without a Real Microword

```
;------------------------------------------------------------
.%live_regs cnt 2 0 0 0 0 0 0
.if (x == 0)
;------------------------------------------------------------
.%live_regs cnt 2 1 0 0 0 0 0
.%live_regs gpr 2 x
alu[--,--,B,x]
```

- Due to scoping, the same register may be repeated in the list. There could be several registers named "tmp" defined in different nested scopes that are all live at the same time.
- The counts should be taken as indications of where the problem may lie and not as a hard-andfast indicator of allocation success. For example, at the worst-case, there may be only 31 GPRs live, but it may still fail allocation due to A/B bank issues.

One of the most useful things to do with the `.lri` file is to filter it based on ".%live_regs cnt", load it into a spread sheet, and then plot the counts. For example:

```
grep "%live_regs cnt" <file.lri >file.dat
```

You can read that file into a spreadsheet, graph the data, and get a plot that looks like Figure 2.2.

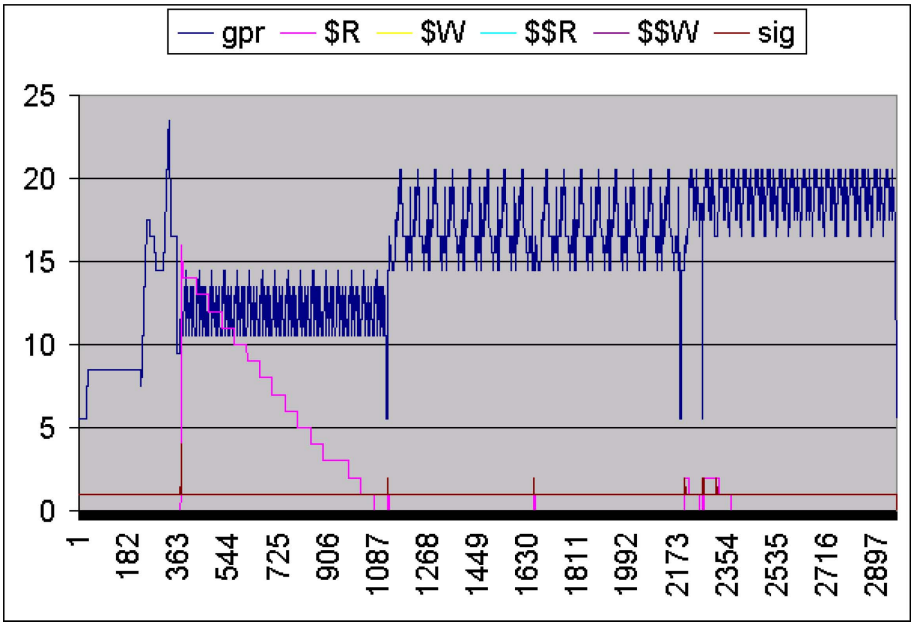


Figure 2.2. Lifetime Register Spreadsheet

From this, one can see that the GPR usage peaks around line 327 with a value of 23.5. Since this project was running in 4-ctx mode, there are 64 GPRs available, so about 40 of them should be available. If, however, the peak at line 327 exceeded 64, then register allocation would probably fail. In that case, by looking at what registers were live in that region, you could hopefully identify some that could be made not live. Possible ideas for coping with excess register pressure would include:

- Rewriting the code to use fewer registers. For example, re-computing a value rather than using a value computed earlier. Similarly, delaying a calculation until just before a result is needed may help.
- Using absolute GPRs rather than relative ones where the value is not needed beyond a context arb, or where the value is the same for all contexts. Absolute GPRs are essentially always live[6], so that an absolute virtual GPR corresponds to an actual physical register. Thus, absolute GPRs should be used for data with some permanence rather than for short-lived temporaries.
- If the neighbor registers are not being used as a FIFO, they can become the repository of constants or pseudo-constants (values computed during startup but which do not change during the main loop).
- Using local memory to replace GPRs. This is particularly effective if one of the local memory index registers is available.
- It is also possible that the live-ranges of some registers are not being computed correctly due to incorrect use or lack of use of some of the directives. This would be indicated if a register was listed as live when it really was not. An example of this would be if a register was used, and then in a later section it was conditionally set and then later conditionally used (i.e. “correlated conditionals”). Without an appropriate “.set” directive, the register would be considered live from the first set to the last use, when in reality, it wasn’t “live” between the two sets

[6]Absolute registers are typically used to communicate between threads. As such, most absolute registers are used by multiple threads. This means that most absolute registers need to be considered “used” whenever there is a context arb. This would make them live most of the time, so as a simplifying assumption, the assembler makes them live always.

of references. In this case, the code produced would not be incorrect, but it may use more registers than necessary. Putting in the appropriate ".set" might reduce the register pressure.

- Another possibility is that there is an outright bug in the source code, and that by fixing the bug, the number of necessary registers would go down.

2.5.12.1 Understanding the LRI Results

In the absence of subroutine calls, the number of live registers gives a good approximation to the number of physical registers that will be needed. This is because normally (i.e. in the absence of subroutine calls), two virtual registers that are live at the same time interfere with each other, and so they need to be allocated to different physical registers. For example, if you had 27 live registers, these 27 registers would interfere with each other, and you would need at least 27 physical registers so that the allocation succeeds.

There is a noticeable exception however: subroutine calls. A virtual register that is live across a subroutine call, but not used in the subroutine will be live within the subroutine.Such a register will interfere with registers used within the subroutine, but they will not interfere with other registers that are live merely across other subroutine calls.

Under these circumstances, the number of live registers (within the subroutine) will exceed the number of physical registers required (and perhaps exceed the number of physical registers in the hardware). This does not indicate an allocation error.

For example, consider the following code:

```
#macro one_call
.begin
.reg x
immed[x, 0]
br[sub#], defer[1]
load_addr[raddr, ret#]
ret#: alu[--,--,b,x]
.end
#endm
.reg global raddr
#repeat 100
one_call
#endloop
stall#: br[stall#]
sub#: nop
nop
rtn[raddr]
```

This code just creates a local virtual register, "x", and makes it live over a subroutine call. This is repeated 100 times. The subroutine itself, does nothing.

This can be allocated with two physical registers: one for "raddr" and one for the "X" registers. The X registers do not interfere with each other, and so they can be all be allocated to the same physical registers. However, they are all live within the subroutine. A portion of the LRI file looks like:

```
.%live_regs cnt 398 1 0 0 0 0 0
.%live_regs gpr 398 x
```

```
load_addr[raddr, M099_ret#]
;-----------------------------------------------------------------
.%live_regs cnt 399 1 0 0 0 0 0
.%live_regs gpr 399 x
M099_ret#: alu[--,--,b,x]
.end
.end_macro_ref one_call
;-----------------------------------------------------------------
.%live_regs cnt 400 0 0 0 0 0 0
stall#: br[stall#]
;-----------------------------------------------------------------
.%live_regs cnt 401 101 0 0 0 0 0
.%live_regs gpr 401 x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x raddr
sub#: nop
```

From this, it can be seen that outside of the subroutine (e.g., at uwords 398-400) there is 0--1 live registers. This is what should be expected. However, within the subroutine (e.g., at uword 401), there are 101 live registers (each of the X's and raddr). Thus, one would be tempted to say that this example needs 101 physical registers, and so register allocation should fail; when in fact all that are needed are two physical registers, and allocation easily succeeds.

The two key points here are:

- If register allocation fails, the assembly will generate an error. Conversely, if there is no error, then register allocation was successful, despite what the LRI file may suggest.
- The number of live registers at different uwords is indicative of where register pressure is greatest, but the values within the body of subroutines may be misleadingly high.

2.5.12.2 Transfer Register Lifetimes

It can be difficult for the assembler to compute when the lifetime ends for a write transfer register. The reason for this is that the instruction that ends the lifetime may only indirectly be related to the instructions using the register.

The same logic is used to extend the lifetime of read transfer registers, although typically this is not necessary as the registers are used after the I/O operation completes. It might be necessary if an I/O operation reads three words and only used the first and third. Then the lifetime of the second one will have to be extended.

The rule for transfer register lifetimes is that the lifetime extends beyond the I/O instruction until one of the following:

- An I/O instruction to the same queue generates a signal, and then that signal is consumed.
- The lifetime is explicitly ended via the .io_completed directive (see Section 2.5.10).
- The end of the defining block is reached.

Note

Based on the I/O instruction, the queue may or may not be known. For MEv2, the queue is almost always unknown, with a few exceptions detailed below. Note also that the word “queue”

in this context refers to a logical queue, which may or may not represent a physical queue. For example, two operations that are stored on the same physical queue but which are processed in such a way that they may become reordered may be represented for the purpose of this section as being on two different queues.

2.6 Arithmetic Notation Feature

The Assembler supports the use of arithmetic notation for simple expressions that can be assembled into a single ALU instruction or into one or two IMMED instructions. The format is:

```
register = expression
```

The instruction must be on a single line, although you can always break it into multiple lines by using the line continuation character, for example:

```
x = y + \
z
```

If you prefer a C-language syntax, you can end the expression with a semicolon, although this is just a special case of a post-comment. The statement can be preceded by labels (either on the same line or on previous lines) and by comments on previous lines; for example:

```
; this is a comment
lab1#:
lab2#: x = 1 ; post-comment
```

A caveat is that there must be some white space between the register name and the ‘=’character; for example:

```
x=1 ; This is invalid as it is interpreted as the opcode "x=1"
```

The white space is necessary because the Assembler accepts some opcodes that contain the ‘=’ characters. The ‘=’ character may or may not be followed by white space.

In the case where the expression evaluates to a constant, it will be assembled into one or two IMMED instructions (actually it is an IMMED/IMMED_W1 pair). If the constant (taking into account shifts) can fit into a single instruction, only one IMMED instruction is used.

If a single instruction is not sufficient for the constant, and the destination is not a general purpose register (GPR) (for example: a transfer register or a neighbor register), then three instructions are generated—the first two to place the constant in a GPR and the third to copy the result to the real destination.

In the case where the expression is a constant expression that does not evaluate to a constant at the start of the assembly process (for example: if the expression includes an imported variable or references the address of a register or a signal), two instructions are used for safety (that is, all 32-bits of the register are set).

If two instructions are generated, a default level-3 warning (Number 5165) is generated. The warning is generated to alert the programmer that what appears to be a single instruction is actually two instructions. The warning might cause problems; for example, when someone codes:

```
br[lab#], defer[1]
x = 0x12345678
```

To avoid this warning, you can always use `#PRAGMA WARNING` to disable it. On the other hand, if you never want the Assembler to generate two instructions from one arithmetic instruction, you can use the same mechanism to turn this warning into an error.

If the expression does not evaluate to a constant, then the Assembler tries to assemble it into a single ALU or ALU_SHF instruction. Examples include:

```
x = y              ; alu[x, --, b, y]
x = ~y             ; alu[x, --, ~b, y]
x = ~(x << 2) | y ; alu[x, y, or, x, <<2]
```

The Assembler tries to resolve constant sub-expressions as early on in the expression tree as possible. Therefore, if the code reads:

```
x = x &~ 1 ; ERROR, this will not assemble
```

then it is not assembled into "`alu[x, x, and~, 1]`". Rather it is assembled into "`alu[x, x, and, (~1)]`" which generates an error since the constant (~1) is too large.

If a shift is allowed but not used, and the constant is too big to fit into the ALU operand, but a shifted version will fit, then the Assembler inserts the appropriate shift. Because of this, "x = x | (1<<20)" assembles correctly as "alu[x,x,or,1,<<20]".

Note also that the unary "~" operator has a higher precedence than

"<<", so that

```
x = x &~ y<<2 ; ERROR, this will not assemble
```

will not assemble since this is interpreted as (x & ((~y)<<2)). In this case, the programmer probably intended:

```
x = x &~ (y<<2) ; This is the correct version.
```

Note that since this is being done by the Assembler itself rather than the preprocessor, it can distinguish between imported variables and registers. For example:

```
.reg x y
.import_var z
x = y ; generates an ALU to copy a register
x = z ; generates an IMMED/IMMED_W1 pair
```

If the expression cannot be converted to either an IMMED or ALU, an error is generated. Note that in some cases of invalid expressions, the expression may be converted to an IMMED or ALU, and then the IMMED/ALU generates the actual error.

2.7 Assembler Optimizer

The optimizer reduces execution time by performing the following optimizations:

1. **Branch target optimization:** Branches to unconditional branches are modified to directly branch to the ultimate branch target. This optimization is particularly useful for nested conditional statements. In addition, branches to consecutive addresses are removed. This optimization is useful when a conditional statement evaluates to false at assembly time.
2. **Defer shadow filling:** The defer shadows for branches and context swapping instructions are filled by moving instructions down into their defer shadows.
3. **NOP removal:** Unnecessary NOPs (NOPs that are not required to avoid violating timing constraints) are removed. For example, NOPs can be used to hide the latency between a local_csr_wr to an index register and the use of that index register. If the use of the index register were to be moved in a previous optimization step, then one or more of the NOPs might become unnecessary and would be removed.
4. **NOP replacement:** NOPs that cannot be safely removed in the above optimization step will be replaced by moving instructions down.

The optimizer is enabled via a Programmer Studio dialog box or command-line option. Once enabled, the optimizer can be disabled and re-enabled for specific code sections by using the #pragma optimize directive.

The following example illustrates each type of optimization.

Un-optimized Code:

```
label#:
.if (x==y)
   .if (x==1)
      immed[z,0]
   .endif
.elif (x>y)
   immed[z,1]
.endif

#define i 1
#define j 2

.if (i==j)
alu[z,z,+,1]
.endif

alu[z,z,+,2]
local_csr_wr[ACTIVE_LM_ADDR_0,0]
nop
nop
nop
alu[z,z,+,*l$index0]
```

```
crc_be[crc_32,z,z]
nop
crc_be[crc_32,z,z]
br [label#]
```

Optimized Code:

```
.if (x==y)
label#:
alu[--,l0000!x,-,l0000!y]
bne[l000_01#]
    .if (x==1)
    alu[--,l0000!x,-,1]
; BRANCH TARGET OPTIMIZATION: changed branch label from l001_01# to l000_end#.
       bne[l000_end#]
      .endif
.elif (x>y)
    l001_01#:
    l001_end#:
br[l000_end#], defer[1]

; BRANCH LATENCY FILL OPTIMIZATION: the microword below was "pushed" down
; position
       immed[l0000!z,0]
l000_01#:
alu[--,l0000!x,-,l0000!y]
ble[l000_02#]
   immed[l0000!z,1]
.endif
.if (i==j)
; BRANCH TARGET OPTIMIZATION: removed branch to next address.
; br[l002_01#]
; The following microwords are unreachable and have been commented out
; alu[l0000!z,l0000!z,+,1]
; End commenting out unreachable code
.endif
l000_02#:
l000_end#:
l002_01#:
l002_end#:
local_csr_wr[active_lm_addr_0,0]
nop
nop
; NOP REPLACEMENT OPTIMIZATION: the microword below was "pushed" down 4
;positions
alu[l0000!z,l0000!z,+,2]
alu[l0000!z,l0000!z,+,*l$index0]
crc_be[crc_32,l0000!z,l0000!z]
; NOP OPTIMIZATION: removed unnecessary NOP.
; nop
br[label#], defer[1]
; BRANCH LATENCY FILL OPTIMIZATION: the microword below was "pushed" down 1
; position
crc_be[crc_32,l0000!z,l0000!z]
```

2.8 Assembler Directives

2.8.1 Supported Assembler Directives

Supported Assembler directives are described in Table 2.11:

Table 2.11. Assembler Directives

Type	Directive	Arguments Expanded	Description
Processing			
Assembler Loops	#for	No	Repeat following lines based on a const expression.
	#repeat, #while	Yes	Repeat following lines based on a const expression.
	#endloop	N/A	End repeated lines.
Assembler Macros	#macro	No	Start defining a macro.
	#endm	N/A	Finish defining a macro.
Conditional Assembly	#ifdef, #ifndef	No	Conditionally skip following lines.
	#if, #elif	Yes, including “defined(name)”	Conditionally skip following lines based on a const expression.
	#else, #endif	N/A	Conditionally skip following lines.
Error Reporting	#error		Displays a message and optionally aborts processing.
	#warning		Displays a warning message without aborting processing.
	#info		Displays a info message without aborting processing and without the macro stack trace.
File Inclusion	#include	No	Start reading lines from another file.
Token Replacement #define	#define	No	Define an expandable token.
	#define_eval	Yes	Define an expandable token to a const expression.
	#undef	No	Undefine an expandable token.
Structured Assembly	.if, .elif	Yes	Generate Microengine instructions that are executed at runtime.
	.if_unsigned, .elif_unsigned	Yes	
	.else, .endif	N/A	
	.while	Yes	

Type	Directive	Arguments Expanded	Description
	.while_unsigned	Yes	
	.endw	N/A	
	.repeat	N/A	
	.until	Yes	
	.until_unsigned	Yes	
	.break, .continue	Yes	
Assembler			
Import Variable	.import_var		Defines a set of symbolic names that will be imported by the Linker.
Local Regions	.begin		Defines a region of code and a new set of registers that are only visible within that region.
	.end		
Register declaration	.reg		Declares register symbolic names which will be allocated to physical registers by the Assembler.
Initialization	.init .init_ctx		Initializes all or part of a block of memory or a register.
Manage register and signal usage warnings	.set .set_sig .set_rd .set_wr		These directives generate no code, affecting “used before set” warnings for registers and signals.
Extend or define register lifetime	.use .use_rd .use_wr		These directives generate no code, but tell the Assembler registers in question are being used up to the point where they are not.
Signal completion of I/O operation	.io_completed .io_completed_type		These directives generate no code, telling the Assembler that an I/O operation has been completed.
Manual Register & Signal Specification	.addr		Manually allocate registers or signals.
Optimization Directives	#pragma optimize		This directive provides precise control over the optimizations performed by the Assembler.
Subroutine Definition	.subroutine .endsub		Define a region of code containing a subroutine.
Memory Allocation	.alloc_mem .global_mem		Allocate a block of memory for a shared resource (i.e. EMEM, IMEM, CTM, CLS, or local memory (LMEM)).
.local_mem			
Local Memory Mode	.local_mem0_mode		Set local memory mode to either abs or rel

Type	Directive	Arguments Expanded	Description
	.local_mem1_mode		
.local_mem2_mode			
.local_mem3_mode			
Number of Contexts	.num_contexts		Set context mode for the microengine to 4 or 8 threads.
Neighbor Mode Directive	.init_nn_mode		Set the initial NN_MODE in the CTX_ENABLE register, either “neighbor” or “self”. If two different values are specified, an error results. If no value is specified, the default value is "neighbor". Note that this is handled by the loader and does not generate any microcode.
Transfer Order	.xfer_order		Define an ordering of transfer registers.
Warning Directives	#pragma warning		This directive provides detailed control over issuing of warnings by the Assembler.
User Data	.app_metadata		Inserts arbitrary text into the `.list` file. The text can be retrieved using a Loader Library API function.
Byte ordering	.endian		Set byte ordering for all following command instruction. The default mode is “big” endian.
Init-CSR	.init_csr #pragma init_csr		These directives declare CSR initializations.
Hierarchical Resource Allocation	.declare_resource .alloc_resource		These directives declare statically allocated generic or memory backed resources pools and allocate resources within those pools.
Assertions	.assert		Specifies link time assertions to be validated by the linker
Memory Unit Queue and Ring Initialization	.load_mu_qdesc .init_mu_ring		These directives define a mmeory unit queue descriptor to be loaded before running code. The `.init_mu_ring` directive is a specialization of `.load_mu_qdesc`.
Linker			
Linker Directives	.image_name .entry .ucode_size		The following directives are passed through the Assembler to the Linker and are listed without comment.

2.8.2 Token Replacement (#define, #undef)

The #define directive causes subsequent instances of the identifier to be replaced with the token string. After this directive, the identifier is referred to as an expandable token. The #undef directive removes the definition for the given identifier.

Macros and expansion tokens share the same name space (i.e., it is illegal to have a macro with the same name as a #define token).

Format:

```
#define identifier token-string
#define identifier(arg1, ...) token-string
#undef identifier
```

Note

There cannot be any spaces between “identifier” and the “(”. This facility is essentially equivalent to that of the C-processor. The expansion of arguments, however, is handled in a similar manner to arguments for macros defined with #macro.

Example:

Note that the following would be valid:

```
#define_eval foo 123
#define_eval foo foo + 123 /* assuming foo is initially numeric */
#define_eval foo abc
#define_eval foo strleft(abcd, 2) /* evaluates to ab */
but the following would not be:
#define_eval foo abc def
```

This is because “abc def” is not a valid expression.

2.8.3 Optimization Directives

The “#pragma optimize” directive allows the various Assembler optimizations to be individually disabled or re-enabled for specified blocks of code. The syntax is as follows:

```
#pragma optimize( "optimization-list", {on | off} )
```

The optimization-list specifies the types of optimizations to enable or restore. An empty list specifies all types of optimizations. Values are:

Optimization Type	Description
b	Fill defer shadows (bubbles).
c	Conditional branch optimization.
n	Optimize NOPs.
t	Optimize branch targets.
d	All of the above execution speed optimizations (b,n, and t), optimize branch targets, fill defer shadows, and replace/remove NOPs. Equivalent to specifying “tbn”.
f	Try to automatically fix A/B Bank conflicts.
s	Try to automatically spill GPRs into local memory.
empty string	Equivalent to specifying all of the above.

Examples:

```
#pragma optimize(“fs”, off)      ;Turn off A/B conflict fixing and spill
                                 ;optimizations
#pragma optimize(“”, on)         ;Turn on all optimizations
```

Specifying “off” marks the instructions following the #pragma optimize directive as not subject to optimization. Specifying “on” marks subsequent instructions as subject to the specified optimizations.

2.8.4 Loops

2.8.4.1 For Loops (#for, #endloop)

Repeats the text-lines with the identifier taking on each of the listed values. The identifier behaves as if it were the target of a #define. That is, it is set at a global scope, overwriting any existing definition. After the loop is ended, the identifier continues to exist with the last value specified in the directive.

Format:

```
#for identifier [arg1, arg2, ...]
text-lines
#endloop
```

Example:

```
#for id[1,2,3]
immed[reg,id]
#endloop
```

Assembles to:

```
immed[reg,1]
immed[reg,2]
immed[reg,3]
```

2.8.4.2 Repeat Loops (#repeat, #endloop)

Repeats text-lines for the indicated number of times. If const-expr evaluates to a negative number, an error results.

Format:

```
#repeat const-expr
...text-lines...
#endloop
```

Example:

```
#repeat (2)
immed[reg,5]
#endloop
```

Assembles to:

```
immed[reg,5]
immed[reg,5]
```

2.8.4.3 While Loops (#while, #endloop)

This repeats text-lines as long as the expression evaluates to a nonzero value.

Format:

```
#while const-expr
...text-lines...
#endloop
```

Example:

```
#define LOOP 3
#while (LOOP > 0)
immed[reg1,5]
#define_eval LOOP (LOOP-1)
#endloop
```

Assembles to:

```
immed[reg,5]
immed[reg,5]
immed[reg,5]
```

2.8.5 Macros (#macro, #endm)

A macro is a series of directives and instructions grouped together as a single command. Macros are identified by an identifier and optional parameters can be passed to the macro for processing.

Format:

```
#macro identifier(arg1, arg2, ...)
...text-lines...
#endm
```

References to macros are defined by:

Format:

```
[label] identifier[arg1, arg2, ...]
```

Examples:

```
#macro test1[param1,param2]
immed[param1,param2]
mylab#:
alu[reg,reg1,+,param1]
br=0[mylab#]
#endm
```

```
l1#: test1[reg2, 5]
l2#: test1[reg2, 6]
```

These two references would expand into:

```
l1#:
immed[reg2, 5]
l1_mylab#:
alu[reg,reg1,+,reg2]
br=0[l1_mylab#]
l2#: immed[reg2, 6]
l2_mylab#:
alu[reg,reg1,+,reg2]
br=0[l2_mylab#]
```

2.8.5.1 Protect Macro Locals

Implementation of the Protect Macro Locals feature addresses a long-standing problem with the Assembler that is essentially caused by the fact that the preprocessor (like the C-preprocessor) is a text-based preprocessor and does not deal with semantically meaningful elements. The problem occurs when a register (or signal or label) is passed in to a macro as an argument, and that macro also declares a local register with the same name.

For example:

```
#macro test[arg]
.begin
.reg tmp
immed[tmp, 0x1234]
alu[arg, arg, +, tmp]
.end
#endm
.reg tmp
test[tmp]
```

This code is expanded by the preprocessor into:

```
.reg tmp
.begin
.reg tmp
immed[tmp, 0x1234]
alu[tmp, tmp, +, tmp] ; !!!! See comments below
.end
```

In the line `alu[tmp, tmp, +, tmp]` ; one of the '`tmp`' registers refers to the local register, and another '`tmp`' to the register that is passed in. However this distinction is lost during the expansion.

Previously, in order to address this issue, the Assembler generated a warning (Number 5155) when *there was a possibility that this condition would be encountered*. The Protect Macro Locals feature is intended to fix the problem by preventing it from occurring in the first place.

Note

The preprocessor is very general and can be used in a variety of ways. This feature only addresses the normal usage that accounts for the vast majority of cases.

When this feature is enabled, the preprocessor looks for register and signal declarations that are defined within a local scope (i.e. within a `.begin/.end` pair) within the macro, and any labels defined within the macro. It does this using the original/unexpanded macro body text. This means that if there are constructs such as unmatched `.begin/.end` directives due to `#if`, then the preprocessor's *behavior gets unpredictable*.

If the preprocessor encounters such tokens that do not match with the name of a macro parameter, it appends a prefix to those tokens and references to them. This means that the above example expands to:

```
.reg tmp
.begin
.reg __M00000__tmp
immed[__M00000__tmp, 0x1234]
alu[tmp, tmp, +, __M00000__tmp]
.end
```

and the local and passed-in registers are distinct. The Assembler then filters out the prefix so that it appears in the list file as:

```
.reg l0000!tmp
.begin
.reg l0001!tmp
immed[l0001!tmp, 0x1234]
alu[l0000!tmp, l0000!tmp, +, l0001!tmp]
```

2.8.5.1.1 Enabling and Disabling the Protect Macro Local Feature

The preprocessor modifies the code and it has the potential to modify it incorrectly. For this reason, the Protect Macros Local feature is implemented with the option to be enabled or disabled on a **per-macro** basis. The enabling/disabling option is controlled by two directives:

```
#protect_macro_locals on

#protect_macro_locals off
```

The two directives increment and decrement the count of how many times the options are turned on. *At the time the macro is defined if the count is greater than 0 then the Protect Macro Local feature is enabled*.

A command-line argument "`-pml`" initializes the count to 1 (indicating that the feature is enabled). If this argument is missing, the count is initialized to 0 (indicating that the feature is disabled).

The presence of the command-line argument allows for several different usage models.

- The feature can be enabled initially (either through the directive or command-line argument), and the `#protect_macro_locals off` directive can be used to disable it for a specified macro.
- Alternately, the feature can be disabled initially, and the `#protect_macro_locals on` directive can be used to enable it for a specified macro.

Note

Since the directives are maintained as a count, the `#protect_macro_locals on/off` pair can nest; that is if there are two "on" directives in a row, then one would need two "off" directives to undo their effect.

The local registers for this feature are defined as registers declared within a `.begin/.end` block (where the `.begin/.end` are in the same macro body) without a global (or similar) keyword.

Identifiers and labels in the source code should avoid substrings of the form: "__M#####__", where # represents a decimal digit. If such substrings appear and the Protect Macro Locals feature is enabled, then these identifiers and labels will not display properly in the list file.

2.8.6 Conditional Assembly (#Ifdef, #If, #else, #elif, #endif)

Conditional directives are similar to the directives used with a standard CPP. If the identifier is defined (for #ifdef), then the following text-lines are included in the output. If the identifier is not defined, the else lines are included. There may be multiple #elif clauses. If none of the #if or #elif clauses are true and there is an #else clause, those text lines will be included. Identifier replacement is done within the constant expression.

Conditional assembly constructs can nest.

Conditional Directives:

#if	#else	#elif
#ifdef	#ifndef	#endif

Format:

```
#ifdef identifier ;   ifdef can be replaced with ifndef
...text-lines...
#else
...text-lines...
#endif
```

Format:

```
#if const-expr
...text-lines...
#elif const-expr
...text-lines...
#else
...text-lines...
#endif
```

Example:

```
#define test_val 5
#if (test_val > 3)
immed[reg,1]
#elif (test_val < 3)
immed[reg,2]
#else
immed[reg,3]
#endif
```

Assembles to:

```
immed[reg,1]
Changing the test_val to 0 assembles to:
immed[reg,2]
Changing the test_val to 3 assembles to:
immed[reg,3]
```

2.8.7 Error Reporting (#error)

Print the specified string with the specified error severity level.

Format:

```
#error [severity] "message string"
```

Error Reporting Severity Levels:

Severity Level	Meaning
0	Information.
1	Warning.
2	Error.
3	Error; abort processing this file.
4	Error; abort all processing.

2.8.8 File Inclusion (#include)

The line of code with the include directive will be replaced with the contents of the named file.

Format:

```
#include “filename.ext"
```

2.8.9 Import Variable (.import_var)

Defines a set of symbolic names that will be imported by the Linker. These directives may be used wherever a numeric constant may be used. A typical example would be assigning the value of a symbol to a register via the immed opcode.

Format:

```
.import_var sym1 sym2 ...
```

The scope of imported variables is essentially global, although the symbol can only be referenced after its definition. An import variable is also considered global across all microengines.

Import variables are run-time constants (is_rt_const(var) will be true) and when used with the preprocessor function is_rt_const they do not need to be prefixed with "i$". When using the preprocessor function isimport, they need to be prefixed with "i$". Use of is_rt_const and no "i$" prefix is recommended. Note that the "i$" prefix, if present, will be removed and the symbol names used to assign values to import variables will not have such a prefix.

2.8.10 Code Block Directive (.begin, .end)

The block directives define a region of code and a new set of registers that are only visible within that region. Blocks are explicitly delimited by ".begin" and ".end" directives.

Format:

```
.begin
.reg name1 name2 ...
...
.end
```

2.8.11 Manual Register Allocation (.addr)

In rare cases a programmer may want to manually allocate registers.

Note

If any GPR is manually allocated, all GPRs must be manually allocated. The same is true for neighbor registers. However, transfer registers and signals can be partially manually allocated and the rest automatically allocated.

Manual register and signal allocation is done with the following directive:

```
.addr name address [bank]
```

- name : Name of register or signal
- address : Address of register
- bank : In the case of GPRs, bank should either be "a" or "b".

2.8.12 Memory Allocation Directives

These directives allow you to “allocate” a block of memory from a shared resource (e.g., MEM). The first, .alloc_mem, is preferred.

```
.alloc_mem name type[+offset] scope size [align] [attrib]
```

```
.local_mem name type[+offset] size [align] [attrib]
.global_mem name type[+offset] size [align] [attrib]
```

- name : Name of block being defined
- type : In which region is the block being allocated (e.g. EMEM, IMEM, CTM, CLS, LMEM). These types may be preceded by "iM.", where 'M' is an absolute island number, if the type name doesn't already imply an island. Example: 'iM.emem' identifies memory on island M, but so does 'emem0'. 'iM.ctm' identifies a specific island CTM, but 'ctm' identifies the local island CTM. The difference between "ctm" and "i0.ctm" (as well as "cls" and "i0.cls") is that the first format resolves to a 0 based address if possible (untranslated, see IMB CPP Address Translation) whereas the full format resolves to a non-zero translated address. The available types are:
  - emem : External Memory spread across islands 24, 25 and 26.
  - emem_cache_upper : Direct Access External Memory Cache, upper 1MiB, spread across islands 24, 25 and 26.
  - imem : Internal Memory spread across islands 28 and 29.
  - ctm : Cluster Target Memory on local island.
  - cls : Cluster Local Scratch on local island.
  - ememN : External Memory on island (24 + N).
  - ememN_cache_upper : Direct Access External Memory Cache, upper 1MiB, on island (24 + N).
  - imemN : Internal Memory on island (28 + N).
  - ctmN : CTM on island N.
  - lmem, lm : Microengine local memory on local microengine. Scope must be "me" or "file". See below for details on scope.
- offset : A byte offset from the start of the memory region. Default is to let the linker choose the memory offset.
- scope : The scope of the symbol ("file", "me", "island", "global"). Scope "me" is similar to ".local_mem global" and "global" is similar to .global_mem, with the exception that CLS and CTM can now have true global scope. Symbols with "file" scope are similar to ".local_mem" and is also affected by the scope of .begin/.end blocks it resides in. Symbols with "me" or wider scope are always global to the file the are declared in and are not affected by .begin/.end blocks.
- size : Size of region in bytes.
- align : Required alignment (in bytes). Default is 8 for Memory Unit memories and 4 for other memory types.

- attrib : "reserved" - The memory block is only allocated and marked as reserved. It has no data, cannot be initialized and will not be zeroed by the loader. A reserved block may, for example, refer to a memory region owned and managed by host drivers. The user's code may access such memory, but mark it as reserved to ensure that no linker allocated memory overlaps with the region.

  attrib : "addr32" and "addr40" - The address of the memory block is only allowed to be addrX bits wide. The default is addr40. These attributes allow users to be specific about requirements on symbol addresses as they are used in code. The linker will verify the allocated address against the address width specified by this attribute. This means code that uses 32-bit addressing mode for symbols that are always assumed to be allocated with bits 32 to 39 as zero can do so safely by adding addr32 to the declaration. For example, an NFP-32xx CLS symbol has always been accessible with 32-bit addressing mode and in NFP-6xxx this is still the case for a local island CLS symbol as long as the IMB CPP Address Translation Tables are configured correctly, which is true in the SDK toolchain. One can now add addr32 to local CLS symbols to ensure that any future changes to the resolved symbol address due to IMB CPP Address Translation Table changes will be reported by the linker as an error. This is more important for NFP-6xxx local island CTM symbols where the current recommended IMB CPP Address Translation Tables are set up so that all off-island Memory Units are accessed using 40-bit addresses and local island CTM can be accessed with a 32-bit address. For local CTM symbols it is important to use addr32 if the code assumes that 32-bit access is possible.

The differences between the symbol scopes are in what happens when two or more microengines declare a block with the same name. In general, any blocks which are declared with "file" or "me" scope result in an individual block for the microengine and blocks declared with "global" scope refer to the same block. Blocks declared with "island" scope refers to the same symbol for microengines in the same island. However, due to the nature of the CLS and CTM region and shared control stores (SCS), some special cases apply.

- CTM, CLS (for local island only), .global_mem: Blocks are global only to all microengines in the same island. If the same declaration is found in list files assigned to other clusters, that block is isolated and may get a different address assigned at link time. When the region is island specific, .alloc_mem should be used for less ambiguity.
- SCS, .local_mem, .alloc_mem with "me" and "file" scope: Blocks are local to the two code sharing microengines for regions other than LM and local to the specific microengine for the LM region even though the blocks will be assigned the same addresses. For example, normally a local block would reserve two blocks of memory if two list files declared the same local block, but for code sharing microengines only a single block is reserved.

The result within the Assembler is that there is a symbolic constant, defined by name, that behaves as if it were an imported symbol and whose value is the base address of the block.

The region LM corresponds to the microengine's local-memory and is only valid for local ME regions (.local_mem) (it can't be shared between microengines). For this region, the alignment must be a multiple of 4.

The region CLS corresponds to the cluster local scratch memory. There is one cluster local scratch memory with size up to 64K bytes per each micro engine cluster.

If two blocks are declared in the same scope, then the parameters must be “compatible”, that is all parameters must either be equal or zero. It is an error to have all of the parameters zero. For example, a programmer could define the same block several times with sizes of 10, 10, 0, 0, 10, and 0. But they could not define the same block with sizes of 10 and 14.

Note that imported symbols and memory block names share the same name-space as GPR register names. Imported symbols (and hence memory block names) take precedence over register names. That is, if there is (in scope) a

GPR with the name XXX and a memory block with the name XXX, then references to XXX will be taken as the memory block.

2.8.13 Memory Block and Register Initialization

There are two directives used to initialize registers or memory blocks.

The ".init" directive is used to initialize registers or blocks of memory:

```
.init name[+offset] value1 value2 …
```

- name : Name of block or register being referenced.
- offset : A byte offset from the start of the block. The offset must be a multiple of 4.
- value*n* : 32-bit value to be stored at the designated address.

The ".init_ctx" directive is used to initialize relative registers, but only for the specified context:

```
.init_ctx ctx relative_reg value
```

- ctx : Context number for the relative register.
- relative_reg : Name of context relative register being referenced.
- value : Value to be stored at the designated address.

The loader will initialize all words with the values specified by *.init*. Any remaining words that are not explicitly initialized will be initialized to zero by the loader.

2.8.14 Local Memory Mode Directives

```
.local_mem0_mode type
.local_mem1_mode type
.local_mem2_mode type
.local_mem3_mode type
```

- type : Either abs or rel.

This sets the mode of the specified local memory index register. If the mode is not specified, it defaults to "rel" (relative).

If the mode is set to "abs" (absolute), then all contexts share one physical register when accessing local memory. When set to "rel", each context has it's own register. Note that the two registers (0 and 1) may be set differently.

2.8.15 Number of Contexts Directive

```
.num_contexts n
```

- n : Number of contexts (8 or 4).

If this directive is not specified, it defaults to 8.

This directive can appear any number of times throughout the module, but if any conflict, the Assembler will generate an error.

2.8.16 Initial Next Neighbor Mode Directive

```
.init_nn_mode mode
```

- mode: either "self" or "neighbor".

This sets the initial NN_MODE in the CTX_ENABLE register. If two different values are specified, an error results. If no value is specified, the default value is “neighbor”.

2.8.17 Structured Assembly

The goal of these directives is to allow programmers to organize the control flow of their programs into structured blocks as opposed to a sea of goto statements. These directives begin with a period (.), not a pound sign (#).

2.8.17.1 Conditional (.if, .elif, .else, .endif, .if_unsigned, .elif_unsigned)

The sequence of the .if, .elif, .if_unsigned, .elif_unsigned,.else and .endif directives is the most general way of writing a multi-way decision. Conditional expressions (cond-expr) are executed in order. If any cond-expr is true, the text lines following it will be executed, and this terminates the whole chain. The last .else directive handles the "none of the above" or default case where none of the above conditional expressions is satisfied. When there is no explicit action for the default, the trailing .else directive can be omitted.

Format:

```
if-part [elif-part]* [else-part] endif-line
if-part: if-line text-lines
if-line: .if[_unsigned] cond-expr
elif-part: elif-line text-lines
elif-line: .elif[_unsigned] cond-expr
else-part: else-line text-lines
```

```
else-line: .else
endif-line: .endif
```

2.8.17.2 Repeat Loops (.repeat, .until, .until_unsigned)

The .repeat, .until, and .until_unsigned directives generate instructions specifying that the text-lines will be executed until the conditional expression is true. The conditional expression is evaluated after the loop is executed.

Format:

```
repeat-line text-lines until-line
repeat-line:.repeat
until-line:.until cond-expr

repeat-line text-lines until-unsigned_line
repeat-line:.repeat
until_unsigned-line:.until_unsigned cond-expr
```

2.8.17.3 2.While Loops (.while, .while_unsigned, .endw)

The .while, .while_unsigned and .endw directives generate instructions specifying that the text-lines will be executed as long as the conditional expression is true. The conditional expression is evaluated before the loop is executed.

Format:

```
while-line text-lines endw-line
while-line:.while cond-expr
endw-line:.endw

while_unsigned_line text-lines endw-line
while_unsigned_line:.while_unsigned cond-expr
endw-line:.endw
```

2.8.17.4 Break and Continue

The break and continue directives generate instructions that skip the remaining portions of .while and .repeat loops. The .break directive causes the loop to terminate, and execution continues at the code following the loop. The .continue directive will cause the next iteration of the loop to occur, provided that the loop condition allows it.

Format:

```
.break
.continue
```

2.8.17.5 Conditional Expressions

A condition-expression is used to select the path for control flow in the previous constructs. The expression has four forms:

- A constant.
- A comparison of a register with either another register or a constant.
- A comparison of a function against a constant.
- A testing of the condition codes.

In the function form, the function can either access one bit of a register, one byte of a register, the carryout, the context, an input state, or a signal. Conditional expressions have the following form:

```
Syntax:
  cond-expr: cc-expr
    non-cc-expr
  non-cc-expr:const
    eq-expr
    gt-expr
    log-expr
  gt-expr: (reg-expr reg-op const)
    (const reg-op reg-expr)
    (reg-expr reg-op reg)
    (reg reg-op reg-expr)
  reg-expr: reg
    reg shift-op const
    reg shift-op reg
  shift-op:  >>
    <<
  reg-op:    ==
    !=
    >
    >=
    <
    <=
    &
  eq-expr:   (func func-op const)
    (const func-op func)
    func
  func:      BIT(reg,pos)
    !BIT(reg,pos)
    BYTE(reg,pos)
    CLS_STATE(state)
    !CLS_STATE(state)
    COUT()
    !COUT()
    CTX()
    INP_STATE(state)
    !INP_STATE(state)
    SIGNAL(signal)
    !SIGNAL(signal)
  func-op:   ==
    !=
  cc-expr:   =0
    !=0
```

```
   0
   =0
   <0
   <=0
 log-expr: (non-cc-expr log-op non-cc-expr)
 log-op:   ||
     &&
```

This defines a condition-expression, used to select the path for control flow in the previous constructs. The expression has four forms: a constant, a comparison of a register with either another register or a constant, a comparison of a function against a constant, or a testing of the condition codes. In the function form, the function can either access one bit of a register, one byte of a register, the carryout, the context, an input state, or a signal.

When a function appears by itself, it is the same as function != 0. The not operator (!) can be applied to functions that return a boolean result. Note that the gt-expr forms result in an ALU opcode followed by a branch opcode. Note also that all numeric constants can be replaced by constant expressions.

The pos and const values must fall into the ranges shown in Table 2.12:

Table 2.12. pos and const Values

Function	Pos	Const
bit	0–31	0–1
byte	0–4	0–255
count	N/A	0–1
ctx	N/A	0–7
inp_state	N/A	0–1
cls_state	N/A	0–1
signal	N/A	0–1
n/a = not applicable		

2.8.18 Warning Directives

Each warning will be associated with a numeric ID and a *warning-level*. The level indicates the degree of seriousness of the warning. Level-1 is the most serious. Level-4 is the least serious. The default is Level-2. Additionally, there is a warning level parameter for the program run, which determines which warnings are generated and which are suppressed. If the warning level is set to *n*, then all warnings whose level is ≤ *n* will be presented. Those >*n* will be suppressed.

Warnings that may be of interest to you under some circumstances, but which would just be “noise” most of the time, would be Level-4 warnings. So by default, these would not appear, but you could cause these to be visible if desired. The warning level is set with the following command line options:

```
-w Same as –W0
-Wn Set the warning level to n with 0 <= n <= 4
-WX Make all warnings (at or below the level given by –Wn) errors
```

Note

Setting the warning level to zero, effectively disables all warnings. Note also that -Wn and –WX can both appear together.

Warnings can be modified using the following directives:

```
#pragma warning(spec:list,…)
#pragma warning(push)
#pragma warning(push,n)
#pragma warning(pop)

spec: Warning specifier from following table
list: One or more space separated warning identifier numbers
```

Spec	Meaning
once	Only display the listed warnings once.
default	Reset the warning attributes to their default value.
1,2,3,4	Apply the indicated warning level to the listed warnings.
disable	Disable the listed warnings. Example: `#pragma warning(disable: 5145).`
error	Treat the listed warnings as errors.

The push and pop pragmas save and restore the warning attributes and global warning level on a stack. The (push,*n*) pragma also sets the global warning level to *n*.

In the case that a warning applies to multiple lines, and these lines have different attributes, the most conservative attributes will be used. That is, if either line treats it as an error, the warning will be assigned to that line, otherwise the line which applies the lowest level to that warning will be used. In short, in such cases, the only way to suppress such a warning is to suppress it at both lines.

The approximate meanings of the warning levels are:

Level	Meaning
1	Severe Warning: The code is probably incorrect. This is not an error because assembly can continue and it is slightly possible that the code may not be incorrect.
2	Medium Warning: This is the default level for warnings.
3	Minor Warning: The code can probably assemble correctly, but it might contain bugs. The programmer should probably take a look at these to be safe, but these can probably be safely ignored.
4	Informative Warning: This is a low-level warning that may warn about possible performance issues or other minor things that do not directly affect the correctness of the program.

2.8.19 User Data Directive

Format:

```
.app_metadata token1 [token2 … tokenN]
```

- tokenN : Text to be inserted into the `.list` file.

Example:

```
.app_metadata “List file was assembled on” __DATE__ “at”  __TIME__
```

2.8.20 Byte Ordering Directive

This directive sets the byte ordering for all command instruction read or write actions. Note that the default byte ordering for command instructions is big endian.

Format:

```
.endian byte_order
```

- byte_order : Either big or little.

Example:

```
.endian little
mem[write,$1,b,0,1]
; The above DRAM write command is equivalent to
mem[write_le,$1,b,0,1]
```

2.8.21 Init-CSR

These directives specify CSR initializations that will be performed, in random order, by the loader before starting microengines. An initialization setting can set values, check for specific values on the target or simply mark the CSR as owned by the NFFW.

The primary difference between .init_csr and #pragma init_csr is that the #pragma version allows the use of the is_csr_set preprocessor function to perform conditional code compilation. This means that code can be compiled based on CSR settings previously set by #pragma init_csr if needed. See the is_csr_set preprocessor function for how to do so.

Also see the Linker section in the Development Tools User's Guide for detailed information on the init-csr modes and conflict checking.

Format:

```
.init_csr tgt:map.map.reg.field [value] [mode]
.init_csr tgt:map.map.reg.field [(value_expr)] [mode]
#pragma init_csr tgt:map.map.reg.field [value] [mode]
```

The #pragma version only accepts constants in decimal or hexadecimal format and cannot accept an expression or symbol for "value". Note that parenthesis are required when specifying an expression or symbolic constant value. See the examples section below for usage examples.

The register initialization mode specified the intention of the NFFW and is accepted or rejected by the loader. The loader may also have a preset register database that states which registers it set, required or reserves and will ensure that NFFW may not override certain CSR. When any single NFFW CSR initialization is rejected, the NFFW load stops.

Table 2.13. Init-CSR Modes

mode	Description
const (default)	Set the CSR to "value". If mode is omitted, "const" is used.
required	Verify that the CSR is set to "value". If not, the NFFW is not loaded.
volatile	Mark the CSR as volatile. This means the NFFW intends to change the CSR at any time to any value. If "value" is specified, the initial value is written on NFFW load, but the NFFW may still change it.
invalid	Verify that the CSR is NOT set to "value". If it is, the NFFW is not loaded. Multipe 'invalid' entries can be used to reject a series of values.

The register or register field name (tgt:map.map.reg.field) is case insensitive and the map names will match those in the NFP-6xxx Databook. The "tgt" part of the name is a top level map and is a shorter, more convenient, name. The names for "tgt" are as follows:

Table 2.14. Init-CSR Lookup Target Map Names

tgt	Map Name	Virtual Maps
arm	ARMGasketMemoryMap	None
cls	ClusterScratchCppMap	iX
ila	IlaCppAddressMap	iX
nbi	NbiAddressMap	iX
pcie	PcieInternalTargetsCpp (same as ClusterScratchSSB, but with CPP offsets)	iX
xpb	ChipXPB	iX shorthand for real map
xpbm	ChipXPBM	iX shorthand for real map
mecsr	MeCsrCPP	iX.meY[.ctxN]

The names for maps, registers and fields are then those that follow in the documentation, but in order to identify target islands and microengines, some virtual maps are introduced. If the virtual maps are omitted, the local island or microengine is referenced and the final absolute register name is resolved in the linker based on where the list file is located. The XPB lookup targets are exceptions since the top level maps already contain the fully resolved CSR offset (island ID included), but to achieve similar operation, the second level maps are aliased to the relevant iX maps and omitting the second level also results in local island register names. For example, "ArmIsldXpbmMap" can be replaced with "i1" and "Me3IsldXpbmMap" with "i35".

For "mecsr" names, all context indirect CSRs should be prefixed with "ctxN" for each context N.

It is recommended to use names of the the specific register fields. Specifying the field allows for better error checking and readability. Specifying a register as a whole does no checking on reserved or read-only bit fields and the user must be aware of how the bit fields for the register will behave when written.

Also see the Linker section in the Development Tools User Guide.

2.8.21.1 Examples

```
// Set local microengine's Mailbox0 on load and state the intention to modify it.
.init_csr mecsr:Mailbox0 0x11223344 volatile
```

```
// Set IndLMAddr0 on load to the value of the symbolic constant
.local_mem lm0 lm+0x200 16
.init_csr mecsr:ctx0.IndLMAddr0 (lm0) volatile
```

```
// Set IndLMAddr0 on load and state the intention to modify it.
#for ctxn [0,1,2,3,4,5,6,7]
  .init_csr mecsr:ctx/**/ctxn/**/.IndLMAddr0 (0x100 * ctxn) volatile
#endloop
```

```
// Verify emem0 ConfigCPP.IgnBulkAlign.
.init_csr xpbm:i24.Island.ExtMuXpbMap.MuConfigReg.ConfigCPP.IgnBulkAlign 1 required
```

```
// Set local island CLS Ring Base
.init_csr cls:Rings.RingBase0.Base 0x100
```

2.8.21.2 Recommended Settings for NFP-32xx code ported to NFP-6xxx

2.8.21.2.1 Default mem[] destination

The following settings will ensure that the island on which the list file is loaded will send all mem[] commands to island 24 (emem0), except for Direct Access locality. This is **not** recommended for new code and is different from the NFP-6xxx recommended settings.

```
.init_csr xpb:IMBXpbMap.Target7AddressModeCfg.Island0 24
.init_csr xpb:IMBXpbMap.Target7AddressModeCfg.Mode 3
.init_csr xpb:IMBXpbMap.Target7AddressModeCfg.AddrMode 1
```

2.8.21.2.2 Default sram[] destination

The following settings will ensure that the island on which the list file is loaded will send all sram[] commands to island 24 (emem0). All SRAM channels will be VQDR and the VQDR base addresses must be set to user-determined locations. The regions in the MU used for VQDR should then also be marked as reserved to prevent normal MU memory symbols to overlap the VQDR regions.

```
// Set a base address for each channel where convenient
.init_csr xpbm:i24.Island.ExtMuXpbMap.MuConfigReg.ConfigVQDR0.MemWinBase 0x10
.init_csr xpbm:i24.Island.ExtMuXpbMap.MuConfigReg.ConfigVQDR1.MemWinBase 0x20
.init_csr xpbm:i24.Island.ExtMuXpbMap.MuConfigReg.ConfigVQDR2.MemWinBase 0x30
.init_csr xpbm:i24.Island.ExtMuXpbMap.MuConfigReg.ConfigVQDR3.MemWinBase 0x40

// The settings below are Netronome recommended and will be the defaults.
// The VQDR target settings must be the same on all islands.
// A list filename can contain different settings, but the toolchain views
// the legacy "sram" resource as a global and shared memory resource and if
// islands use different destination islands, they are no longer sharing
// the same memory.
// Also important to remember is the nfp_sram_read and nfp_sram_write
// functions in the NFP API. These are subject to IMB CPP Address Translation
// CSRs of the island the function is called from (ARM, PCIeN, etc.) and
// if the list file used settings different from these islands, the loader
// will not initialise the correct VQDR memory.
// Send all VQDR commands to island 24 (emem0)
//.init_csr xpb:IMBXpbMap.Target2AddressModeCfg.Island0 24
//.init_csr xpb:IMBXpbMap.Target2AddressModeCfg.Island1 24
// Use 32-bit addressing
//.init_csr xpb:IMBXpbMap.Target2AddressModeCfg.AddrMode 0
// and mode 1.
//.init_csr xpb:IMBXpbMap.Target2AddressModeCfg.Mode 1
```

```
// Reserve 32 MB for each channel at the emem0 VQDR base addresses.
// Smaller or larger reservations can be used for smaller or larger VQDR channels.
.alloc_mem sram_reserved0 i24.emem+0x10000000 global (0x100000 * 32) reserved
.alloc_mem sram_reserved1 i24.emem+0x20000000 global (0x100000 * 32) reserved
.alloc_mem sram_reserved2 i24.emem+0x30000000 global (0x100000 * 32) reserved
.alloc_mem sram_reserved3 i24.emem+0x40000000 global (0x100000 * 32) reserved
```

2.8.22 Hierarchical Resource Allocation

With the directives ".declare_resource" and ".alloc_resource" one can declare generic resource pools and allocate resource symbols within those pools. A resource pool can be allocated within another generic resource pool or within a normal memory symbol. This allows for hierarchical resource pools which are either generic or backed by a memory resource. The resulting resource symbols are used in the same way as memory symbols, resolved at link time and become symbols in the NFFW file symbol table.

Many of the resources used by microcode applications are not directly related to regular memory regions, for example allocating ring numbers, event filters or channel identifiers. Typically such identifiers would be manually assigned and defined using predefined macros. Generic resource pools can aid in managing such resources, not only simplifying it but also provide pool and resource scoping.

Format of .declare_resource:

```
.declare_resource name scope size [base[+offset] | +offset]
```

- name : The name of this resource pool.
- scope : The scope of this resource pool. It works similar to the scope argument to .alloc_mem, except "file" scope is not supported. This means that all resources are global to the file they are declared in, after which they have the scope specified here.
- size : The byte size of this resource pool.
- base : Optional. Specifies that this resource pool is nested inside the resource "base", which can be either another resource pool or a memory symbol.
- offset : Optional. If specified on "base", this specifies the offset of this resouce pool within the base resource. The offset will always be relative to the base resource's own offset. If specified without a "base" resource, this resource pool is a generic resource with the first value being "offset", in other words a range of numbers from "offset" to "offset + size - 1" inclusive. If the offset is not specified, it will be allocated by the linker if the resource has a base resource, otherwise it will be 0.

Format of .alloc_resource:

```
.alloc_resource name resource[+offset] scope size [align]
```

- name : The name of this resource symbol.
- resource : The resource this symbol will be allocated from.

- offset : Optional. Specifies the fixed byte offset within "resource" at which the symbol must be allocated.
- scope : The scope of the resource symbol ("file", "me", "island", "global"). The scope of the symbol cannot be wider than the resource it is allocated from. See the description of scope for .alloc_mem for more information.
- size : Size of resource symbol in bytes.
- align : Required alignment (in bytes). Default is 1 for generic resources, for memory backed resources it is 8 for Memory Unit resources and 4 for other memory types.

Any memory backed resource symbol can be initialized using .init similar to how memory symbols are initialized.

2.8.22.1 Examples

2.8.22.1.1 Basic Usage

The following example shows a basic generic resource used to allocate a series of numbers scoped to the microengine. Note that the example uses the phrase "Most likely allocated". This is what will happen in a small example, but a user should not assume compact or sequential allocation even in cases like this where it seems clear.

```
.reg vals[4]

.declare_resource numbers me 16
.alloc_resource numA numbers me 1
.alloc_resource numB numbers me 1
.alloc_resource numC numbers me 1
.alloc_resource numD numbers me 1

immed[vals[0], numA] // Most likely allocated as 0
immed[vals[0], numB] // Most likely allocated as 1
immed[vals[0], numC] // Most likely allocated as 2
immed[vals[0], numD] // Most likely allocated as 3
```

The following example adds an offset to the resource pool so that the first allocation start at 8.

```
.reg vals[4]

.declare_resource numbers me 16 +8
.alloc_resource numA numbers me 1
.alloc_resource numB numbers me 1
.alloc_resource numC numbers me 1
.alloc_resource numD numbers me 1

immed[vals[0], numA] // Most likely allocated as 8
immed[vals[0], numB] // Most likely allocated as 9
immed[vals[0], numC] // Most likely allocated as 10
immed[vals[0], numD] // Most likely allocated as 11
```

The following example adds a manually allocated symbol in the middle of the likely allocation range to illustrate how to "reserve" a range of numbers in this pool.

```
.reg vals[4]

.declare_resource numbers me 16
.alloc_resource rsvd numbers+2 me 4

.alloc_resource numA numbers me 1
.alloc_resource numB numbers me 1
.alloc_resource numC numbers me 1
.alloc_resource numD numbers me 1

immed[vals[0], numA] // Most likely allocated as 0
immed[vals[0], numB] // Most likely allocated as 1
immed[vals[0], numC] // Most likely allocated as 6
immed[vals[0], numD] // Most likely allocated as 7
```

2.8.22.1.2 Basic Scoping

The following example shows how scoping can be used on a generic resource. Assume that the example is assigned to ME 0 and ME 1 in island A and ME 0 and ME 1 in island B.

```
.reg g, i, m

// Declare a global resource, all islands have access to it.
.declare_resource numbers global 16

// Allocate a global number. All MEs will use the same numA value.
.alloc_resource numA numbers global 1

// Allocate an island scoped number. All MEs in the same island
// will have the same numB value
.alloc_resource numB numbers island 1

// Allocate an ME scoped number. Each ME will have it's own
// numC value.
.alloc_resource numC numbers me 1

immed[g, numA]
immed[i, numB]
immed[m, numC]
```

In the above example, a likely allocation for the code assigned to the states microengines is shown below.

From ME	numA	numB	numC
iA.me0	0	1	2
iA.me1	0	1	3
iB.me0	0	4	5
iB.me1	0	4	6

2.8.22.1.3 Memory Backed Resource

This example shows two simple memory backed resource pools residing in the same memory symbol. It is very much similar to a the previous examples, but the allocated resource symbols effectively become memory symbols.

```
.reg vals[4]

.alloc_mem memsym cls me 64
.declare_resource memresA me 32 memsym
.declare_resource memresB me 32 memsym

.alloc_resource res0 memresA me 8
.alloc_resource res1 memresA me 16
.alloc_resource res2 memresB me 24
.alloc_resource res3 memresB me 8

immed[vals[0], res0] // Most likely allocated as 0x00
immed[vals[1], res1] // Most likely allocated as 0x08
immed[vals[2], res2] // Most likely allocated as 0x20
immed[vals[3], res3] // Most likely allocated as 0x38
```

Extending the above example, one can initialise the allocated resource symbols or the base memory symbol using the .init directive just as one would for memory symbols. Note that once any memory backed resource symbol in any part of a hierarchical resource tree is initialized, all resource symbols and the base memory symbol itself are all handled as initialized symbols. This is because the base memory symbol should always fully contain all symbols allocated from it. Any symbols without initialization directives will still be zero initialized, but as .data regions and not as .bss regions.

```
.reg vals[4]

.alloc_mem memsym cls me 64
.declare_resource memresA me 32 memsym
.declare_resource memresB me 32 memsym

.alloc_resource res0 memresA me 8
.alloc_resource res1 memresA me 16
.alloc_resource res2 memresB me 24
.alloc_resource res3 memresB me 8

.init memsym+8 0xaabbccdd
.init res3+0 0x11223344

// For the likely allocation below, the following
// .init will conflict with the memsym+8 .init
// and an error will be reported since different
// values are used to initialize the same physical
// memory.
//.init res1+0 0x00000000

immed[vals[0], res0] // Most likely allocated as 0x00
immed[vals[1], res1] // Most likely allocated as 0x08
immed[vals[2], res2] // Most likely allocated as 0x20
immed[vals[3], res3] // Most likely allocated as 0x38
```

The memory dump of the memsym symbol is shown below.

```
0x00000000: 00000000 00000000 aabbccdd 00000000
0x00000010: 00000000 00000000 00000000 00000000
0x00000020: 00000000 00000000 00000000 00000000
0x00000030: 00000000 00000000 11223344 00000000
```

2.8.22.1.4 Hierarchical Resources

This example shows how new resources can be allocated from other resources and symbols resources then allocated from various depths of the resource tree. In this example all resource pools are of the same scope in order to only illustrate the hierarchical nature. The next example introduces scoping, which may seem more intuitive when nesting resources.

```
.reg vals[10]

.declare_resource pool_top me 32
.declare_resource pool_midA me 14 pool_top
.declare_resource pool_midB me 14 pool_top

.declare_resource pool_botA0 me 6 pool_midA
.declare_resource pool_botA1 me 6 pool_midA
.declare_resource pool_botB0 me 6 pool_midB
.declare_resource pool_botB1 me 6 pool_midB

.alloc_resource res0 pool_botA0 me 2
.alloc_resource res1 pool_botA0 me 2
.alloc_resource res2 pool_botA1 me 2
.alloc_resource res3 pool_botA1 me 2
.alloc_resource res4 pool_botB0 me 2
.alloc_resource res5 pool_botB0 me 2
.alloc_resource res6 pool_botB1 me 2
.alloc_resource res7 pool_botB1 me 2

.alloc_resource res8 pool_midB me 2
.alloc_resource res9 pool_top me 2

immed[vals[0], res0] // Most likely allocated as 0x00
immed[vals[1], res1] // Most likely allocated as 0x02
immed[vals[2], res2] // Most likely allocated as 0x06
immed[vals[3], res3] // Most likely allocated as 0x08
immed[vals[4], res4] // Most likely allocated as 0x0e
immed[vals[5], res5] // Most likely allocated as 0x10
immed[vals[6], res6] // Most likely allocated as 0x14
immed[vals[7], res7] // Most likely allocated as 0x16
immed[vals[8], res8] // Most likely allocated as 0x1a
immed[vals[9], res9] // Most likely allocated as 0x1c
```

Taking a smaller version of the previous example and introducing scoping, the example below shows how each microengine can allocate a resource local to itself, share a resource with other microengines in the local island and also use globally shared resources.

```
.reg vals[9]

.declare_resource pool_top global 256
.declare_resource pool_midA island 56 pool_top

.declare_resource pool_botA0 me 6 pool_midA
.declare_resource pool_botA1 me 6 pool_midA

.alloc_resource res0 pool_botA0 me 2
.alloc_resource res1 pool_botA1 me 2

.alloc_resource res2 pool_midA island 1
.alloc_resource res3 pool_midA me 1

.alloc_resource res4 pool_top global 1
.alloc_resource res5 pool_top island 1
.alloc_resource res6 pool_top me 1

immed[vals[0], res0]
immed[vals[1], res1]
immed[vals[2], res2]
immed[vals[3], res3]
immed[vals[4], res4]
immed[vals[5], res5]
immed[vals[6], res6]
```

The symbol table of the above example is shown below, assuming the same code was assigned to the following microengines: i35.me0, i35.me7, i38.me2, i38.me3. Note that the linker created new symbols for resource pools that are nested inside other resource pools.

```
.Name                          Value      Size
------------------------------ ---------- ------
i35._R$pool_midA               0x00000000 0x0038
i35.me0._R$pool_botA0          0x00000000 0x0006
i35.me0.res0                   0x00000000 0x0002
i35.me0._R$pool_botA1          0x00000006 0x0006
i35.me0.res1                   0x00000006 0x0002
i35.res2                       0x0000000c 0x0001
i35.me0.res3                   0x0000000d 0x0001
i35.me7._R$pool_botA0          0x0000000e 0x0006
i35.me7.res0                   0x0000000e 0x0002
i35.me7._R$pool_botA1          0x00000014 0x0006
i35.me7.res1                   0x00000014 0x0002
i35.me7.res3                   0x0000001a 0x0001
res4                           0x00000038 0x0001
i35.res5                       0x00000039 0x0001
i35.me0.res6                   0x0000003a 0x0001
i35.me7.res6                   0x0000003b 0x0001
i38._R$pool_midA               0x0000003c 0x0038
i38.me2._R$pool_botA0          0x0000003c 0x0006
i38.me2.res0                   0x0000003c 0x0002
i38.me2._R$pool_botA1          0x00000042 0x0006
i38.me2.res1                   0x00000042 0x0002
i38.res2                       0x00000048 0x0001
i38.me2.res3                   0x00000049 0x0001
i38.me3._R$pool_botA0          0x0000004a 0x0006
i38.me3.res0                   0x0000004a 0x0002
```

```
i38.me3._R$pool_botA1         0x00000050 0x0006
i38.me3.res1                  0x00000050 0x0002
i38.me3.res3                  0x00000056 0x0001
i38.res5                      0x00000074 0x0001
i38.me2.res6                  0x00000075 0x0001
i38.me3.res6                  0x00000076 0x0001
```

2.8.22.1.5 Scoped counters

This is a small example that shows how a global resource is used to allocate global, island and microengine scoped resource symbols for counters. The example simply increments all three counters and can be triggered with a mailbox message. For this example, the code is assigned to iA.me0, iB.me0 and iB.me1.

```
.alloc_mem mem_counters emem global 64
.declare_resource counters_pool global 64 mem_counters

.alloc_resource cntr_glb  counters_pool global 8
.alloc_resource cntr_isld counters_pool island 8
.alloc_resource cntr_me   counters_pool me 8

.reg v
.reg addr_glb_H addr_glb_L
.reg addr_isld_H addr_isld_L
.reg addr_me_H addr_me_L

.init addr_glb_H (cntr_glb >> 8)
.init addr_glb_L (cntr_glb & 0xFF)
.init addr_isld_H (cntr_isld >> 8)
.init addr_isld_L (cntr_isld & 0xFF)
.init addr_me_H (cntr_me >> 8)
.init addr_me_L (cntr_me & 0xFF)

.init v 1

.if (ctx() != 0)
    ctx_arb[kill]
.endif

local_csr_wr[mailbox0, 1]
nop
nop
nop

.while (v)
    local_csr_rd[mailbox0]
    immed[v, 0]
    .if (v == 2)
        mem[incr64, --, addr_glb_H, <<8, addr_glb_L]
        mem[incr64, --, addr_isld_H, <<8, addr_isld_L]
        mem[incr64, --, addr_me_H, <<8, addr_me_L]
        local_csr_wr[mailbox0, 1]
    .endif
    nop
    nop
    ctx_arb[voluntary]
.endw
```

```
ctx_arb[kill]
```

If the above code is executed and each ME gets two mailbox messages (mailbox0 = 2), the result is that the global counter has the value 6, the island counter in island A has value 2 and in island B has value 4, each microengine counter has the value 2.

2.8.23 Assert Directive

The .assert directive is a link time assertion function that will fail and abort linking if the given expression does not evaluate to true (nonzero)

Format:

```
.assert (expression) ["message"]
```

The expression can contain constants, memory symbols, signal and register addresses or predefined import variables. The optional message will be displayed if the assertion expression evaluates to zero. If no message is given, the expression will be displayed.

Example:

```
// make sure the list file is assigned to the first ME in an island
.assert(__MENUM == 0)
```

Example:

```
.alloc_mem emem_r1 emem0+0x40 global 64
.assert ( (relbase(emem_r1) % 0x40) == 0)
```

Use .assert with relbase on the symbol offset to verify alignment or allocation within a range. In the above example the assert will make sure that the memory region we are working with is aligned to 0x40. If this is not the case the assert will cause the linking phase to error and bail out.

2.8.24 Memory Unit Queue and Ring directives

The memory unit queue initialization is done with microcode generated by the linker, executed by the loader before finally loading user code. The same code is used in simulation and hardware loaders, which means these directives result in the simulator being run for a few hundred clock cycles to finish executing the init-code. In order to reduce this simulation run time, all queue descriptor loading routines are distributed across all microengines used by the firmware to run in parallel. There is no relationship between the location of the directive and the ME that handles it.

The linker will merge directives that have the same format and the same island ID and ring number. If there are any conflicts, an error will be reported. There are still invalid cases that the linker may not detect, such as when the same memory region is used for two different rings (which would lead to corrupt data) by using the same symbol to hold the queue descriptor or different symbols defining the same descriptor.

2.8.24.1 Load MU Queue Desciptor (.load_mu_qdesc)

This directive has two formats. One uses a symbol which is pre-initialized with a queue descriptor (referred to as type A here) and the other takes four 32-bit words (expressions) directly (type B).

2.8.24.1.1 Type A

Format:

```
.load_mu_qdesc <iid> <ring_num> <user_symbol>
```

`iid` must be a string of the form "iX" or a valid island alias (e.g. i24 or emem0). This is required even though the island might be deduced from the symbol location and the user has to ensure that the island ID is the same as the island the user symbol is located in.

`ring_num` can be a simple constant or any expression (placed in round brackets) that resolves to a constant in the range 0 to 1023 at link time. It is therefore possible to allocate ring numbers from a generic resource and use them symbol name here.

`user_symbol` can be any symbol or bracketed expression, there is no validation of the resolved address and the `iid` argument. This means you can define the Queue Descriptor to be located at any 16-byte aligned offset within a memory symbol. This argument defines the location of the queue descriptor which will be loaded with `mem[rd_qdesc` and waited on with `mem[push_qdesc`. It can be the same as the actual ring symbol of course, the only requirement is that there are four 32-bit words at the location which are the queue descriptor. These words need to be initialised with `.init` to ensure that the memory is ready before init-code is run.

2.8.24.1.2 Type B

Format:

```
.load_mu_qdesc <iid> <ring_num> <(word_0)> <(word_1)> <(word_2)> <(word_3)>
```

`iid` and `ring_num` are the same as for Type A above.

Each `word_N` can be any expression that resolves at link time. Each word needs to be in round brackets to make it clear to the assembler preprocessor which expression is which word_N. These four words are directly used as the queue descriptor and the user must take care when writing these expressions (similar to when doing this in microcode at run time) since invalid fields may result in runtime bugs that are hard to detect.

This format will use 16 bytes in a linker created symbol (usually called `.ld_mem`) where the four words are loaded by init-code, used to load the queue descriptor. After all ring init-code blocks are distributed between microengines, each microengine gets its own 16 bytes in the `.ld_mem` symbol, which means the size of this symbol in each memory

unit island depends on the number of microengines used to initialise queues for that island. The init-code for this directive is summarized as:

- `mem[write to ld_mem`
- `mem[read to separate MU engine access`
- `mem[rd_qdesc`
- `mem[push_qdesc to ensure MU engine separation`

The linker will use a load time import variable called `__iX_emem_cache_only` which is resolved by the loader for island X. It will be 1 if ConfigCPP.DirAccWays=0xff and 0 otherwise. This lets the loader override the q_loc field in the descriptor if the target is using emem cache only. Type A does not use this.

2.8.24.2 Initialize MU Ring (.init_mu_ring)

This directive is a more convenient way to initialize the more popular memory unit type-2 queue used for rings. This directive can be replaced by a more verbose `.load_mu_qdesc` that does the same, so everything explained above applies here as well.

Format:

```
.init_mu_ring <ring_num> <ring_sym> [count [q_loc]]
```

`ring_num` is the same as for `.load_mu_qdesc`, but there is no `iid` parameter for this directive. The linker is in control of the content of the queue descriptor and therefore needs less information before allocating memory symbols than for `.load_mu_qdesc`.

`ring_sym` is a memory symbol that is the entire ring itself. The symbol's size and alignment must therefore meet the ring type requirements.

`count` is the number of **bytes** the ring is already filled with on initialization (q_count << 2). A memory symbol is filled with zero at load time by default, but `.init` directives can be used to set the value of the ring content that is deemed pre-loaded in the first `count` bytes.

`q_loc` allows control of the ring's queue locality. It can have a value of `high`, `low`, `direct` or `discard` (0, 1, 2 or 3) and defaults to `high` or `direct` depending on the memory type of the ring symbol. This value can still be overridden by the loader depending on `__iX_emem_cache_only`.

2.8.24.3 Example

```
#define HARD_CODED_RING_0   100
#define HARD_CODED_RING_1   122
#define HARD_CODED_RING_2   123

.declare_resource AllRingNumbers global 64
.alloc_resource RingNumA AllRingNumbers global 1
.alloc_resource RingNumB AllRingNumbers global 1

.alloc_mem RingA emem1 global 2048 2048
.alloc_mem RingB emem1 global 2048 2048
```

```
.alloc_mem RingC emem2 global 2048 2048
.alloc_mem RingD emem2 global 4096 4096
.alloc_mem RingE emem2 global 4096 4096

.alloc_mem QDescA emem1 global 16
.init QDescA \
  (0x0 | (RingA & 0x03ffFFFC)) \
  (2 | (RingA & 0xffFFfffc)) \
  (0x0 | ((RingA >> 8) & 0x03000000)) \
  0

// RingA initialized from QDescA
.load_mu_qdesc emem1 RingNumA QDescA
.load_mu_qdesc emem1 RingNumB \
  (0x0 | (RingB & 0x03ffFFFC)) \
  (2 | (RingB & 0xffFFfffc)) \
  (0x0 | ((RingB >> 8) & 0x03000000)) \
  0

.load_mu_qdesc emem2 HARD_CODED_RING_0 \
  (0x0 | (RingC & 0x03ffFFFC)) \
  (2 | (RingC & 0xffFFfffc)) \
  (0x0 | ((RingC >> 8) & 0x03000000)) \
  0

// RingD, empty
.init_mu_ring HARD_CODED_RING_1 RingD

// RingE prepopulated with 4 words.
.init RingE+0  0x11111111
.init RingE+4  0x22222222
.init RingE+8  0x33333333
.init RingE+12 0x44444444
.init_mu_ring HARD_CODED_RING_2 RingE 16
```

2.9 Processor Type and Revision

Over time, network processors will be released in different types and revisions with different features. Microcode written to take advantage of a particular processor type or revision will fail if it is run on the wrong processor.

To deal with this issue, you can specify a type and range of revisions for which you want your microcode assembled. This is done using the following command line options: **-chip <SKU>** (where SKU is a valid SKU show by the **-chip_list** option) and **–REVISION_MIN** and **–REVISION_MAX**. The legacy options **-nfp-6xxxc**, **-nfp3216**, **-nfp3216c**, **-nfp3240** and **nfp3240c** can also be used but the **-chip** is preferred. The **–REVISION_MIN** and **–REVISION_MAX** options are only needed when the SKU has multiple chip revisions.

For simplicity, the `-REVISION` option can be used to set the minimum and maximum to the same value. These options will target the assembly to a particular type and revision of processor. Several predefined preprocessor macros will be defined according to type and revision.

For more information on the predefined macros and on writing version specific microcode, please see the *Netronome NFP-6XXX Databook* for the network processor.

Table 2.15. File Extensions

Extension	File Details/usage	How it is created
.uc	Microcode source file. This is an input file to the Assembler.	Created by the programmer.
.uci	Contains details of register allocation and assembly time errors and warnings.	Created by the Assembler.
.list	Output of the Assembler. It contains microcode, Linker directives and source code comments. This file is used as the input file to the nfld Linker.	Created by the Assembler.
.ucp	Output of the Assembler preprocessor phase which expands macros, includes files, conditionally compiles code, does structured assembly and token replacements. It can be used for debugging.	Created by the Assembler.
.nffw	Netronome Flow Firmware file. This is a collection of data from several `.list` files. It contains microcode, memory and register initialization information as well as debugging data. It is used as an input file to the loader and is an ELF file.	Created by the Linker.

3. Technical Support

To obtain additional information, or to provide feedback, please email <support@netronome.com> or contact the nearest **Netronome** technical support representative.

Appendix A. NFAS Warnings

A.1 Introduction

This section contains the descriptions of the NFAS Assembler Warnings.

Table A.1. NFAS Warnings

Warning Number	Message	Section
4101	Register was defined but never used.	Section A.2
4700	TYPE NAME is used before being set.	Section A.3
4701	TYPE NAME may be used before being set.	Section A.4
4702	Unreachable Code.	Section A.5
5000	Command line option overrides previous option.	Section A.6
5002	The qualifier “any” is being ignored due to “--”. Make sure Bit-16 is set in the WAKEUP_EVENTS register.	Section A.7
5003	Signal is set while already set.	Section A.8
5004	Signal may be set while already set.	Section A.9
5008	TYPE NAME is set but not used.	Section A.10
5009	Use of TYPE transfer register before I/O operation is complete.	Section A.11
5011	Terminating I/O operation at end of block because of NAME.	Section A.12
5101	Option -REVISION=REVISION overrides previous Minimum Revision of REVISION.	Section A.13
5102	Option -REVISION=REVISION overrides previous Maximum Revision of REVISION.	Section A.14
5103	Option -REVISION_MIN=REVISION overrides previous Minimum of REVISION.	Section A.15
5104	Option -REVISION_MAX=REVISION overrides previous Maximum of REVISION.	Section A.16
5113	Old register allocator will try for VALUE more seconds before giving up.	Section A.17
5114	WARNING: “CONSTANT_EXPRESSION”.	Section A.18
5115	Manually allocated address for NAME conflicts with NAME at FILENAME:LINE.	Section A.19
5116	Return register may not contain a valid address.	Section A.20
5117	Unable to determine end of operation: Queue is unknown and no signal is being generated.	Section A.21

Warning Number	Message	Section
5118	The use of numbered signals is obsolete and will be removed in future versions. Please use named signals: NUMBER.	Section A.22
5121	Operand synonym hides previous ".import_var" definition.	Section A.23
5122	Operand synonym translated into itself.	Section A.24
5124	Local register hides previous ".operand_synonym" definition.	Section A.25
5125	Local register hides previous local definition.	Section A.26
5126	Local register hides previous global definition.	Section A.27
5127	Global TYPE NAME is hidden by NAME declared at FILE LINE.	Section A.28
5128	Declaration for NAME hides global/module NAME declared at FILE LINE.	Section A.29
5129	Changing “all” to “any” for ctx_arb[kill].	Section A.30
5130	NAME "NAME" has been renamed “NAME”. Future Assembler versions may not support the old name.	Section A.31
5131	The reflector is sending a signal to both, but the signal was not manually specified with .addr for OPCODE opcode.	Section A.32
5133	Label is not followed by a valid uword.	Section A.34
5134	The directives .xfer_order_rd and .xfer_order_wr are obsolete. For sanity checking, please use “.reg read” or “.reg write”.	Section A.35
5135	CRC type “crc_16” is not supported, defaulting to “crc_ccitt”.	Section A.36
5136	Option -CPU=n will be phased out. Please use OPTION.	Section A.37
5137	Use of old-style Reflector Tokens will be removed in the next release. Please update your code.	Section A.38
5138	The “ffs” operator will be phased out for instruction “alu”. Please use the “ffs” instruction instead.	Section A.39
5139	The "vnop" instruction will be phased out. Please use "nop".	Section A.40
5140	Reference to unreachable label was modified.	Section A.41
5141	Use of operand_synonym is obsolete and will be removed in future versions. Please use #define instead.	Section A.42
5143	A PROCESSOR_REVISION processor revision was specified when targeting multiple processor types.	Section A.43
5144	In the .init directive, the first data item starts with a '+'. Did you mean this to be an offset? If so, there should be no spaces before the offset.	Section A.44
5145	Register should be declared as an aggregate because it is referenced as such at FILENAME(NUMBER).	Section A.45
5146	Register should be referenced as an aggregate because it is declared as such at FILENAME(NUMBER).	Section A.46
5147	Ignoring repeated instance of specifier for directive.	Section A.47

Warning Number	Message	Section
5148	If the context for “INDIRECT_REGISTER_NAME” is swapped in, the write will not take effect because it has been placed in the defer slot of a context swapping instruction.	Section A.48
5149	Ignoring invalid specifier for directive.	Section A.49
5150	Import variable does not begin with “i$”, so “isimport()” incorrectly returned false.	Section A.50
5151	Declaration for “REGISTER” hides previous declaration at FILENAME(NUMBER).	Section A.51
5153	Command line parameter is obsolete and has been ignored.	Section A.52
5154	This read of neighbor register will not reflect the value written.	Section A.53
5155	Declaration of “REGISTER_OR_SIGNAL” matches a macro argument. References to the macro argument will be hidden.	Section A.54
5156	Mismatching endianness for "BYTE_ALIGN" and "BYTE_ALIGN" at line FILENAME(NUMBER).	Section A.55
5157	Target for "LABEL" resides in the same page. A "br" will provide better performance.	Section A.56
5158	Page was marked "default" and is zero-sized.	Section A.57
5159	INSTRUCTION is used across the context-swapping instruction at FILENAME(NUMBER).	Section A.58
5161	Immediate operand, NAME == 0xVALUE, cannot fit in N-bit field in opcode.	Section A.59
5162	Missing qualifier on doubled signal.	Section A.60
5163	Declaration for "NAME" hides register declared at FILENAME(NUMBER).	Section A.61
5164	No page was declared for this uword -- using default page in default region.	Section A.62
5165	This arithmetic expression has generated two instructions, is this OK?	Section A.63
5166	There are more than one DIRECTIVE_NAME directives in the program!	Section A.64
5167	A local_csr_wr to an index register is in a defer shadow and consequent use will access old value.	Section A.65
5168	Length operand is not used by this instruction. Please replace it with "--”.	Section A.66
5169	Transfer register is not used by this instruction. Please replace it with "--".	Section A.67
5170	There are no DRAM registers in extended mode. All transfer registers should be specified as $n or $.	Section A.68
5171	"num_sigs" token will have no effect since no "sig_done" or "ctx_swap" were specified.	Section A.69

Warning Number	Message	Section
5172	"uwr" and "urd" instructions will not be supported in the next chip release.	Section A.70
5173	Addressing mode token is deprecated. Please use a new 40-bit addressing mode syntax.	Section A.71
5174	pragma addressing has been deprecated. Please use a new 40-bit addressing mode syntax.	Section A.72
5175	Specified memory block alignment is ignored when a specific memory offset is requested.	Section A.73
5176	Constant exceeds 32-bit range.	Section A.74
5177	Immediate value could cause an increment of NN_GET.	Section A.75
5178	Only signal[0] may be used for an instruction expecting data_error signals.	Section A.76
5179	Local symbol shadows wider scope symbol with the same name.	Section A.77

A.2 NFAS Warning (level 4) 4101

Register was defined but never used.

Description: The specified register was defined but was not referenced.

Example:

```
.reg x
; no other reference to x
```

How to fix: No fix required. To get rid of the warning, delete the specified register declaration.

A.3 NFAS Warning (level 1) 4700

TYPE NAME is used before being set.

Description: The specified register or signal is used before being set. The TYPE field identifies the type of NAME. The TYPE can be “Signal”, “Read Transfer Register”, “Write Transfer Register”, or “Register”.

This warning indicates that the specified register or signal is definitely being used before it is set.

Example:

```
.reg x
alu[--,--,b,x]
```

How to fix: This warning indicates a programming error. The source code should be rewritten to set the register to an appropriate value before it is used.

A.4 NFAS Warning (level 3) 4701

TYPE NAME may be used before being set.

Description: The specified register or signal may be used before being set. The TYPE field identifies the type of NAME. The TYPE can be “Signal”, “Read Transfer Register”, “Write Transfer Register”, or “Register”.

This warning indicates that the specified register or signal is set before being used on some possible paths of execution, but is used before being set on others. This may be caused by a programming error, or it may be due to the Assembler assuming that all conditionals are independent.

This warning does not indicate that there is definitely a programming error, but rather that there is a possibility of a programming error.

For more details, see Section 2.5.10.

Example:

```
Example
.reg r1 r2
… ; set r1
.if (r1 == 1)
immed[r2, 0]
.endif
… ; does not change r1
.if (r1 == 1)
alu[--,--,b,r2]
.endif
```

How to fix: The programmer should check whether there are any valid paths where the specified register is used before being set. For example, in the above example, if the second conditional were “.if (r1 == 2)” then it would be possible for the ALU instruction to be executed without the IMMED. This indicates a programming problem, and the code should be rewritten to avoid this situation.

If, however, all valid paths do set the register before using it (in the above example, this is true because either both IFs are taken or both are skipped), then one can put a “.set NAME” immediately before the first branch, so that all paths back from the use see the set. Note that it is incorrect to put the .set between an actual assignment and the use. In this case, the Assembler would assume that the actual assignment was irrelevant.

Alternately, the warning-level mechanism (e.g., a command-line argument of –W2) can be used to suppress this warning.

A.5 NFAS Warning (level 2) 4702

Unreachable Code.

Description: This is the first line of a block of code that cannot be reached during normal microcode execution. The Assembler will comment out the unreachable code. This may be due to a gross programming bug, the omission of the TARGETS parameter on JUMP instructions, or by having invalid values stored in the register used in RTN.

Example:

```
#macro incr(arg, count)
.if (arg == 0)
alu[count, count, +, 1]
.else
alu[count, count, +, 2] ; unreachable
.endif
#endm
…
incr[0, count]
```

How to fix: If the unreachable line is due to a programming error, the error needs to be corrected. Alternatively, a “#pragma warning” can be used to disable this particular warning, but there is no good way to prevent spurious warnings generated by this other than using a “#pragma warning” to disable each one as it occurs.

A.6 NFAS Warning (level 1) 5000

Command line option overrides previous option.

Description: The command line option overrides a previous option. This warning is generated when a command line option was repeated or two mutually exclusive command line options were provided.

Example:

```
nfas -nfp3216 -nfp3240 test.uc
```

How to fix: Remove the unwanted option.

A.7 NFAS Warning (level 3) 5002

The qualifier “any” is being ignored due to “--”. Make sure Bit-16 is set in the WAKEUP_EVENTS register.

Description: The code contains “ctx_arb[--], any”. Because of the “--”, the “any” qualifier is being ignored. The programmer needs to make sure that in computing the value that is written into the WAKEUP_EVENTS register, that bit-16 (which indicates “any”) is being set.

In other words, when used with “ctx_arb[--]”, the “any” token has no functional significance and is merely a hint to the reader of the code. If this hint is to be accurate, then the programmer must make sure that bit-16 is set in the WAKEUP_EVENTS register.

This is not an indication of a programming error, it is a reminder to the programmer.

Example:

```
ctx_arb[--], any
```

How to fix: Either the “any” token can be removed, or a “#pragma warning” can be used to disable this warning.

A.8 NFAS Warning (level 1) 5003

Signal is set while already set.

Description: The indicated signal is definitely set while it is already set; i.e., there is a missing CTX_ARB or “BR_!SIGNAL” between the two sets. The problem here is that one does not know when the second signal arrives because it will be masked by the first signal.

This warning indicates a programming error.

Example:

```
sram[read, $x, addr,0, 1], sig_done[sig] ; This sets sig
sram[read, $y, addr,1, 1], sig_done[sig] ; This sets sig again
```

How to fix: A CTX_ARB, BR_SIGNAL, or BR_!SIGNAL needs to be inserted as appropriate to clear the signal between the two settings of it.

A.9 NFAS Warning (level 3) 5004

Signal may be set while already set.

Description: The indicated signal is possibly set while it is already set; i.e., there is a missing CTX_ARB or “BR_!SIGNAL” between the two sets. This may be caused by a programming error, or it may be due to the Assembler assuming that all conditionals are independent.

This warning does not indicate that there is definitely a programming error, but rather that there is a possibility of a programming error.

Example:

```
.set_sig s
.if (a == 0)
sram[read, $x, a,0, 1], sig_done[s]
.endif
nop
.if (a == 0)
ctx_arb[s]
.endif
;;; .io_completed s
sram[read, $x, a,0, 1], ctx_swap[s]
```

The problem occurs above because the Assembler assumes that the first IF can be taken and the second one skipped, which would result in the last SRAM using the signal before it is cleared.

How to fix: If the cause is a programming error, then that error needs to be corrected. If it is due to a situation similar to the above example, then the indicated “.io_completed” can be used to indicate that the I/O operation is completed and that the signal can be safely re-used. Alternately, after inspection, the programmer may decide to use the Warning Level or #pragma warning mechanism to mask this warning.

A.10 NFAS Warning (level 4) 5008

TYPE NAME is set but not used.

Description: The specified register or signal is set but not used. The TYPE field identifies the type of NAME. The TYPE can be “Signal”, “Read Transfer Register”, “Write Transfer Register”, or “Register”. A common cause would be setting the register again before it is being used or reading a transfer register and not using the results.

This warning does not necessarily indicate a programming error; it is of a more informational nature. However, it is possible that this warning is caused by a programming error.

Example:

```
immed[x, 0] ; Warning 5008
...
immed[x, 1]
...
alu[--,--,b,x]
```

The problem is that the value used in the ALU is determined by the second IMMED, and the first IMMED is therefore immaterial.

```
alu[dummy,reg,-,5]
bne[label#]
```

The problem is that dummy is not referenced and the ALU is needed to generate a condition code.

```
.xfer_order $a $b $c
sram[read, $a, addr,0, 3], ctx_swap[sig]
alu[--,--,b,$a]
alu[--,--,b,$c]
```

The problem here is that $b is not being used.

How to fix: If the assignment is not necessary but the ALU is needed for condition codes, then the destination should be "--". Otherwise, if the assignment is truly not needed, it can be removed from the code. In the case of the third example, the ".use" directive can be used to generate a "use" of the register without generating actual microcode.

A.11 NFAS Warning (level 1) 5009

Use of TYPE transfer register before I/O operation is complete.

Description: A transfer register of the given type (READ/WRITE) is being used in an I/O operation, but it is being referenced before the I/O operation is known to be complete. This may work some of the time, but with different loading, this may fail.

Example:

```
immed[$x, 0]
sram[write, $x, addr,0, 1], sig_done[s]
...
immed[$x, 1] ; Warning 5009
ctx_arb[s]
```

The problem here is that if the SRAM write is suitably delayed, then the $X can get over-written with 1 before the SRAM unit fetches the value 0.

```
sram[read, $x, addr,0, 1], sig_done[s]
ctx_arb[s], defer[1]
alu[--,--,b, $x] ; Warning 5009
```

Instructions in the defer shadow of a CTX_ARB are executed before the CTX_ARB returns, so references to the transfer registers in the defer shadow are considered to be executing before the I/ O operation completes. In this case, the ALU will execute before the SRAM/read completes, and the wrong value of $x will be used.

How to fix: Wait until the I/O operation completes.

A.12 NFAS Warning (level 1) 5011

Terminating I/O operation at end of block because of NAME.

Description: An I/O operation was not completed when the end of a .begin/.end type of block was reached, and one of the transfer registers or signals being used in the I/O operation was scoped locally to the block.

The problem is that the registers/signals only have existence within their defining block. After the block is finished, the associated physical registers/signals are available for reuse. But if the I/O operation is not really completed, then the associated registers and signals are not really available.

Example:

```
.sig s
.begin
.reg $x
sram[read, $x, addr,0, 1], sig_done[s]
nop
nop ; Warning 5011
.end
...
ctx_arb[s]
```

The problem is that after the ".END", $x is still going to have a value written into it, but the programmer has indicated that use of $x should cease outside of that block.

How to fix: Make sure that all I/O operations are completed before the block is terminated, or declare the appropriate registers/signals within a higher block/globally.

A.13 NFAS Warning (level 2) 5101

Option -REVISION=REVISION overrides previous Minimum Revision of REVISION.

Description: The minimum revision number was already set by a previous –REVISION or –REVISION_MIN command line option. Note, the –REVISION option sets both the minimum and maximum target revisions to the specified value.

Example:

```
nfas –REVISION_MIN=0 –REVISION=1 file.uc
```

How to fix: Remove the unwanted revision command line option.

A.14 NFAS Warning (level 2) 5102

Option -REVISION=REVISION overrides previous Maximum Revision of REVISION.

Description: The maximum revision number was already set by a previous –REVISION or –REVISION_MAX command line option. Note, the –REVISION option sets both the minimum and maximum target revisions to the specified value.

Example:
```
nfas –REVISION=0 –REVISION=1 file.uc
```

How to fix: Remove the unwanted revision command line option.

A.15 NFAS Warning (level 2) 5103

Option -REVISION_MIN=REVISION overrides previous Minimum of REVISION.

Description: The minimum revision number was already set by a previous –REVISION_MIN or –REVISION command line option.

Example:
```
nfas –REVISION=0 –REVISION_MIN=1 file.uc
```

How to fix: Remove the unwanted revision command line option.

A.16 NFAS Warning (level 2) 5104

Option -REVISION_MAX=REVISION overrides previous Maximum of REVISION.

Description: The minimum revision number was already set by a previous –REVISION_MAX or –REVISION command line option.

Example:
```
nfas –REVISION_MAX=0 –REVISION_MAX=1 file.uc
```

How to fix: Remove the unwanted revision command line option.

A.17 NFAS Warning (level 2) 5113

Old register allocator will try for VALUE more seconds before giving up.

A.18 NFAS Warning (level 1) 5114

WARNING: “CONSTANT_EXPRESSION”.

Description: This warning indicates that the specified constant expression generated a warning. The only warning at present is: `Future assembler versions will not allow “=” for equality expressions. Please use “==”`

Example:

```
immed[reg, (0=0)]
```

How to fix: Replace the “=” with “==”.

A.19 NFAS Warning (level 1) 5115

Manually allocated address for NAME conflicts with NAME at FILENAME:LINE.

Description: The two registers or signals named had their addresses manually allocated, they are both in use at the same time, and their addresses conflict.

The warning is reported for one of the address directives and points to the other address directive.

This warning is indicative of a programming error.

Example:

```
.reg $x $y
.addr $x 0
.addr $y 0
immed[$x,0]
immed[$y,0]
alu[--,--,b,$x]
alu[--,--,b,$y
```

How to fix: Assign one of the registers or signals to a different address or make sure that both registers/signals are not being used at the same time.

A.20 NFAS Warning (level 1) 5116

Return register may not contain a valid address.

Description: For the Assembler to correctly allocate registers and signals, it needs to know where RTN statements go. To do this, there is a requirement that the register used in the RTN instruction needs to have it's value loaded with LOAD_ADDR or be a copy of such a value, and it cannot be used in a previous RTN instruction since being loaded.

This warning indicates that this condition was not met. The result is that the flow-graph computed by the Assembler will be incomplete, and there is a strong potential that register/signal allocation may be faulty.

This warning is indicative of a programming error.

Example:

```
load_addr[reg, lab#]
alu[reg,reg,+,1] ; this makes the register invalid
...
rtn[reg]
```

How to fix: Make sure that the register used in the RTN instruction always has a valid return value in it.

A.21 NFAS Warning (level 2) 5117

Unable to determine end of operation: Queue is unknown and no signal is being generated.

Description: An I/O operation was issued with no signal being generated. In limited cases, this is allowed, but in general the ordering between separate I/O operations is not guaranteed, so a signal is required. Otherwise, there is no way to determine when the I/O operation completes.

Example:

```
sram[read, $x, addr,0, 1]
```

How to fix: Supply a signal, either through CTX_SWAP or SIG_DONE.

A.22 NFAS Warning (level 2) 5118

The use of numbered signals is obsolete and will be removed in future versions. Please use named signals: NUMBER.

Description: In an earlier version of the Assembler, signals could be specified as numeric values. This usage has been replaced with named signals. There is no longer any reason to use numeric signals, and use of these may cause strange effects (for example, if a numeric signal is used in the context of a doubled signal).

Example:
```
sram[read, $x, addr,0, 1], sig_done[3]
```

How to fix: Replace the signal with a named signal. If necessary, use the .ADDR directive to assign it to desired numerical value. For example:

```
.sig SIG3
.addr SIG3 3
sram[read, $x, addr,0, 1], sig_done[SIG3]
```

A.23 NFAS Warning (level 3) 5121

Operand synonym hides previous ".import_var" definition.

A.24 NFAS Warning (level 4) 5122

Operand synonym translated into itself.

A.25 NFAS Warning (level 4) 5124

Local register hides previous ".operand_synonym" definition.

A.26 NFAS Warning (level 4) 5125

Local register hides previous local definition.

A.27 NFAS Warning (level 4) 5126

Local register hides previous global definition.

A.28 NFAS Warning (level 4) 5127

Global TYPE NAME is hidden by NAME declared at FILE LINE.

Description: A global register is declared within the scope of a register with local scope. References to this name will refer to the local one until the scope of that local register is exited.

Example:

```
.begin
.reg x
.reg global x
immed[x,0] ; This refers to the local x, NOT the global one
.end
```

How to fix: If the programmer wants to reference the global variable, then one of the names needs to be changed.

A.29 NFAS Warning (level 4) 5128

Declaration for NAME hides global/module NAME declared at FILE LINE.

Description: A local register is declared with the same name as a global register. References to this name will refer to the local one until the scope of that local register is exited.

This is an informational warning and does not indicate a programming problem.

Example:

```
.begin
.reg global x
.reg x
immed[x,0] ; This refers to the local x, NOT the global one
.end
```

How to fix: If the programmer wants to reference the global variable, then one of the names needs to be changed.

A.30 NFAS Warning (level 2) 5129

Changing “all” to “any” for ctx_arb[kill].

Description: The “ctx_arb[kill]” must have no optional tokens or the “any” token. If the code uses the “all” token, then the Assembler changes it to “any” and generates this warning.

Example:
```
ctx_arb[kill], all
```

How to fix: Remove the “all” token or change it to “any”.

A.31 NFAS Warning (level 2) 5130

NAME has been renamed “NAME”. Future Assembler versions may not support the old name.

Description: During development of the Assembler, some names have changed. For compatibility with earlier releases, this version still accepts the old name, but such support will be removed in future versions.

Example:
```
l#: br_inp_state[fcififo_full, l#]
```

How to fix: Change the name as indicated.

A.32 NFAS Warning (level 1) 5131

The reflector is sending a signal to both, but the signal was not manually specified with .addr for opcode.

Description: When SIG_BOTH is specified for a reflector operation, the signal needs to be manually allocated (to the same address) in both Microengines (MEs). If it is not manually allocated in the Microengine containing the reflect command, then this warning is generated.

This warning indicates a programming error. It is unlikely that the code will function correctly.

Example:
```
reg $x
.reg remote $r
.sig s
reflect[read, $x, 0, $r, 0, 1], sig_both, ctx_swap[s]
```

How to fix: The signal needs to be manually allocated (using the .addr directive) to the same address in both this and the remote Microengine.

A.33 NFAS Warning (level 2) 5132

Value of 2 specified in “TOKEN” qualifier for “cam_lookup” opcode may result in an address which exceeds the amount of Local Memory.

Description: The optional lm_addr0 or lm_addr1 token indicates that the lookup result should also be loaded into the given local memory address register. The immediate value provided by the token is written to bits[11:10] of the register. A lm_addr token value of two results in a base DWORD address of 512. Since the size of local memory is only 640 DWORDs, the result of the lookup could result in an address that exceeds the valid local memory range.

Example:

```
cam_lookup [dest, src], lm_addr0[2]
```

How to fix: No fix is required if it is known that the lookup result will not exceed the valid memory range.

A.34 NFAS Warning (level 3) 5133

Label is not followed by a valid uword.

Description: A label is applied to a directive that is not followed an actual uword/instruction. Thus, the label really applies to an undefined instruction following the assembled code.

Example:

```
nop
l#: .import_var y
; end of file
```

How to fix: Either remove the label or add more uwords at the end of the input.

A.35 NFAS Warning (level 4) 5134

The directives .xfer_order_rd and .xfer_order_wr are obsolete. For sanity checking, please use “.reg read” or “.reg write”.

Description: In previous versions of the Assembler, a transfer register could be declared as “read-only” or “write-only” by using the .xfer_order_rd or .xfer_order_wr directives. This was supported for

backwards compatibility and to support implicit register declarations. The preferred mechanism is to declare them with either ".reg read" or ".reg write".

In the current version of the Assembler, the lifetimes of transfer registers that are declared (implicitly or explicitly) as BOTH (i.e., both READ and WRITE) are tracked separately, so the only benefit of using .xfer_order_rd or .xfer_order_wr was checking that a read-transfer register was not written-to, and vice versa.

This feature is no longer supported. In particular, code with uses .xfer_order_rd or .xfer_order_wr to make transfer registers read or write only should still assemble in the same manner as before. The only feature lacking is that if source code changes use one of these registers in the "wrong" manner, this change would not generate an error. If this checking is desired, the code must be changed to use ".reg read" or ".reg write".

Example:

```
.xfer_order_rd $x
```

How to fix: Use ".reg read" or ".reg write".

A.36 NFAS Warning (level 1) 5135

CRC type "crc_16" is not supported, defaulting to "crc_ccitt".

Description: In previous versions of the Assembler, the CRC-CCITT polynomial was misnamed CRC-16. Specifying CRC type "crc_16" actually results in a CRC-CCITT calculation. Future versions of the Assembler will generate an error when "crc_16" is specified as a CRC type.

Example:

```
crc_be [crc_16, dest, src ]
```

How to fix: Replace the "crc_16" operand with "crc_ccitt", for example:

```
crc_be [crc_ ccitt, dest, src ].
```

A.37 NFAS Warning (level 1) 5136

Option -CPU=n will be phased out. Please use OPTION.

Description: When using the Assembler from the command line, the processor type is no longer specified using the "-CPU" option. The command line options "-nfp3216" and "-nfp3240" should be used instead.

Example:
```
nfas –CPU=2 filename.uc
```

How to fix: Use the appropriate “-nfpDDDD” option, for example:
```
nfas -nfp3216 filename.uc
```

A.38 NFAS Warning (level 1) 5137

Use of old-style Reflector Tokens will be removed in the next release. Please update your code.

Description: The syntax for the sig_initiator, sig_both, and sig_remote tokens for the reflector mode of the cap instruction has changed. The old syntax will not be supported in future releases.

Example:
```
reflect[write, $xfer0, 0, $reflect_in0, 0, 1], ctx_swap [reflect_done], \
   sig_initiator

reflect[write, $xfer0, 0, $reflect_in0, 0, 1], ctx_swap [reflect_done], \
   sig_both
reflect[write, $xfer0, 0, $reflect_in0, 0, 1], sig_done [reflect_done], \
   sig_remote
```

How to fix: Update the syntax for the sig_initiator, sig_both, and sig_remote tokens.

A.39 NFAS Warning (level 1) 5138

The “ffs” operator will be phased out for instruction “alu”. Please use the “ffs” instruction instead.

Description: Previous versions of the Assembler supported the ffs operator for the alu instruction in addition to the ffs instruction. Future versions of the Assembler will only support the ffs instruction.

Example:
```
alu [dest, --, ffs, src]
```

How to fix: Use the ffs instruction, for example:
```
ffs [dest, src]
```

A.40 NFAS Warning (level 1) 5139

The “vnop” instruction will be phased out. Please use “nop”.

Description: In previous versions of the Assembler, the optimizer would remove vnop instructions but not nop instructions. Since the optimizer now removes nop instructions, there is no difference between the two, making the vnop instruction obsolete.

Example:

```
crc_be [crc_16, dest, src ]
vnop
crc_be [crc_16, dest, src ]
```

How to fix: Replace vnop instructions with nop instructions.

A.41 NFAS Warning (level 1) 5140

Reference to unreachable label was modified.

Description: This warning is generated when a load_addr instruction references an unreachable label. This indicates that the register loaded by the load_addr instruction was never used by a rtn instruction.

Example:

```
load_addr [rtn_addr, rtn_label#]
br [subroutine#]

rtn_label#:
       alu [--, --, b, 0]
       ...
       ctx_arb [kill]
subroutine#:
       alu [--, --, b, 1]
       ; missing rtn [rtn_addr]
```

How to fix: Add a rtn instruction (if missing) or remove the unnecessary load_addr instruction and the unreachable code.

A.42 NFAS Warning (level 2) 5141

Use of operand_synonym is obsolete and will be removed in future versions. Please use #define instead.

A.43 NFAS Warning (level 1) 5143

A PROCESSOR_REVISION processor revision was specified when targeting multiple processor types.

A.44 NFAS Warning (level 3) 5144

In the .init directive, the first data item starts with a '+'. Did you mean this to be an offset? If so, there should be no spaces before the offset.

Description: A '+' on a data item is interpreted as being the sign for the data item. When the Assembler finds a “+' on the first data item, it is likely that a space has been inadvertently inserted between the region name and the optional offset, so the Assembler reports a warning.

Example:

```
.local_mem samp_block DRAM 100
.init samp_block +1 -1
.init samp_block +16 0x1234
```

How to fix: Either remove the space or the '+':

```
.local_mem samp_block DRAM 100
.init samp_block 1 -1
.init samp_block+16 0x1234
```

A.45 NFAS Warning (level 1) 5145

Register should be declared as an aggregate because it is referenced as such at “FILENAME(NUMBER)”.

Description: Previous versions of the Assembler somewhat supported aggregate notation by removing brackets from register names that were in aggregate notation, e.g., $name[0] would become $name0. This meant that register references could use aggregate notation, but declarations

could not. Now that the Assembler has true support for aggregates, the Assembler will first try to match an aggregate reference to an aggregate declaration. If the register is not found, the Assembler will default to the old behavior and remove the brackets. Microcode should be updated to take advantage of the aggregate support.

Example:

```
; "aggregate" declaration prior to full aggregate support
.reg $name0 $name1 $name2 $name3

immed[$name[0],0]
```

How to fix: Replace declarations with aggregates:

```
; aggregate declaration now supported
.reg $name[4]

immed [$name[0],0]
```

A.46 NFAS Warning (level 1) 5146

Register should be referenced as an aggregate because it is declared as such at “FILENAME(NUMBER)”.

Description: Previous versions of the Assembler somewhat supported aggregate notation by removing brackets from register names that were in aggregate notation, e.g., $name[0] would become $name0. This meant that aggregate and non-aggregate names (e.g., $name[0] and $name0) could be used interchangeably. If a declaration is updated to use aggregate notation (perhaps in response to warning #5145), any non-aggregate references which do not match another register will match the aggregate declaration, but generate this warning. Microcode should be updated to take advantage of the full support for aggregates.

Example:

```
; declaration updated to use aggregate notation
.reg $name[4]

immed[$name[0], 0]
immed[$name1, 0] ; non-aggregate reference of register will match $name[1],
but trigger warning
```

How to fix: Update all non-aggregate references to use aggregate notation. The example above becomes:

```
.reg $name[4]
immed[$name[0], 0]
immed[$name[1], 0]
```

A.47 NFAS Warning (level 1) 5147

Ignoring repeated instance of specifier for directive.

Description: A specifier for the given directive was provided more than once.

Example:

```
#pragma optimize ( "dd", off)
```

How to fix: Remove extra instance of specifier:

```
#pragma optimize ( "d", off)
```

A.48 NFAS Warning (level 2) 5148

If the context for “INDIRECT_REGISTER_NAME” is swapped in, the write will not take effect because it has been placed in the defer slot of a context swapping instruction.

Description: Due to the latency in writing local CSRs, the active CSR for the swapped in context will be loaded with the indirect CSR before the write to the indirect CSR has completed. If the swapped in context is the same as the indirect write context (CSR_CTX_POINTER), then the indirect write is effectively dropped.

Example:

```
ctx_arb[signal], defer[1]
local_csr_wr [indirect_lm_addr_0, value]
```

How to fix: If by design, the swapped in context can never match the context selected by CSR_CTX_POINTER, then the warning can be disabled by:

```
#pragma warning (disable: 5148)
```

Otherwise, the local_csr_wr should be moved out of the defer shadow.

A.49 NFAS Warning (level 2) 5149

Ignoring invalid specifier for directive.

Description: The given specifier is not one recognized by the given directive, and it is therefore being ignored.

Example:

```
#pragma optimize ("g", off)
```

How to fix: Provide the correct specifier. If the directive is in an include file processed by another tool (e.g., a compiler) the warning can be ignored or disabled.

A.50 NFAS Warning (level 1) 5150

Import variable does not begin with “i$”, so “isimport()” incorrectly returned false.

Description: The preprocessor function isimport() looks for an “i$” prefix to determine whether a token is an import variable or not. The preprocessor must rely on this naming convention because it has no knowledge of microcode syntax and therefore does not understand .import_var directives. The function and naming convention were introduced with version 3.5 of the Assembler.

To assist the programmer, the preprocessor maintains a list of tokens for which isimport() returned false. If an import variable is later declared with one of those token names, the Assembler reports this warning. This warning may occur when using standard macros, as they make use of the isimport() function.

For more information on the new naming convention, please see the description for the .import_var directive.

Example:

```
#macro mymacro(a)
    #if (isimport(a))
      ; do something
    #else
      ; do something else
    #endif
#endm

.import_var noprefix

mymacro(noprefix)
```

How to fix: Add the “i$” prefix to the import variable:

```
.import_var i$withprefix
mymacro (i$withprefix)
```

A.51 NFAS Warning (level 4) 5151

Declaration for “REGISTER” hides previous declaration at FILENAME(NUMBER).

Description: A local register is declared with the same name as another local register in an enclosing begin/end block. References to this name will refer to the one in the innermost block until the scope of that local register is exited.

This is an informational warning and does not necessarily indicate a programming problem. However, this warning may be useful in detecting the situation where a macro declares a register with the same name as a macro parameter.

Example:

```
#macro mymacro(result)
.begin
   .reg tmp
    immed[tmp, 0x1234]
    alu[result,--,b,tmp]
.end
#endm

.begin
   .reg tmp
   mymacro(tmp)
.end
```

How to fix: If the programmer wants to reference the outermost variable, then one of the names needs to be changed. Otherwise, this warning can be ignored or disabled.

A.52 NFAS Warning (level 1) 5153

Command line parameter is obsolete and has been ignored.

A.53 NFAS Warning (level 1) 5154

This read of neighbor register will not reflect the value written.

A.54 NFAS Warning (level 1) 5155

Declaration of “REGISTER_OR_SIGNAL” matches a macro argument. References to the macro argument will be hidden.

Description: The specified register or signal was declared locally (i.e., within a .begin – .end block) within the body of a macro, but there was a register/signal with the same name at a higher scope, which was passed in as a macro argument. This means that references to the external register/signal

within the scope of the locally-declared register/signal will refer to the local version rather than the global one. This is almost certainly not what the programmer intended. This is a serious warning because, while it is possible that the code is written correctly and will behave correctly, in most cases this warning means that the code will not function correctly.

Example:

```
#macro swap[reg1,reg2]
.begin
    .reg tmp
    alu[tmp,--,b,reg1]
    alu[reg1,--,b,reg2]
    alu[reg2,--,b,tmp]
.end
#endm
.begin
.reg tmp tmp2
swap[tmp2,tmp]
```

This will expand into:

```
.begin
.reg tmp
alu[tmp,--,b,tmp2]
alu[tmp2,--,b,tmp]
alu[tmp,--,b,tmp]
.end
```

which does not match the programmer's intent.

How to fix: Change the name of either the local or the outer version of the indicated register to something different. In the above example, one could fix the problem by changing the macro as shown below (changing tmp to _tmp):

```
#macro swap[reg1,reg2]
.begin
    .reg _tmp
    alu[_tmp,--,b,reg1]
    alu[reg1,--,b,reg2]
    alu[reg2,--,b,_tmp]
.end
#endm
```

A.55 NFAS Warning (level 1) 5156

Mismatching endianness for "BYTE_ALIGN" and "BYTE_ALIGN" at line FILENAME(NUMBER).

Description: There are big endian and little endian versions of the byte_align instruction. A given byte_align sequence may not be interrupted by a byte_align instruction of a different type. The Assembler follows all paths and issues this warning if it finds any mismatches.

Example:

```
.if (big_endian)
byte_align_be[dest,src]
.else
byte_align_le[dest,src]
.endif
; more code here
.if (big_endian)
byte_align_be[dest,src]
.else
byte_align_le[dest,src]
.endif
```

How to fix: The example above demonstrates the case of a correlated conditional, which the Assembler does not understand. In this case, the warning can be disabled using the "#pragma warning" directive. In other cases, the warning may indicate a valid mismatch and the instruction should be changed to appropriate big or little endian version.

A.56 NFAS Warning (level 2) 5157

Target for "LABEL" resides in the same page. A "br" will provide better performance.

Description: Executing a br_page pseudo-instruction requires a considerable amount of overhead that can be avoided if the label resides in the same page as the br_page.

Example:

```
loop#:
; code
br_page[loop#]
```

How to fix: Replace the br_page instruction with a "br" instruction.

A.57 NFAS Warning (level 1) 5158

Page was marked "default" and is zero-sized.

Description: The default page for a region cannot be empty. If it is, the Assembler will ignore the default keyword and select the first non-zero page in the region as the default page.

Example:

```
.region one
.page one default
; no code (or unreachable code)
.page one
```

How to fix: Ensure that there are microwords following the page directive and that code in the page is reachable. Warning 5140 will also be issued for unreachable code.

A.58 NFAS Warning (level 2) 5159

INSTRUCTION is used across the context-swapping instruction at FILENAME(NUMBER).

Description: Certain multi-instruction sequences rely on microengine state that is not stored during a context swap. If another context executes the same instruction, the internal state will be modified and the instruction sequence will have an unpredictable result.

Example:

```
byte_align_be[dest,src]
sram[read, $x, address, 0, 1], ctx_swap[signal]
byte_align_be[dest,src]
```

How to fix: The code should be reordered so that context-swapping instruction does not fall in the middle of the sequence that generated the warning. If no other contexts are using the instruction in question, then the warning may be disabled via the "#pragma warning" directive.

A.59 NFAS Warning (level 1) 5161

Immediate operand, NAME == 0xVALUE, cannot fit in N-bit field in opcode.

Description: Immediate operands are constrained to varying bit widths, depending on the particular instruction. If a symbol whose value is known at assembly-time exceeds the instruction's limit, the Assembler issues this warning.

Example:

```
.local_mem lm_block LM 32
local_csr_wr[active_lm_addr_0, lm_block]
```

How to fix: The immed and immed_w1 instructions can be used to move large values into a temporary register. For example:

```
immed[temp,lm_block]
local_csr_wr[active_lm_addr_0, temp]
```

If the value is expected to exceed the bit-width of the operand, then the symbol should be used in an expression, with the high-ordered bits masked off, for example:

```
immed[temp,(lm_block&0xffff)]
```

A.60 NFAS Warning (level 3) 5162

Missing qualifier on doubled signal.

Description: When a doubled signal is used in a br_signal or br_!signal instruction, a qualifier should be provided to indicate whether the lower or upper half of the signal is being consumed. Lower half qualifiers are "[0]" and "[write]". Upper half qualifiers are "[1]" and "[read]". If no qualifier is provided, the Assembler will default to the lower half of the signal.

Example:

```
mem[lookup, $x, addr, <<8, 0, 1], sig_done[mem_sig]
wait#: br_!signal[mem_sig,wait#]
```

How to fix: In this example, the signal generated by the lookup command is a doubled signal, so the signal name must be qualified, for example:

```
wait#: br_!signal[mem_sig[write],wait#]
```

In addition, the upper half of the signal should also be consumed, for example:

```
wait2#: br_!signal[mem_sig[read],wait2#]
```

A.61 NFAS Warning (level 3) 5163

Declaration for "NAME" hides register declared at FILENAME(NUMBER).

Description: Imported variables and memory blocks share the same name space as registers. If one has the same name as a previously declared register, then uses of the name will reference the import variable or memory block, rather than the register.

Example:

```
.reg port_no
.import_var port_no
alu[--,--,b,port_no] ; references import variable
```

How to fix: There several ways of fixing this. First, the scope of the register can be reduced by enclosing it inside a begin/end block. Also, one of the names can be changed so there is no conflict. For import variables, this can be done by prefixing all instances of the name with 'i$". Note: code running on the core should not be changed as the prefix is stripped off prior to generating the list file. Finally, the warning can always be turned off using the "#pragma warning" directive.

A.62 NFAS Warning (level 2) 5164

No page was declared for this uword -- using default page in default region.

Description: When using microcode overlays, the Assembler implicitly creates an unnamed region. If there is no .page directive preceding the first microword, the Assembler will create a page in the unnamed region, which may not have been the intent. This could indicate that a .page directive was inadvertently left out.

Example:

```
.region one
.region two
; missing .page directive
alu[--,--,b,0]
br_page[l0#]
.page one
alu[--,--,b,0]
br_page[l1#]
```

How to fix: Insert the appropriate .page directive before the first microword of the program.

A.63 NFAS Warning (level 3) 5165

This arithmetic expression has generated two instructions, is this OK?

Description: When the Assembler evaluates simple expressions, it can be assembled into one or two IMMED instructions. In cases when two instructions are generated this warning is produced.

Example:

```
.reg a
a=0x12345
```

How to fix: This might cause problems if the Assembler assignment is placed into defer slot. To avoid this warning use #PRAGMA WARNING to disable it.

A.64 NFAS Warning (level 2) 5166

There are more than one DIRECTIVE_NAME directives in the program!

Description: Some Linker directives may be used only once, e.g. .image_name, ucode_size. The Assembler will flag multiple use of these directives.

Example:

```
.image_name img0
...
.image_name img1
```

How to fix: Avoid multiple usage of the .image_name and .ucode_size directives.

A.65 NFAS Warning (level 1) 5167

A local_csr_wr to an index register is in a defer shadow and consequent use will access old value.

Description: A local_csr_write to an index register is in a branch defer shadow and at the branch target the dependent index is used. There is not enough latency between the write of the CSR and the use of the index register, so the old value will still be used. This is probably unintentional and may indicate a bug in user code.

Example:

```
br[target#], defer[1]
local_csr_wr[ActLMAddr0, 20]
...
target#:
alu[--, tmp, -, *l$index0]
```

How to fix: Either move the local_csr_wr of the index CSR out of the defer shadow or add other instructions before the dependant index register

A.66 NFAS Warning (level 1) 5168

Length operand is not used by this instruction. Please replace it with "--".

Description: A number of extended mode instructions do not use length operand, however most of the instruction addressing modes require one. In this case it is suggested that programmers use “--” instead of any numeric value.

Example:
```
msf0[fast_wr,--,3,a,1]
```

How to fix: Avoid using length operand in instructions that do not require it.

A.67 NFAS Warning (level 1) 5169

Transfer register is not used by this instruction. Please replace it with "--".

A.68 NFAS Warning (level 4) 5170

There are no DRAM registers in extended mode. All transfer registers should be specified as $n or $.

Description: In extended mode there are no separate SRAM and DRAM transfer registers. The Assembler is using a unified transfer register model therefore all $$ or $$n registers are treated similarly to $ or $n register.

Example:
```
dram[test_and_swap,$$1,c,d,2],sig_done[s21]
```

How to fix: Avoid using DRAM transfer register notation.

A.69 NFAS Warning (level 2) 5171

"num_sigs" token will have no effect since no "sig_done" or "ctx_swap" were specified.

A.70 NFAS Warning (level 2) 5172

"uwr" and "urd" instructions will not be supported in the next chip release.

A.71 NFAS Warning (level 1) 5173

Addressing mode token is deprecated. Please use a new 40-bit addressing mode syntax.

A.72 NFAS Warning (level 1) 5174

pragma addressing has been deprecated. Please use a new 40-bit addressing mode syntax.

A.73 NFAS Warning (level 1) 5175

Specified memory block alignment is ignored when a specific memory offset is requested.

A.74 NFAS Warning (level 1) 5176

Constant exceeds 32-bit range.

A.75 NFAS Warning (level 1) 5177

Immediate value could cause an increment of NN_GET.

A.76 NFAS Warning (level 1) 5178

Only signal[0] may be used for an instruction expecting data_error signals.

A.77 NFAS Warning (level 1) 5179

Local symbol shadows wider scope symbol with the same name.

Appendix B. NFAS Error Messages

B.1 Error Messages

This section contains the descriptions of the Error Messages.

Table B.1. Error Messages

Error Number	Message
3001	Arithmetic annotation generated uword(s) can't fit into the defer slot.
3002	Arithmetic expression is misplaced hence invalid.
3003	A DRAM REGISTER transfer register is not allowed because the number of contexts is not 8.
3004	Too many GPRs to fit into A/B Banks.
3005	Too many registers.
3006	Too many signals.
3007	Local Memory Allocation failed.
3008	The address of manually allocated register REGISTERA conflicts with the address of REGISTERB.
3009	The context portion of address 0xVALUE for register is outside the specified bounds.
3010	The address of register must be less than VALUE.
3011	The address for REGISTERA causes the address for REGISTERB to be too large.
3012	The address for REGISTERA causes the address for REGISTERB to be too small.
3013	GPR Register Spilling could not find a viable solution REASON.
3014	A "default" page was declared at LINE for region.
3015	This page cannot be the default page for region, because another page in this region contains the first instruction of the program.
3016	Missing region name for directive.
3017	Too many operands supplied for opcode.
3018	Region was previously declared at LINE.
3019	Region names must start with a letter or an underscore.
3020	Region names must only contain letters, numbers, or underscores.
3021	Undeclared region.
3022	If specified, the second operand must be keyword "default".
3023	Too many operands supplied for directive.
3024	Previous page falls through to this page.
3025	Instruction can not be placed in the shadow of INSTRUCTIONB.
3026	Missing operand for instruction.

Error Number	Message
3027	Too many operands for instruction.
3028	Return register was loaded with "INSTRUCTIONA" instead of "INSTRUCTIONB" at line LINE.
3029	Error in parser.
3030	Odd numeric signal used as a doubled signal.
3031	Signal which is manually allocated at odd address VALUE used as a doubled signal.
3032	Register was not manually allocated.
3033	Signal was not manually allocated.
3034	Invalid Expression.
3035	Expression cannot be converted to alu/immed.
3036	Instruction is not supported by this version of the assembler.
3037	Instruction is only available on the IXP2850.
3038	Microstore Allocation failed.
3039	This uword cannot be used at the maximum end of a deferment shadow.
3040	Directive is no longer supported. Please use "#define" instead.
3041	Unexpected opcode detected.
3042	Unable to open FILE_TYPE file.
3043	Number of uwords exceeds size of microstore.
3044	One or more warnings detected.
3045	Unexpected error occurred during processing phase.
3046	Unable to delete out-of-date .list file.
3047	Insufficient arguments presented on command line.
3048	Output filename is too long.
3049	Output filename is missing for -o option.
3050	Invalid flag on command line.
3051	Invalid revision on command line.
3052	Unknown register/signal name for &remote() function.
3053	Illegal value for option.
3054	Include path is too long.
3055	Include path is missing for -I option.
3056	Symbol definition is missing for -D option.
3057	Command line switch "OLD_SWITCH_NAME" has been renamed "NEW_SWITCH_NAME".
3058	Invalid warning level switch.
3059	Invalid warning switch.
3060	Invalid switch.
3061	Context is missing for option.

Error Number	Message
3062	Invalid context for option.
3063	Local memory start/size is missing for -lm option.
3064	Local memory start value specified in -lm option overflows 32-bit integer.
3065	Local memory size specified in -lm option is too large.
3066	Local memory start/size values specified in -lm option contain unknown characters.
3067	Local memory start address specified in -lm option must be a multiple of 4.
3068	Local memory start address specified in -lm option must be >= 0.
3069	Local memory size specified in -lm option must be a multiple of 4.
3070	Ustore size is missing for option.
3071	Option must precede the first filename.
3072	Input filename is too long.
3073	The optimizer can't be used in SIMPLE mode.
3074	Illegal range specified by options -REVISION_MIN=VALUE and -REVISION_MAX=VALUE.
3075	Command line switch used in environment variable "AS_NFPTYPE" has been renamed.
3076	Local memory ending address must be ≤ MAX_VALUE.
3077	Local memory start address must be < MAX_VALUE.
3078	The -o option is only valid for a single input file.
3079	No input microcode file was specified.
3080	User-specified Ustore Size VALUE exceeds physical limit VALUE.
3081	Start context specified on the command line exceeds end context VALUE.
3082	Cannot use USTORE_MODE microstore because it conflicts with PREVIOUS_USTORE_MODE mode.
3083	Missing index on register.
3084	Register, index exceeds declared length VALUE.
3085	Unexpected index on register.
3086	Invalid transfer register.
3087	Attempt to use undeclared REGISTER_SIGNAL_TYPE.
3088	Invalid Local CSR name.
3089	A signal may not be declared as an aggregate.
3090	Invalid aggregate size "0" for "REGISTER".
3091	Aggregate size for STRING exceeds maximum of VALUE.
3092	Registers are of different fundamental address types and cannot be sequentially allocated.
3093	Registers do not address a common xfer reg space thus, cannot be sequentially allocated.
3094	Register cannot be sequentially allocated after itself.
3095	Registers, REGISTERA and REGISTERB, cannot both be sequentially allocated just above REGISTERC.

Error Number	Message
3096	Multiple sequential allocation to register REGISTERA after both REGISTERB and REGISTERC.
3097	One or more register allocation conflicts exist. Refer to FILENAME for more information.
3098	Register was not manually allocated to a register bank/address.
3099	Register REGISTERA and REGISTERB are manually allocated to the same bank.
3100	Signal/Register only declared "extern".
3101	GPR is explicitly required on both A and B banks.
3102	Signal/Register was already manually allocated at FILENAME:LINE.
3103	The address of Signal must be greater than 0.
3104	The address of Signal must be less than 16.
3105	Register cannot be assigned a negative address.
3106	Register is a neighbor destination reg and cannot be manually allocated.
3107	Register is visible and has the same name as the one defined in FILENAME at VALUE.
3108	Invalid shift specifier.
3109	Invalid import shift specifier.
3110	MEMORY_REGION/REGISTER NAME with an offset of 0 was previously initialized to VALUE.
3111	MEMORY_REGION/REGISTER NAME with an offset of 0 was previously initialized to VALUE for all contexts.
3112	MEMORY_REGION/REGISTER NAME with an offset of 0 was previously initialized to VALUE for context CONTEXT_VALUE.
3113	Block has no non-zero size specified.
3114	Block has no non-zero alignment specified.
3115	The name is reserved and cannot be used.
3116	Invalid constant expression.
3117	Parenthesis may only be used with signals() or &remote() functions.
3118	Address operator applied to non-register.
3119	Address operator can only be applied to Transfer registers or Signals.
3120	Invalid aggregate.
3121	To take the address of part of a doubled signal, please take the address of the signal itself and optionally add one.
3122	Invalid aggregate.
3123	An address_operator cannot be used in this context.
3124	An unresolved constant expression cannot be used in this context.
3125	A symbolic constant cannot be used in this context.
3126	Immediate operand cannot be negative in opcode.
3127	Immediate operand cannot fit in %d-bit field in opcode.

Error Number	Message
3128	"REGISTER" is not a visible register in this microengine.
3129	Signal used in mask() function is not currently active and is used in both a single and doubled manner.
3130	Unexpected operand on token.
3131	Label was instantiated more than once.
3132	Token does not take arguments.
3133	Multiple qualifiers detected.
3134	Unexpected token associated with opcode.
3135	Both left and right operands are constant values.
3136	Indexed neighbor destination registers must have post-increment suffix.
3137	Attempt to use a Read-Modify-Write instruction to a remote neighbor register without a matching local register.
3138	The local neighbor register corresponding to the remote named neighbor register for a Read-Modify-Write instruction was not manually allocated.
3139	Extraneous text found following instruction.
3140	Unexpected syntax.
3141	Indexes cannot use offset and post-modify at the same time.
3142	Post-modify must be applied to destination when index is used as src and dest.
3143	Register types REGTYPEA and REGTYPEB cannot both be source operands.
3144	Register was simultaneously specified on both A and B sources.
3145	Insufficient or invalid operands supplied for opcode.
3146	Invalid import variable name.
3147	Expecting a token specifier.
3148	Expecting a token specifier for operand.
3149	"defer" qualifier has no operands in opcode.
3150	"defer" qualifier has too many operands in opcode.
3151	"defer" qualifier specifies value must be >=0 in opcode.
3152	"defer" qualifier specifies value > VALUE in opcode.
3153	A opcode is inside the "defer shadow" of opcode.
3154	Defer Shadow extends beyond end of code.
3155	Defer Shadow extends beyond end of page.
3156	REGISTER is declared as a write transfer register and is being used as a read.
3157	REGISTER is declared as a read transfer register and is being used as a write.
3158	A path exists whereby condition codes were used for %s before being loaded.
3159	A path exists whereby invalidated condition codes were used for %s due to the context swap at line VALUE.

Error Number	Message
3160	\"%s\" may not immediately follow the P3 branch unless it is in its defer slot.
3162	Dependent instructions "INSTRUCTIONA" and "INSTRUCTIONB" are not both in the defer shadow.
3163	Invalid shift control specifier for opcode.
3164	Specified label does not exist.
3165	Specified label is not in the same page.
3166	Instruction branch to an invalid address.
3167	Previous uword must be an ALU or ALU_SHF opcode in order to specify an indirect shift on current uword.
3168	Left operand of previous uword must be a register to specify indirect shift on current uword.
3169	Previous OPERATOR operator must be "AND", "AND~", "XOR", "OR", "B" or "~B" in order to specify an indirect shift on current microword.
3170	End of file found within %s before the end of block. Enclosing block starts at line LINE.
3171	The constant for the CAM state value must be 0-VALUE.
3172	'SIGNAL' is a reserved signal name and cannot be used.
3173	A signal index is not allowed here.
3174	Invalid index value \"%s\" for signal.
3175	The use of numeric signals is no longer supported. Please use named signals.
3176	Invalid %s name.
3177	Invalid signal name.
3178	"SIGNAL" is an invalid signal for opcode.
3179	%s signal is both doubled and not doubled.
3180	Reference to unreachable code.
3181	Bad compiler-generated input. Could not find the VALUEth compiler-generated microword.
3182	The ctx is outside the specified bounds.
3183	In 4-context mode, the valid contexts are 0, 2, 4, or 6.
3184	Insufficient number of operands supplied for opcode.
3185	A shift specifier is incompatible with the ALU-only operator \"%s\" in opcode.
3186	Operand requires unrestricted addressing, which is incompatible with a shift specifier in opcode.
3187	Immediate %s-hand operand cannot fit in VALUE-bit field in opcode.
3188	No shift specifier supplied for opcode.
3189	The absolute register cannot be specified in the opcode.
3190	%s is not supported by opcode.
3191	Invalid shift amount specified for opcode.
3192	Destination operand must be a register value.
3193	Indexed neighbor registers cannot be used with opcode.

Error Number	Message
3194	Immediate source operand cannot fit in VALUE-bit field in opcode.
3195	Invalid byte enable load specifier for opcode.
3196	Unrecognized opcode.
3197	No destination register specified for opcode.
3198	A recognized numeric specifier was not supplied for immediate data to opcode.
3199	Immediate operand cannot fit in VALUE-bit field in opcode.
3201	Immediate OPERAND operand cannot be negative in opcode.
3202	Immediate OPERAND operand cannot fit in VALUE-bit field in opcode.
3203	Invalid opcode detected.
3204	2nd argument for OPCODE opcode must be in the range VALUE1-VALUE2.
3205	Byte compare data cannot fit in 8-bit field in opcode.
3207	\"%s\" is an invalid context number for opcode.
3208	Context number VALUE must be in the range VALUE1-VALUE2.
3209	\"%s\" is an invalid input state for opcode.
3210	\"%s\" is an invalid branch destination.
3211	Opcode cannot be followed within two instructions by a %s using the same signal.
3212	Opcode cannot be followed within three instructions by a %s using the same signal.
3213	The BCC instruction cannot follow the conditional P3 branch.
3214	An index is not allowed on a non-doubled signal.
3215	SIG+1 should be SIG.
3216	SIG should be SIG+1.
3217	This instruction is not allowed in the shadow of the local_csr_wr at %s because of the intervening I/O instruction at %s.
3218	Invalid CSR name for opcode.
3219	1 cycle is required between this write to \"%s\" and the one on line.
3220	A \"%s\" to \"%s\" may not be placed in the defer slot of a context swapping instruction.
3221	Instruction must be followed by an "immed" instruction.
3222	5 cycles are required between the CRC instruction on line %s and this read of the "crc_remainder" CSR.
3223	No operands exist for qualifier.
3224	LABEL is not a valid label in qualifier of opcode.
3225	Missing "targets" token.
3226	No operands are expected for opcode.
3227	Invalid shift specifier for OPCODE opcode: must begin with ">>" or "<<".
3228	Invalid shift specifier for OPCODE opcode: must consist of ">>" or "<<" followed by a number.
3229	Invalid source operand for opcode.

Error Number	Message
3230	Illegal aggregate element. Only entire aggregates may be specified in directives.
3231	An index register cannot be used in the opcode.
3232	\"%s\" is not a transfer register.
3233	All registers in %s must be declared with the same scope.
3234	Third parameter must specify "a" or "b" register bank.
3235	Undeclared %s \"%s\".
3236	Register is not a GPR and cannot be manually allocated to the bank.
3237	Register is not an reg and cannot be manually allocated as such.
3238	Import variable was previously defined.
3239	Import variable conflicts with previous use by .operand_synonym directive.
3240	Memory block names must start with alphabetical character or underscore.
3241	Invalid Memory block name.
3242	Insufficient or invalid operands supplied for directive.
3243	Register may only appear in one .xfer_order directive.
3245	Invalid region for opcode.
3247	Incorrect memory type for opcode, use ".local_mem".
3248	Memory block size must be positive and greater than zero.
3249	Memory block alignment must be positive and greater than zero.
3250	Memory block alignment for REGION region must be a multiple of 4.
3251	Scratch Block size exceeds size of scratch memory.
3252	Memory region was previously defined.
3253	Memory region was previously defined as %s to %s microengine(s).
3254	Memory region was previously defined for a different memory region type.
3255	Attributes for Memory region conflict with earlier definition.
3256	\"%s\" is not a valid context number for directive.
3257	Memory block/register names must start with [a-z_$@].
3258	Invalid register.
3259	Unknown memory block.
3260	Unknown memory block/register.
3261	"VARIABLE" is an imported variable.
3262	\"%s\" is an operand_synonym.
3263	Invalid offset for memory block.
3264	Offset must be a multiple of 4.
3265	Garbage after memory block name.
3266	Register initializations can only take one value.

Error Number	Message
3267	Local registers cannot be initialized.
3268	Register is not a context-relative register.
3269	Initializations extend beyond memory block.
3270	Directives cannot nest.
3271	%s encountered within block.
3272	%s is not within directive.
3273	Register is listed multiple times in opcode.
3274	The number of blocks is limited to VALUE.
3275	Keyword appears within register name list.
3276	Keyword "remote" cannot appear with other keywords.
3277	Keyword "extern" can only appear with "read" or "write".
3278	Keywords "read" and "write" cannot be used with the opcode.
3279	Keyword "doubled" cannot be used with the opcode.
3280	Keyword "doubled" cannot be used on a non-volatile signal.
3282	The number of local regions is limited to VALUE.
3283	Attempt to declare a %s with the same name as a symbolic constant.
3284	Cannot declare registers.
3285	Cannot use "visible", "read", "write", or "remote" keywords when declaring GPRS.
3286	Cannot use "read" or "write" keywords when declaring neighbor regs.
3287	No register names supplied for opcode.
3288	Invalid register name: Prefix 'a$' is reserved for internal use.
3289	Declaration conflicts with declaration at %s.
3290	Extern %s %s conflicts with %s declared at %s %d.
3291	The Read/Write attributes of %s must agree with the declaration at %s %d.
3292	Redundant definition for %s: previously declared.
3293	Unexpected compiler-generated fiid value.
3294	Unexpected text.
3295	Invalid number of contexts for opcode.
3296	Number of contexts specified conflicts with earlier value.
3297	Local Memory mode must be "abs" or "rel".
3298	Local Memory Mode specified conflicts with earlier value.
3299	Init NN mode must be "self" or "neighbor".
3300	Init NN Mode specified conflicts with earlier value.
3301	The directive takes one operand.
3302	Invalid register name.

Error Number	Message
3303	Use of %s with transfer register.
3304	Register may not be referenced within 3 cycles of its use in the "URD" at %s.
3305	Invalid signal/transfer register name.
3306	Specifier list must be in quotes for directive.
3307	Second operand must be "on" or "off" for directive.
3308	Performance optimizations were not turned off.
3309	Branch/ctx_arb bubbles optimizations were not turned off.
3310	Branch target optimizations were not turned off.
3311	NOP optimizations were not turned off.
3312	A/B bank conflict fixing was not turned off.
3313	Register spilling was not turned off.
3314	Invalid specifier for "#pragma warning".
3315	Specifier cannot be used with other specifiers.
3316	Extraneous parameters for specifier.
3317	Unknown type %s for opcode.
3318	Unknown specifier for "#pragma warning".
3319	Extraneous text following specifier for "#pragma warning".
3320	Illegal level value for "push" specifier.
3321	Directive was already found at line LINE.
3322	Unknown token for opcode.
3323	Unexpected operands for directive.
3324	Directive should have two operands.
3325	Second operand of directive should be a number, as instruction address.
3326	The operand of directive should be a number.
3327	Insufficient or invalid operands supplied for compiler-generated directive.
3328	Expecting "--" for return register in compiler-generated ".scope function" directive.
3329	Expecting "NIL" for return uword list in compiler-generated ".scope function" directive.
3330	Only one compiler-generated ".scope global" directive may be specified.
3331	Missing compiler-generated directive.
3332	Too many compiler-generated ".scope end" directives.
3333	"br" qualifier for instruction requires an address label operand.
3334	"br" qualifier for instruction takes exactly 1 address label operand.
3335	"LABEL" is not a valid label for the "br" qualifier for instruction.
3336	No operands were found for opcode.
3337	Signals VOLUNTARY, KILL, BPT, and -- cannot be combined %s.

Error Number	Message
3338	Tokens cannot be used together.
3339	At least one signal must be specified for opcode.
3340	The "io_completed" optional token can only be used with ctx_arb[--].
3341	Insufficient or invalid operands supplied for token.
3342	For a doubled signal, \"%s+1\" was specified. Please use \"%s\" instead.
3343	For a doubled signal whose low half was consumed, \"%s[1]\" must be specified.
3344	For opcode \"%s+1\" should be \"%s\".
3345	Signal should be \"%s[%d]\".
3346	Signal should be \"%s[%s]\".
3347	Use of old style reflector tokens is not supported with "ind_targets".
3348	Qualifier has no operands.
3349	Qualifier has too many operands.
3350	SIG_DONE and CTX_SWAP tokens specified for opcode.
3351	This opcode/command cannot use the "ctx_swap" token because it uses two signals.
3352	DEFER qualifier cannot be specified without CTX_SWAP qualifier for opcode.
3353	The IND_TARGETS token can only be used with INDIRECT_REF.
3354	Index registers cannot be the target of an I/O operation.
3355	Transfer register must be specified as "--".
3356	A valid transfer register must be specified for opcode.
3357	Ref count starting at xfer reg, "REGISTER" does not completely map into a previously specified contiguous region of xfer regs (incomplete .xfer_order).
3358	An SRAM REGISTER transfer register is not allowed because the number of contexts is not 8.
3359	The reference is not preceded by a uword which forms an indirect specifier.
3360	A command must be specified for opcode.
3361	Ref Count must be >= 1.
3362	Ref Count must be <= %d.
3363	Max Ref Count must be >= 1.
3364	Max Ref Count must be <= %d.
3365	Ref Count must be a number or max_<number>.
3366	\"%s\" is not an transfer register for opcode.
3367	%s write transfer register is not allowed for opcode.
3368	"sig_remote" qualifier needs Microengine numbers for opcode.
3369	Microengine number for SIG_REMOTE must be numeric.
3370	Illegal remote microengine address.
3371	No arguments specified for token.

Error Number	Message
3372	Operand for token is not numeric.
3373	SIG_REMOTE with operands cannot be used with old-style reflector tokens.
3374	The tokens sig_initator, sig_remote, and sig_both are mutually exclusive.
3375	A signal must be specified.
3376	This instruction is context-swapping but not sending itself a signal.
3377	TARGETS token is required with SIG_REMOTE for opcode.
3378	TARGETS token is invalid with SIG_INITIATOR for opcode.
3379	Invalid value for FLAGS token argument.
3380	Invalid value for TGT_CMD field.
3381	Max_nn ref count is only valid with indirect_ref.
3382	Invalid command for opcode.
3383	The token is no longer supported.
3385	Token "no_pull" for opcode requires indirect_ref token.
3386	Token cannot be used with command %s%s for opcode.
3387	Command requires a ref count of at least 2.
3388	Command for opcode requires indirect_ref token.
3389	Token is not allowed for command for opcode.
3390	Token is not allowed for command with no transfer registers for opcode.
3393	A transfer register must be specified unless it is overridden with "indirect_ref".
3394	"REGISTER_NAME" is not an register for opcode.
3395	Ref Count must be in the range 1..3.
3396	Fast write data must fit in %d bits.
3397	Fast write data must be a number or "ALU".
3398	Previous uword does not generate ALU output.
3399	Invalid csr for opcode.
3400	CSR is obsolete. Please use "CSR_NAMEB" instead.
3401	Invalid csr for opcode.
3402	Invalid csr for opcode: Invalid Register Number for REGISTER.
3403	Token cannot be used with command for opcode.
3404	It is illegal to FAST_WR to the %s CSR.
3405	\"%s\" is not a valid Remote_xfer.
3406	A DRAM transfer register is not allowed for write operations for opcode.
3407	A DRAM remote transfer register is not allowed for read operations for opcode.
3408	Remote Ctx must be -- or in 0...7.
3409	A valid remote context is needed.

Error Number	Message
3410	With local CSRS, the remote context should be "--".
3411	Max_nn ref count is only valid with indirect_ref.
3412	Microengine from SIG_REMOTE token does not agree with microengine from instruction.
3413	Invalid CAM name.
3414	Invalid CAM size.
3415	Declaration for CAM %s.
3416	Opcode requires VALUE operands.
3417	Implicitly declared CAM.
3418	Only one of TOKEN tokens is allowed.
3419	Qualifier has no operands for opcode.
3420	Qualifier has too many operands in opcode.
3421	Qualifier has a non-numeric operand in opcode.
3422	Value specified in qualifier must be >=0 for opcode.
3423	Value specified in qualifier must be <= VALUE for opcode.
3424	A instruction with a "lm_addr" token may not be placed in the defer slot of a context swapping instruction.
3425	Incorrect number of operands supplied for opcode.
3426	CRC type operand is missing.
3427	Invalid CRC type for opcode.
3429	1 cycle is required between this CRC instruction and the one on line LINE.
3430	An aborted CRC instruction may not immediately follow another if the CRC type or bytes token differ from the preceding instruction.
3431	Instruction must be preceded by a priming instruction.
3433	Did not find a preceding instruction.
3434	Mismatching endianness for \"%s\" and \"%s\" at line LINE.
3435	Shift operand for opcode must be a right shift.
3436	Opcode requires that the previous microword must be an ALU or ALU_SHF opcode in order to set the shift-in value.
3437	Opcode requires that the previous ALU or ALU_SHF operation be a "b", "~b", "and", "~and", "and~", "xor", or "or".
3438	Instruction must be preceded by another instruction.
3439	Type \"%s\" for instruction does not match previous type.
3440	Step for instruction does not follow previous step.
3441	Operand differs from previous operand for opcode.
3442	Indexed neighbor registers cannot be used as the destination operand for opcode.
3443	Increment/Decrement is only allowed on the final step (i.e. the step before the "last" token).

Error Number	Message
3444	Missing token for opcode.
3445	Incompatible tokens specified for opcode.
3446	Instruction must be followed by another instruction.
3447	Instruction must be preceded by instruction.
3448	Source operand does not match previous source operand at line.
3449	Post-modify of index registers is only allowed on \"pop_count%d\".
3450	Missing dest op in opcode.
3451	First operand must be a GPR or a Local Memory Index.
3452	Operand must be \"*u$index0++\" or \"*u$index1++\".
3453	The CSR may not be written within 3 instructions of the URD instruction at line.
3454	A maximum of 4 "URD" instructions may be issued consecutively.
3455	Instruction may not be used in the 4 instructions preceding the URD at line.
3456	Instruction may not be used in the 3 instructions preceding the URD at line.
3457	The CSR may not be read here because it is referenced by the instruction at line.
3458	A multi-step instruction sequence must complete by the 3rd instruction following the first URD in a set.
3459	Instruction may not be used within 3 cycles of the "URD" at %s.
3460	Instruction may not be used within 3 cycles of a URD.
3461	A maximum of 7 "UWR" instructions may be issued consecutively.
3462	A multi-step instruction sequence must complete by the 6th instruction following the first UWR in a set.
3463	Instruction may not be used within 6 cycles of the \"%s\" at %s.
3464	Instruction may not be used within 6 cycles of "UWR".
3465	Invalid csr for opcode: Invalid ME for %s.
3466	Invalid nfp3200 legacy option - only nfp3216 are supported.
3467	The constant for the CAM entry number must be 0-VALUE.
3468	Instruction is only available on NFP32xxc processors.
3469	Invalid specifier in pragma addressing. Valid specifiers are "32bit" and "40bit".
3470	Ambiguous command addressing tokens. Only one address token can be specified.
3471	Invalid command target name %s.
3472	Invalid action for target.
3473	Immediate action for target needs indirect_ref token.
3474	Crypto unit is not supported in legacy mode.
3475	Instructions are supported only in extended mode.
3476	\"%s\" is supported only in legacy mode. Use "-legacy" switch to assemble this instruction.
3477	\"%s\" addressing mode cannot have indirect_ref token since it is always used indirect.

Error Number	Message
3478	\"%s\" is supported only in extended mode.
3479	Contradictory tokens.
3480	Signals are not supported for immediate address and data mode.
3481	Immediate data mode must be used with indirect_ref token.
3482	Missing second operand for immediate data and address format.
3483	"TOKEN" requires indirect ref token.
3484	Token cannot be used with target and action.
3485	Register is found in the transfer register position with action which does not require a transfer register. \"%s\" addressing mode is being used.
3486	Endian mode must be either "big" or "little".
3487	\"%s\" requires length of 1.
3488	Previous uword does not generate ALU output - third party addressing is always using indirect.
3489	Incorrect type of memory lock operation specified in the length field. Valid options are:1,2,3,4. Found: ACTUAL_VALUE.
3490	Invalid number of dram signals for opcode.
3491	Number of dram signals specified conflicts with earlier value.
3492	Invalid or missing value for "num_sigs" token. Valid syntax: num_sigs[n] where n is 1 or 2.
3493	\"%s\" requires length of 3 or 7.
3494	Transfer register is not expected with no_pull token.
3495	Incorrect max_nn number. Valid max_nn numbers are: \"%s\".
3496	Incorrect syntax for 40-bit addressing mode. Valid syntax: target[action,Xfer,src1,<<8,src2,cnt] or target[action,Xfer,src1,src2,<<8,cnt].
3497	NFP3200 does not support action for command.
3498	Structured assembly directives .endif/.endw have to be followed by at least one instruction.
3499	Local memory allocation request for \"%s\" is outside local memory address range.
3500	Local memory block overlaps other memory block(s). Check offset parameter in .local_mem directives.
3501	Memory offset value must be positive.
3502	Attributes for Memory offset conflict with earlier definition.
3503	Invalid parameters for directive. Valid parameter is PARAMETER.
3504	Alignment specified in the directive conflicts with the earlier value.
3505	DRAM locality specified in the directive conflicts with the earlier value.
3506	Ref Count must be multiple of 2.
3507	Xfer reg is not part of a contiguous region of xfer regs (missing .xfer_order).
3508	Third party addressing mode must be "32bit" or "40bit".
3509	Third party addressing mode conflicts with earlier setting.

Error Number	Message
3510	Indirect ref mode must be "ixp_compatible" or "nfp_v1".
3511	Indirect ref mode conflicts with earlier setting.
3512	%s conflicts with assembler option.
3513	Incorrect syntax for 32-bit third party addressing mode. Valid syntax: target[action,thirdparty,src1,src2,cnt].
3514	Incorrect syntax for 40-bit third party addressing mode. Valid syntax: target[action,thirdparty,src1,<<8,src2,cnt] or target[action,thirdparty,src1,src2,<<8,cnt].
3515	Memory block alignment of VALUE is invalid for offset.
3516	Constant exceeds 32-bit range.
3517	Constant exceeds signed 32-bit range.
3518	Invalid operand for directive.
3521	A signal index[1] is not allowed here.
3522	40-bit addressing mode can't be used with IXP compatible setting.