



# **Netronome ® Network Flow Processor 6xxx**

**NFP SDK version 5**

## **Flow Processor Core Programmer's Reference Manual**

**- Proprietary and Confidential - - ccs:3d9198b6b223 -**

# **Netronome ® Network Flow Processor 6xxx : Flow Processor Core Programmer's Reference Manual**

Copyright © 2006-2018 Netronome Systems, Inc. - Proprietary and Confidential. All rights reserved.

## **COPYRIGHT**

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

## **WARRANTY**

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

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

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

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

## **LIABILITY**

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

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

## Revision History

Date	Revision	Author	Description
June 3, 2011	v0.01 - cs:?	R. Fortino	Initial Release of EAS Document Template
Oct 12, 2012	v0.1 - cs:?	M. Secules	Initial compilation of the NFP-6xxx PRM
June 10, 2013	v0.2 - cs:?	Netronome	NFP-6xxx PRM Alpha Release
June 23, 2013	v0.3 - cs:?	Netronome	NFP-6xxx PRM Post Alpha Release. Add missing instructions, Update Glossary, Update address map pics, add Unrestricted/ Restricted addr mode table in 2.1.1
June 24, 2013	v0.4 - cs:?	Netronome	Update MEM(Ring put, journal,..) get/put_freely description, Update I/O instr src_op1 src_op2 descr to include Restricted source operand, change lb_push_stats_* ref_cnt to 1 to 16, fix format in Table 2.3
June 25, 2013	v0.5 - cs:?	Netronome	mem[Lookup Engine] update, mem[microQ] table bits 4 and 5 switched, mem[ticket] update
June 26, 2013	v0.6 - cs:?	Netronome	Update CPP_commands address descr, Update fast_journal_sig descr, Add memory_lock, update Glossary, add four svg images
June 30, 2013	v0.7 - cs:?	Netronome	Update Figure 2.5-2.6, added mem[lookup] section Notes pertaining to address bit definitions for lookup operation and table configuration
July 08, 2013	v0.8 - cs:?	Netronome	Add links to notes, Update Figure 2.6
July 09, 2013	v0.9 - cs:?	Netronome	Change in PRM all bit declarations to common format[x:y], Update CPP_Command Fields, Add missing commands to MEM(Ring) commands table, Update mem(lookup Engine) for addr bits, Remove ntest_cmds
July 11, 2013	v0.10 - cs:?	Netronome	Update section 2.1.6.4 Signal Pair with Swap examples.
July 17, 2013	v0.11 - cs:?	Netronome	Consistent use of opt_tok, fixed ref_cnt error in interthread_signal, opt_tok category for interthread_signal
July 24, 2013	v0.12 - cs:?	Netronome	Add note to mem[inc and like, commands ref_cnt --, replaced with Not Required, Add length to MU Immed.
July 31, 2013	v0.13 - cs:?	Netronome	Fixed Glossary typo, Remove reference to Yeldham in section 2.1.3.3.8, In CLS(Hash Operaiotnions) changed undefined reference to NFP-6xxx Databook, Re-worded intro sentence in section 2.2.40 Memory Unit and replaced flavor with variant in Table 2.7. Renamed titles of MU External nums 0-3. Add note to CLS reflect section 2.2.37.6 that only CSRs and Xfer In/Out registers are supported.
Aug 08, 2013	v0.14 - cs:?	Netronome	Rename IND_CSR to CMD_INDIRECT_REF_0, Fixed PCIe ref_cnt in cmd instruction, Update reference document names in Related Documents section, Improve CPP command description.
Aug 11, 2013	v0.15 - cs:?	Netronome	Decouple I/O and non-I/O opt_tok and created new section for non I/O Instructions.

Aug 14, 2013	v0.16 - cs:?	Netronome	Add NFP-6xxx Indirect Ref Format mode description to section 2.2.40.5 MEM(List Enqueue) Note 1, along with example code.
Aug 25, 2013	v0.17 - cs:?	Netronome	Move Non-I/O opt_tok from I/O Instr Format section to new section. In CLS Ring opers specify ring index for each instruction and add Notes explanation at end of table. Add NN_FULL to section 2.1.4.4.9. Add CLS test_add64 and statistic_imm to ref_count command field. Remove CLS subimm, test_addsat64_imm and test_subsat64_imm from Table 3.1
Sept 7, 2013	v0.18 - cs:?	Netronome	Re-write CT instructions section and MEM(Stats). Fix typo of ref_cnt sentence in several places in PRM, Fix Figure 3.9 to show 3MB regions. Add table heading in scrtion 2.1.4.1 Instructions that support no_cc token.
Sept 16, 2013	v0.19 - cs:?	Netronome	Add sequence encode in ring_add_to_tail_ptr description, Extended description for cam_lookup_add_lock, cam_lookup_add_extend, cam_lookup_add_inc, add cam_lookup24_add_extend, cam_lookup24_add_lock. MEM(Packet Engine Operations) pe_dma_to_memory... commands - correction in ref_cnt and xfer registers used. Removed pe_dma_to_memory_buffer_free_le from MEM(Packet Engine Operations) list of commands.
Sept 25, 2013	v0.20 - cs:?	Netronome	Update mem(lookup) ref_cnt description. Replace NBI with Packet Processing Subsystem in Table 2.1 Summary of FPC Instructions and in section 2.2.43. Add more info in forming source operands for queue array accesses. Revert pe_ema_to_memory_indirect_xx commands made in V0.18, and changed ref_cnt to 1 and moved to Rd XFER registers.
Oct 3, 2013	v0.21 - cs:?	Netronome	Update to MEM(Packet Engine Operations) for src_op and xfer fields to include reference to databook and more description.
Oct 9, 2013	v0.22 - cs:?	Netronome	Remove read_pci and write_pci from PRM, update src_op1/2 fields.
Nov 4, 2013	v0.23 - cs:?	Netronome	Update MEM[Load Balance] and ARM[] ref_cnt.
Nov 18, 2013	v0.24 - cs:?	Netronome	Added ILA TCAM instruction to opt Tok row, remove SRAM[CSR_xx] instructions, add deprecated note to SRAM instructions.
Dec 08, 2013	v0.25 - cs:?	Netronome	Update CLS Statistics add to value not overwrite value. Remove cls[queue_lock] xfer register update on error (deprecated), CLS[reflect] addr encoding changed CTX to signal CTX for clarity. cls[ringops] ring_add_to_tail_ptr, ring_read, ring_write ring_ordered_lock/unlock specify bit[20] must be set. Also add_to_tail_ptr must be supported, add to PRM. mem[queuelock] add 40-bit address support.
Dec 13, 2013	v0.26 - cs:?	Netronome	Update mem[] commands sytnax to show 32-bit and 40-bit addressing, remove reference to saturate in mem[add], add back in read_pci and write_pci.
Jan 06, 2014	v0.27 - cs:?	Ed Lombardo	Add mem[microq], ct[reflect], cls[reflect] example code snippets, update ct[reflect] address encode table for 40-bit address and description of address fields.

Jan 09, 2014	v0.28 - cs:?	Ed Lombardo	Move add_tail in ref_cnt up with ring_add_to_tail_ptr, change CLS ring Operations reference to ref_cnt should start with 1 instead of 0, Change mem(Work Queue operations) work queue now supports 16 32-bit long words
Jan 19, 2014	v0.29 - cs:?	Ed Lombardo	Changed mem CmpAndWr and CmpAndWr32 to Deprecated and referenced to new mnemonic. Fix mem(CAM/TCAM) Table alignment error. Fixed wording in lookup-engine hashLUTable note.
Jan 22, 2014	v0.30 - cs:?	Ed Lombardo	Add mem[cam], mem[tcam] and cls[cam] and cls[tcam] examples. Update mem[tcamxxx_lookup24/32] incorrectly shows correct number of bits compared
Jan 26, 2014	v0.31 - cs:?	Ed Lombardo	Add xpb[r/w] examples. Corrected description for mem[compare_write_or_incr], adjusted crypto sections to align all non-crypto sections in both crypto and non-crypto PRMs. Added back Bytemask limit to [6:0] for imm atomic non-arithm operations in mem[atomic] section Note 3.
Jan 31, 2014	v0.32 - cs:?	Ed Lombardo	Better explained max_nn used in Changing the Reference Count Using Indirect References. CLS Atomic instructions test_and_clr_imm and test_and_sub_imm description changed to refer to replacement commands. Corrected 2.2.37.5 CLS(Lock Queue Operations) Example- replace xfer with --. Specify in CLS Ring Queue Lock/Unlock addr[31:20] should be 0x1, MEM Atomic changes to mem_compare and friends, Change Reference Documents section to be more inclusive.
Feb 05, 2014	v0.33 - cs:?	Ed Lombardo	Change CLS[test_compare_write] description push-back data is unconditional, CLS[incr/incr64/decr/decr] description to include sub/sub_imm is used. Correct MU write8_swap_le to indicate little endian. Update address map for CTM Reflect for base address and M calculation. Specify indirect_ref to override ref_cnt in various places that were missing (mem[ring operations], mem[rd/wr]). CTM reflect master number starts at FPC+4. CLS Ringops add note how rings are configured via CLS address space. Add notes for data_master limitations in NFP-32xx indirect ref formats. Add note where to find Island and Master IDs in NFP-6xxx databook.
Feb 09, 2014	v0.34 - cs:?	Ed Lombardo	Add Programming Examples section to PRM, added IPv4 Processing and TCP checksum examples. Update figure 3.25 XPB CTM Reflector NN Memory MAP. Add in cls[reflect] section other_data_master is FPC+4. CT[reflect] add reference to databook for island#.
Feb 16, 2014	v0.35 - cs:?	Ed Lombardo	Add flags in PRM to hide ILA TCAM references, remove Neuron references in PRM, Improve ref_cnt description in 2.1.6.2, Add missing note to commands when ref_cnt is greater than 8 use indirect_ref. Add example code for MU Lists and Ring Queues, Correct mem[stats] ref_cnt limit.
Feb 23, 2014	v0.36 - cs:?	Ed Lombardo	Fix sub_sat group to show unsigned value. Add Note that explains ref_cnt in terms of immed or indirect influence on CPP LENGTH. Update CLS ref_cnt immed. Update CLS reflect list for additional commands. Fix mem[atomic] ref_cnt min/max for test commands.

			Added Thornham to Harrier Coding Recommendations section to PRM. Add ME-ME example code in CT[] section.
Feb 27, 2014	v0.37 - cs:?	Ed Lombardo	Add PCIe example code. Add Immediate Addressing Mode section describing 5 addressing modes. Add in section Indirect References to Another FPC example code for NFP-6xxx.
Mar 4, 2014	v0.38 - cs:?	Ed Lombardo	Corrected in mem(Work Queue operations) ref_cnt from 4 to 16. Change ALU_SHF left shift indirect to include B and ~B.
Mar 7, 2014	v0.39 - cs:?	Ed Lombardo	Corrected CLS ring_write ref_cnt to 1-32.
Mar 12, 2014	v0.40 - cs:?	Ed Lombardo	Improved description for ref_cnt usage for the immed atomic with readback vs non readback and added separate Note 4. Reordered the Notes for CLS(Atomic) section. Added note for Immed Address/ Data to point out this is used for debug.
Mar 14, 2014	v0.41 - cs:?	Ed Lombardo	Remove CSL ring_put_offset and put_offset, not supported in RTL.
Mar 23, 2014	v0.42 - cs:?	Ed Lombardo	MEM atomic commands- Add Note 8, modify notes 3 and 4.
Mar 26, 2014	v0.43 - cs:?	Ed Lombardo	MEM atomic commands test_addsat and test_subsat missing reference to Note 8.
Mar 30, 2014	v0.44 - cs:?	Ed Lombardo	Fix CLS TCAM lookup32 example for mistakes in compare example and Table 2.25. Also add more explanation to bit mask result.
Jun 10, 2016	v0.50 - cs:?	M. Secules	Add missing commands. Add Warnings and Notes to draw attention to special requirements.

# Table of Contents

<b>1. Introduction .....</b>	11
1.1. Scope .....	11
1.2. Related Documents .....	11
1.3. Terminology .....	12
1.4. Definitions .....	13
<b>2. NFP-6xxx FPC Instruction Set .....</b>	14
2.1. Instruction Syntax .....	18
2.1.1. Source and Destination Selection .....	19
2.1.2. Two Source Operand Selection Rules .....	20
2.1.3. Non-I/O Optional Tokens (opt_tok) .....	21
2.1.4. Condition Codes .....	22
2.1.5. Branch Defer (defer[n]) .....	24
2.1.6. I/O Instruction Format .....	26
2.1.7. Coding Restrictions .....	47
2.1.8. NFP-6xxx FPC Permitted Coding Sequences .....	55
2.2. Instruction Set .....	57
2.2.1. ALU .....	57
2.2.2. ALU_SHF .....	59
2.2.3. ASR .....	60
2.2.4. BCC (BRANCH CONDITION CODE) .....	62
2.2.5. BR .....	63
2.2.6. BR_BCLR, BR_BSET .....	63
2.2.7. BR=BYTE, BR!=BYTE .....	64
2.2.8. BR_CLS_STATE, BR !_CLS_STATE .....	64
2.2.9. BR=CTX, BR!=CTX .....	65
2.2.10. BR_INP_STATE, BR !_INP_STATE .....	66
2.2.11. BR_SIGNAL, BR !_SIGNAL .....	67
2.2.12. BYTE_ALIGN_BE, BYTE_ALIGN_LE .....	68
2.2.13. CAM_CLEAR .....	71
2.2.14. CAM_LOOKUP .....	72
2.2.15. CAM_READ_TAG .....	73
2.2.16. CAM_READ_STATE .....	74
2.2.17. CAM_WRITE .....	75
2.2.18. CAM_WRITE_STATE .....	75
2.2.19. CRC LE, CRC BE .....	76
2.2.20. CTX_ARB .....	78
2.2.21. DBL_SHF .....	80
2.2.22. FFS .....	81
2.2.23. HALT .....	82
2.2.24. IMMED .....	83
2.2.25. IMMED_B0, IMMED_B1, IMMED_B2, IMMED_B3 .....	84
2.2.26. IMMED_W0, IMMED_W1 .....	85
2.2.27. JUMP .....	86
2.2.28. LD_FIELD, LD_FIELD_W_CLR .....	87
2.2.29. LOAD_ADDR .....	88
2.2.30. LOCAL_CSR_RD .....	89
2.2.31. LOCAL_CSR_WR .....	90
2.2.32. MUL_STEP .....	91

2.2.33. NOP / NOP_VOLATILE .....	93
2.2.34. POP_COUNT .....	94
2.2.35. RTN .....	94
2.2.36. ARM .....	95
2.2.37. Cluster Local Scratch (CLS) .....	96
2.2.38. Cluster Target (CT) .....	133
2.2.39. Memory Unit (MU) .....	151
2.2.40. SRAM .....	234
2.2.41. Interlaken Look Aside (ILA) .....	255
2.2.42. Packet Processing Subsystem .....	256
2.2.43. PCI Express (PCIe) .....	257
2.2.44. Crypto .....	260
<b>3. FPC Address Map .....</b>	<b>262</b>
3.1. Functional Units .....	267
3.2. Target Memory Maps .....	270
3.2.1. ARM Memory Map .....	271
3.2.2. PCIe Memory Map .....	271
3.2.3. ILA Memory Map .....	273
3.2.4. NBI Memory Map .....	273
3.2.5. MU External Memory Map .....	274
3.2.6. MU Direct Access Memory External Memory Map .....	277
3.2.7. MU Direct Access Memory Internal Memory Map .....	278
3.2.8. MU Direct Access Target CTM Memory Map .....	279
3.2.9. CLS Direct Access Memory Map .....	285
3.2.10. XPB Target Addressing .....	291
3.2.11. Crypto Memory Map .....	301
<b>4. Control and Status Registers .....</b>	<b>303</b>
4.1. Local Control and Status Registers .....	303
4.2. Microengine Local Control and Status Registers .....	303
4.2.1. Detailed Register Descriptions .....	306
<b>5. Programming Examples .....</b>	<b>341</b>
5.1. IPv4 Packet processing .....	341
5.2. TCP Header Checksum .....	345
<b>6. NFP-32xx to NFP-6xxx Coding Recommendations .....</b>	<b>351</b>
<b>Glossary .....</b>	<b>352</b>
<b>7. Technical Support .....</b>	<b>358</b>

## List of Figures

2.1. Load Immediate .....	83
2.2. Dequeue Buffer .....	239
2.3. Example of the Three Dequeue Modes .....	240
2.4. Enqueue One Buffer at a Time Using the Enqueue Command .....	242
2.5. Enqueue a String of Buffers to a Queue .....	243
2.6. Read Queue Descriptor Commands .....	249
2.7. Write Queue Descriptor Commands .....	254
3.1. ARM Memory Map .....	271
3.2. PCIe Memory Map .....	272
3.3. ILA Memory Map .....	273
3.4. NBI Memory Map .....	274
3.5. MU External 0 Memory Map .....	275
3.6. MU External 1 Memory Map .....	276
3.7. MU External 2 Memory Map .....	277
3.8. MU Direct Access Memory External Memory Map .....	278
3.9. MU Direct Access Memory Internal Memory Map .....	279
3.10. MU Direct Access ARM CTM Memory Map .....	280
3.11. MU Direct Access PCIe CTM Memory Map .....	281
3.12. MU Direct Access FPCx CTM Memory Map .....	282
3.13. MU Direct Access ILA CTM Memory Map .....	283
3.14. MU Direct Access Self Island CTM Memory Map .....	284
3.15. MU Direct Access Crypto CTM Memory Map .....	285
3.16. CLS Direct Access ARM Memory Map .....	286
3.17. CLS Direct Access PCIe Memory Map .....	287
3.18. CLS Direct Access FPC Memory Map .....	288
3.19. CLS Direct Access ILA Memory Map .....	289
3.20. CLS Direct Access Self Island Memory Map .....	290
3.21. CLS Direct Access Crypto Memory Map .....	291
3.22. XPB CTM Inter-thread Memory Map .....	293
3.23. XPB CTM Reflector CSR Memory Map .....	294
3.24. XPB CTM Reflector XFER Memory Map .....	295
3.25. XPB CTM Reflector Next-Neighbor Memory Map .....	296
3.26. XPB CTM Reflector Rings Memory Map .....	297
3.27. XPB Chip Level Memory Units Only Part 1 .....	298
3.28. XPB Chip Level Memory Units Only Part 2 .....	299
3.29. XPB Chip Level Non Memory Islands Only Part 1 .....	300
3.30. XPB Chip Level Non Memory Islands Only Part 2 .....	301
3.31. Crypto Memory Map .....	302
5.1. IPv4 Header Fields .....	342
5.2. TCP Pseudo-Header Fields .....	346

## List of Tables

2.1. Summary of FPC Instructions .....	14
2.2. Source/Destination Choices for Addressing Modes .....	19
2.3. Source/Destination Choices for Addressing Modes (UnRestricted/Restricted) .....	20
2.4. Legal Combinations of A and B Operands .....	21
2.5. Non-I/O Command Token Descriptions .....	21
2.6. Execution Unit Condition Code Values by Instruction Type .....	23
2.7. List of Instructions that support no_cc token .....	24
2.8. Branch Defer Summary .....	26
2.9. CPP Addressing Modes .....	28
2.10. Reference Count Sizes .....	29
2.11. I/O Command Token Descriptions .....	30
2.12. CPP Command - cpp_command Source Fields .....	30
2.13. NFP Legacy Mode Indirect Reference Formats .....	38
2.14. Previous ALU Result - PREV_ALU .....	40
2.15. Previous ALU Result Description .....	41
2.16. Indirect CSR - CMD_INDIRECT_REF_0 .....	43
2.17. Indirect CSR Description .....	43
2.18. Instructions and Optional Tokens that Use Signals .....	44
2.19. Signal Restrictions for Each I/O Instruction [command] .....	45
2.20. Branch on Condition Code Instructions .....	62
2.21. Initial Register Contents .....	69
2.22. Initial Register Contents .....	69
2.23. CAM_LOOKUP Result .....	73
2.24. CAM_READ_STATE Result .....	74
2.25. TCAM Table layout for cls[tcam_lookup32] example .....	131
2.26. TCAM Table layout for mem[tcam512_lookup32] example .....	174
2.27. Type 0 - List descriptor .....	175
2.28. Type 0 - Link format for NFP enqueue and dequeue command .....	175
2.29. Type 0 - Link format in memory when using ME to construct chain of packets .....	176
2.30. Type 1 - List descriptor .....	177
2.31. Type 1 - Link format for NFP enqueue and dequeue command .....	177
2.32. Type 1 - Link format in memory when using ME to construct chain of packets .....	177
2.33. Type 2 - List descriptor .....	178
2.34. Type 2 - Link format for enqueue command .....	179
2.35. Type 2 - Link format from dequeue command .....	179
2.36. Type 3 - List descriptor .....	179
2.37. Type 3 - Link format for NFP enqueue and dequeue command .....	179
2.38. Type 3 - Link format in memory when using a ME to construct chain of packets .....	180
2.39. Ring Size Encoding for type 0, 1 and 3 queue .....	217
2.40. Ring Type Encoding .....	217
2.41. Memory Unit Addressing Modes .....	218
2.42. Ring Size encoding for Type 2 Rings .....	220
2.43. SRAM Ring Descriptor Format .....	250
2.44. SRAM Ring Size Encoding .....	250
3.1. FPC I/O Access .....	262
4.1. Microengine Local Control and Status Register Latencies .....	303
4.2. MeCsrCPP Address Map .....	306

# 1. Introduction

---

## 1.1 Scope

This manual is a reference for microcode programming of the Netronome Network Flow Processor NFP-6xxx. The intended audience for this book includes developers and system programmers.

This preliminary early access release of the manual is organized as follows:

- [Chapter 2](#) describes the micro-instruction set and provides example microcode.
- [Chapter 3](#) describes the FPCs memory view of the functional units.
- [Chapter 4](#) describes the internal registers and provides examples of their use.
- [Chapter 7](#) provides information on technical support.

## 1.2 Related Documents

Descriptive Name	Description
Netronome ® Network Flow Processor NFP-6240/6320/6480-xC Datasheet,	Contains summary information on the Netronome Network Flow Processor NFP-6xxx including a functional description, signal descriptions, electrical specifications, and mechanical specifications.
Netronome ® NFP-6xxx-xC Network Flow Processor Databook	Contains detailed reference information on the Netronome Network Flow Processor NFP-6xxx.
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 Assembler System User's Guide	Describes the syntax of the NFP-6xxx's assembly language, supplies assembler usage information, and lists assembler warnings and errors.
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 Compiler LibC: Reference Manual	Specifies the subset and the extensions to the language that support the unique features of the Netronome Network Flow Processor NFP-32xx product line.

## 1.3 Terminology

Acronym	Description
3DES	Triple DES
AES	Advanced Encryption Standard
API	Application Programming Interface
CAM	Content Addressable Memory
CLS	Cluster Local Scratch
CPP	Command Push Pull Bus. Used within an island to provide the 'last mile' connectivity to masters. The CPP interconnect consists of buses for Command, Push Data and Pull Data.
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSR	Control and Status Registers
DA	Descriptor Added
DES	Data Encryption Standard
DMA	Direct Memory Access
DSF CPP	Distributed Switch Fabric CPP interconnect. The DSF CPP interconnect is the main global bus system in the Netronome Flow Processing Architecture.
ECC	Error Correcting Code
FIFO	First In First Out
FPA	Flow-Processor Architecture
FPC	Flow-Processor Core
GPIO	General Purpose IO
GPR	General Purpose Register
HWM	High Water Mark
ILA	Interlaken Look-Aside
I/O	Input / Output
LM	Local Memory
LRU	Least recently used
LSB	Least Significant Byte/Bit
LWBE	Little-Endian Word, Big-Endian bytes
microcycle	FPC core clock cycle to execute one instruction.
microword	Microcode instruction word
MRU	Most recently used
MU	Memory Unit

Acronym	Description
NBI	Network Block Interface
NFAS	Network Flow Assembler System
NFLD	Network Flow Linker
NFP	Netronome Network Flow Processor
NPCC	Netronome Packet Processing Cores
PCIe	PCI Express
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver/Transmitter
UDR	Unlinked Descriptor Read
VQDR	Virtual Quad Data Rate
XFER	Transfer

## 1.4 Definitions

Virtual register is a named register declared in code without the need to define its physical location. The assembler will allocate virtual registers to physical registers based on scope and live-range. For the purposes of this document, the word *scope* of a virtual register refers to that portion of the flattened input source in which it is possible to refer to that virtual register. The *live-range* (or *lifetime*) of a virtual register refers to that portion of the code where that register contains a value that will be used later. Generally, the live-range extends from when the register is set to where that value is last used.

Note that the scope and live-range may not coincide. In particular, the live-range may extend outside of the scope due to, for example, a subroutine call. Even if the subroutine is outside of the scope of the register, the register may be live within the subroutine. Similarly, the live-range may be smaller than the scope.

Transfer registers come in two variants. One is a *read* or *in* transfer register. These are registers that may be used as the source of an ALU operation. The name comes from the fact that they typically are used for read I/O operations. The other is a *write* or *out* transfer register. These may be used as a destination of an ALU operation. They are typically used for write I/O operations. A virtual register may be defined as a *read* transfer register, as a *write* transfer register, or as a *R/W* (or *both*) register. In the first two cases, the virtual register is allocated from either the read or write banks of transfer registers. In this third case, the virtual register is allocated in both the read and write banks at the same address. These are needed for I/O operations that do both reads and writes at the same time. Historically, all transfer registers fell into this category.

Flow Processor Core (FPC) and Microengine (ME) are interchangeable between this document and other NFP-6xxx documents.

## 2. NFP-6xxx FPC Instruction Set

For quick reference, the FPC instruction set is summarized and grouped by function in [Table 2.1](#). In the sections that follow, the instructions are defined in alphabetical order.

**Table 2.1. Summary of FPC Instructions**

Instruction	Description	Section
<b>General Instructions</b>		
ALU	Perform an ALU operation.	<a href="#">Section 2.2.1</a>
ALU_SHF	Perform an ALU and shift operation.	<a href="#">Section 2.2.2</a>
ASR	Perform an arithmetic right shift on a register.	<a href="#">Section 2.2.3</a>
BYTE_ALIGN_BE, BYTE_ALIGN_LE	Concatenate data in two registers and put any four bytes into the destination register.	<a href="#">Section 2.2.12</a>
CRC LE, CRC BE	Compute CRC LE (Little Endian). Compute CRC BE (Big Endian).	<a href="#">Section 2.2.19</a>
DBL_SHF	Concatenate two 32-bit words, shift the result, and save a 32-bit word.	<a href="#">Section 2.2.21</a>
FFS	Determine position number of LSB set in a register.	<a href="#">Section 2.2.22</a>
IMMED	Load immediate 16-bit word and sign-extend or zero fill with shift.	<a href="#">Section 2.2.24</a>
IMMED_BO, IMMED_B1, IMMED_B2, IMMED_B3	Load immediate byte to a field.	<a href="#">Section 2.2.25</a>
IMMED_WO, IMMED_W1	Load immediate 16-bit word to a field.	<a href="#">Section 2.2.26</a>
LD_FIELD, LD_FIELD_W_CLR	Load byte(s) into specified field(s).	<a href="#">Section 2.2.28</a>
LOAD_ADDR	Load instruction address.	<a href="#">Section 2.2.29</a>
LOCAL_CSR_RD, LOCAL_CSR_WR	Read and write FPC Local CSRs.	<a href="#">Section 2.2.30</a> , <a href="#">Section 2.2.31</a>
MUL_STEP	Multiply two unsigned numbers.	<a href="#">Section 2.2.32</a>
NOP	No operation and virtual no operation.	<a href="#">Section 2.2.33</a>
POP_COUNT	Determine the number of bits set in a register.	<a href="#">Section 2.2.34</a>
<b>Branch and Jump Instructions</b>		
BCC	Branch on condition code.	<a href="#">Section 2.2.4</a>
BR	Branch unconditionally.	<a href="#">Section 2.2.5</a>
BR_BCLR, BR_BSET	Branch on bit clear or bit set.	<a href="#">Section 2.2.6</a>
BR=BYTE, BR!=BYTE	Branch on byte equal or not equal to literal.	<a href="#">Section 2.2.7</a>

Instruction	Description	Section
BR_CLS_STATE, BR_!CLS_STATE	Branch if the state of the specified cluster local scratch state name is equal to 1 or equal to 0.	<a href="#">Section 2.2.8</a>
BR=CTX, BR!=CTX	Branch on current context.	<a href="#">Section 2.2.9</a>
BR_INP_STATE, BR_!INP_STATE	Branch on event state (e.g., SRAM done).	<a href="#">Section 2.2.10</a>
BR_SIGNAL, BR_!SIGNAL	Branch if signal asserted or de-asserted.	<a href="#">Section 2.2.11</a>
JUMP	Jump to label.	<a href="#">Section 2.2.27</a>
RTN	Return from a branch or a jump.	<a href="#">Section 2.2.35</a>
<b>I/O and Context Swap Instructions</b>		
ARM	Read or write from/to the address of the shared memory block or peripherals inside the ARM11 subsystem.	<a href="#">Section 2.2.36.1</a>
CLS (Atomic Operations)	Issue a memory reference to perform an atomic operation on data in Cluster Local Scratch memory.	<a href="#">Section 2.2.37.1</a>
CLS (CAM Lookup)	Treat area of Cluster Local Scratch SRAM as Content Addressable Memory and perform lookup on it.	<a href="#">Section 2.2.37.2</a>
CLS (CSR)	Move data between FPCs and Cluster Local Scratch CSRs.	<a href="#">Section 2.2.37.3</a>
CLS (Hash Operations)	Create a 64-bit hash index over a set of transfer register values.	<a href="#">Section 2.2.37.4</a>
CLS (Lock Queue Operations)	Issue an atomic and exclusive enqueue-and-lock/unlock-and-dequeue operation on a Cluster Local Scratch lock-queue variable.	<a href="#">Section 2.2.37.5</a>
CLS (Read and Write)	Move data between FPCs and Cluster Local Scratch memory.	<a href="#">Section 2.2.37.8</a>
CLS (Reflect)	Issue a reflect operation via Cluster Local Scratch to access transfer registers of other FPCs in the current cluster.	<a href="#">Section 2.2.37.6</a>
CLS (Ring Operations)	Add entries to or remove entries from a ring in Cluster Local Scratch SRAM.	<a href="#">Section 2.2.37.7</a>
CLS (TCAM Lookup)	Treat area of Cluster Local Scratch SRAM as a TCAM and perform lookup on it.	<a href="#">Section 2.2.37.9</a>
CT (Cluster Target)	The Cluster Target MU Misc Engine instructions.	<a href="#">Section 2.2.38.1</a>
CTX_ARB	Perform context swap and wake on event.	<a href="#">Section 2.2.20</a>
HALT	Puts the current thread to sleep without waking up any other thread and asserts the FPC's ATTN output signal.	<a href="#">Section 2.2.23</a>

<b>Instruction</b>	<b>Description</b>	<b>Section</b>
ILA (Interlaken Look-Aside)	Interlaken Look-Aside subsystem communicates to external TCAMs, search Processors, FPGA accelerators, or other Thornham type devices that provide specialized search or processing operations. ILA commands provide software an interface to these above external devices.	<a href="#">Section 2.2.41.1</a>
MEM (Atomic Operations)	Logic and arithmetic commands.	<a href="#">Section 2.2.39.1</a>
MEM (CAM and TCAM lookup)	Allow two types of operations on CAM and TCAM – add new entries and lookup existing entries.	<a href="#">Section 2.2.39.2</a>
MEM (Lists)	Four types of Link lists, type 0, 1, 2 and 3	<a href="#">Section 2.2.39.3</a>
MEM (Lists Dequeue)	This command removes a segment or a buffer from the queue and is only applicable to lists.	<a href="#">Section 2.2.39.4</a>
MEM (Lists Enqueue)	Add buffer to queue and update queue tail pointer. Applicable to only Lists.	<a href="#">Section 2.2.39.5</a>
MEM (Load Balance Engine Operations))	The Load Balance Engine in the MU provides software with acceleration for dynamic load balancing. The Load Balance Engine accepts commands to aid in dynamic load balancing of flows travelling over the same pipe.	<a href="#">Section 2.2.39.6</a>
MEM (Lookup Engine Operations)	The Lookup Engine in the Memory Unit allows for commands to perform various lookup table operations and data matching functions.	<a href="#">Section 2.2.39.7</a>
MEM (Memory Lock)	Support locking of specific ranges of memory as opposed to semantic locking supported by queue locking commands.	<a href="#">Section 2.2.39.8</a>
MEM (MicroQ Operations)	Manage a 128 or 256 bits cache line as a FIFO or LIFO lists with 32-bit or 16-bit entries.	<a href="#">Section 2.2.39.9</a>
MEM (Packet Engine Operations)	The Packet Engine buffers moves and queues packet data.	<a href="#">Section 2.2.39.10</a>
MEM (Push Queue Descriptor)	Reads an entire queue descriptor into the read transfer registers.	<a href="#">Section 2.2.39.11</a>
MEM (Queue)	Supports two classes of queues – linked lists and circular buffers.	<a href="#">Section 2.2.39.12</a>
MEM (Queue Lock Operations)	Manage a 128 or 256 bit cache line as a FIFO of semantic lock claimants.	<a href="#">Section 2.2.39.13</a>

<b>Instruction</b>	<b>Description</b>	<b>Section</b>
MEM (Read and Write)	Move data between the memory unit and the FPC transfer registers or other data masters.	<a href="#">Section 2.2.39.14</a>
MEM (Read/Write Queue Descriptor)	Applicable both to lists and rings. Moves queue array descriptor in memory to/from queue array entry.	<a href="#">Section 2.2.39.15</a>
MEM (Ring Add to Tail)	Adds a specified amount to the ring tail pointer, wrapping it appropriately.	<a href="#">Section 2.2.39.16</a>
MEM (Ring Put, Journal, Get and Pop Operations)	These commands are only applicable to rings (circular buffers) and add (put / journal) or remove from head (get) or tail (pop) a requested number of entries from a circular buffer.	<a href="#">Section 2.2.39.17</a>
MEM (Rings/Circular Buffers)	Configure a ring of queue descriptors of ring types 0, 1, 2, and 3.	<a href="#">Section 2.2.39.18</a>
MEM (Rings Fast_journal)	Applicable to only Rings.	<a href="#">Section 2.2.39.19</a>
MEM (Rings Queue Read and Write)	Read or write the contents of a queue.	<a href="#">Section 2.2.39.20</a>
MEM (STATS Operations)	The Stats Engine is a Memory Unit subcomponent to logging statistics. These commands are provided to the Stats Engine to handle statistics processing for the Memory Unit.	<a href="#">Section 2.2.39.21</a>
MEM (Ticket Release)	Used for reordering management.	<a href="#">Section 2.2.39.22</a>
MEM (Work Queue Operations)	Enable work to be scheduled among a pool of threads.	<a href="#">Section 2.2.39.23</a>
NBI (Packet Processing Operations)	The Packet processing subsystem interfaces to the Ethernet MAC and Interlaken ports of the MAC block and processes packets on Ingress and Egress. The NBI provides a wide array of dedicated hardware to support the high bandwidth processing. NBI commands allow the FPC to transmit packets via the TM and write and read packets to/from the CTM or MU. Refer to the Netronome NFP-6xxx Databook Packet Processing Subsystem section.	<a href="#">Section 2.2.42.1</a>
PCIe	Issue a request to the PCI Express unit.	<a href="#">Section 2.2.43.1</a>
SRAM (Read and Write)	Move data between the FPCs and SRAM. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.5</a>
SRAM (Atomic Operations)	Issue a reference to perform an atomic operation on data in SRAM in NFP-6xxx.	<a href="#">Section 2.2.40.1</a>

<b>Instruction</b>	<b>Description</b>	<b>Section</b>
	These commands are deprecated in NFP-6xxx.	
SRAM (Read Queue Descriptor)	Issue a memory reference to an SRAM Channel to read the Queue Descriptor into the SRAM. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.6</a>
SRAM (Write Queue Descriptor)	Issue a memory reference to an SRAM Channel to move data from the Queue Descriptor into SRAM. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.8</a>
SRAM (Enqueue)	SRAM enqueue. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.3</a>
SRAM (Dequeue)	SRAM dequeue. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.2</a>
SRAM (Ring Operations)	Issue an SRAM Ring put or get command to the SRAM. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.7</a>
SRAM (Journal Operations)	Issue an SRAM Journal command to the SRAM Channel. These commands are deprecated in NFP-6xxx.	<a href="#">Section 2.2.40.4</a>
<b>CAM Instructions</b>		
CAM_CLEAR	Clears all entries in the FPC CAM.	<a href="#">Section 2.2.13</a>
CAM_LOOKUP	Search the FPC CAM for the tag value.	<a href="#">Section 2.2.14</a>
CAM_READ_TAG	Read the tag for the specified CAM entry into the destination register.	<a href="#">Section 2.2.15</a>
CAM_READ_STATE	Read the State bits for the specified CAM entry.	<a href="#">Section 2.2.16</a>
CAM_WRITE	Write a value to the tag for the specified CAM entry.	<a href="#">Section 2.2.17</a>
CAM_WRITE_STATE	Write the value for the State bits into the specified FPC CAM entry.	<a href="#">Section 2.2.18</a>

## 2.1 Instruction Syntax

This section describes the syntax for the NFP-6xxx FPC instruction set.

## 2.1.1 Source and Destination Selection

Many instructions specify one or two source operands, and/or one destination operand. The choice of source and destination addressing is specified for each of those instructions as either Context-Relative, Absolute/Relative or Indexed.

- Context-Relative means the only choice is Context-Relative Mode. Context-Relative mode specifies Context-Relative register for a Context, thus allowing up to 8 different context to share the same microcode and maintain separate data.
- Absolute/Relative means that the instruction allows either Absolute Mode or Context-Relative Mode. Absolute addressing means that any GPR can be read or written by any one of the eight Contexts in an FPC. There is no instruction that allows only Absolute Mode.
- Indexed means Indexed Addressing Mode that uses CSR pointers for accessing Register Files. There are 2 classes of indices: Class 1: those which only have an "active" version of the pointer; Class 2: those which have an "active" and an "indirect" version of the pointer.

Refer to the *Databook* section "Addressing Modes" for further details on the three Addressing Modes and how they are used.

[Table 2.2](#) shows which register type operands that can be accessed in each mode.

**Table 2.2. Source/Destination Choices for Addressing Modes**

Register Type	Context-Relative	Absolute	Indexed
GPR	Yes	Yes	No
Transfer	Yes	No	Yes
Neighbor	No	No	Yes
Local Memory	No	No	Yes

8-bit Immediate data can also be specified as a source. The immediate data is supplied in the cmd[] type instructions like mem[] and others, and can range from 0 to 0x7F. Larger immediate data are constructed with IMMED instructions, which are defined in [Section 2.2.24](#).

All instructions that specify a destination can also specify “No Destination” (using the -- notation). This is typically used to set ALU condition codes for branches without modifying any registers.



### Note

When a Transfer Registers is used as source, the Transfer **In** Register is used. When a Transfer Register is used as a destination, the Transfer **Out** Register is written.

The choice of source and destination addressing can be specified for each instruction as either Restricted or Unrestricted (see [Table 2.3](#) ).

- Unrestricted indicates that there are enough bits in the instruction encoding to support all the possible options for source and destination addressing.
- Restricted indicates that there are fewer bits and only a subset of the unrestricted options are possible.

**Table 2.3. Source/Destination Choices for Addressing Modes (UnRestricted/Restricted)**

Register Type	Unrestricted Addressing	Restricted Addressing
GPR	Context-Relative Absolute	Context-Relative
Transfer Registers	Context-Relative Indexed	Context-Relative Indexed
Neighbor	Context-Relative Indexed	Indexed
Local Memory	Indexed Offset (0-15)	Offset (0-7), Note 5
Other	immed no dest “--”	immed no dest “--”

**Notes:**

1. 8-bit Immediate data can be specified as a source for both Unrestricted and Restricted with the exception of I/O instructions which use 7-bit immediate data.
2. When a transfer register is used as source, the read transfer register is used. When a transfer register is used as a destination, the write transfer register is written.
3. The notation “--” when used as a source means no source; used for single-operand instructions.
4. The notation “--” when used as a destination means no destination; typically used to set Condition Codes.
5. The post increment and post decrement operations are considered index mode operations and therefore cannot be performed in the restricted address mode.

All instructions that specify a destination can also specify “No Destination” (using the “--” notation). This is typically used to set ALU condition codes for branches without modifying any registers.

## 2.1.2 Two Source Operand Selection Rules

For instruction types that specify two source operands, there are some restrictions as to which registers can be used. [Table 2.4](#) shows the legal combinations of A and B operands. For example, the “No” in column 3, row 3 means that two Transfer Registers can not be used as source operands in an instruction.

**Table 2.4. Legal Combinations of A and B Operands**

A Source	B Source					
	GPR (A Bank)	GPR (B Bank)	Transfer	Neighbor	Local Memory	Immed
GPR (A Bank)	No	Yes	Yes	Yes	Yes	Yes
GPR (B Bank)	Yes	No	Yes	Yes	Yes	Yes
Transfer	Yes	Yes	No	No	Yes	Yes
Neighbor	Yes	Yes	No	No	Yes	Yes
Local Memory	Yes	Yes	Yes	Yes	No	Yes
Immediate	Yes	Yes	Yes	Yes	Yes	No

The above can be summarized by three rules:

- The same source cannot be used as both A and B operands.
- Any of Transfer, and Neighbor cannot be used as both A and B operands.
- Immediate cannot be used as both A and B operands.

### 2.1.3 Non-I/O Optional Tokens (opt\_tok)

Non-I/O instructions are ALU type instructions. The table below lists the Non I/O optional tokens along with short description. Tokens specific to an instruction are define in the instruction definition.

**Table 2.5. Non-I/O Command Token Descriptions**

Token	Description
gpr_wrboth	By specifying gpr_wrboth with instruction types which specify a destination GPR a dual write-back of the result of the targeted execution unit to the same address at both banks (A and B) is performed. The destination field carries the single address where the GPR write is to take place in both banks. The actual bank name (A or B) that appears in the destination field of the affected instruction may be either A or B.
load_cc	Load ALU condition codes based on result formed. If not specified then CC is unchanged. Used with instructions: LD_FIELD and LD_FIELD_W_CLR.
no_cc	Instructions that update the condition codes may be disabled with the No Condition Code (no_cc) opt_tok. When no_cc is specified with an instruction condition codes are not updated.
predicate_cc	The condition code prior to instruction execution will be compared with the one specified in the encoded value of the context-specific IndPredCC CSR. If this CC is the same as the one specified in the IndPredCC CSR, then the write-back to the destination register will be performed and the flags will be updated. If this CC is NOT the same as the one specified in the IndPredCC CSR, then the write-back to the destination register will NOT be performed and the flags will not be updated.

### 2.1.3.1 Introduction to GPR Write Both opt\_tok

GPRs are physically partitioned into two independent banks: GPR A and GPR B. Instructions using GPR as a destination, specifically refer to either A or B bank, Programmers may specify for an instruction that the write-back be directed to the same address at both banks. GPR Write Both (gpr\_wrboth) opt\_tok will cause dual write-back of data and the actual bank name (A or B) that appears in the destination field of the instruction is irrelevant; either bank name may be used.

### 2.1.3.2 Introduction to No Condition Code opt\_tok

Most instructions that normally update the condition codes (N,Z,C,V) may use the No Condition Code (no\_cc) opt\_tok. When an instruction has no\_cc specified the condition codes are NOT updated. Refer to [Section 2.1.4.1](#).

### 2.1.3.3 Introduction to Predicate Condition Code

The optional token predicate\_cc (Predicate Condition Code) is new to Mev2.8. This optional token applies only to instructions that write-back to the destination specified in the instruction. When the instruction is executed with predicate\_cc specified in opt\_tok field, the condition code prior to instruction execution will be compared with the one specified in the encoded value of the pre-configured context-specific IndPredCC CSR. Depending upon the encoded value in the IndPredCC register, one of the following two conditions will occur:

- If the CC is the same as the one specified in the IndPredCC CSR, then the write-back to the destination will be performed and the flags will be updated.
- If the CC is the NOT the same as the one specified in the IndPredCC CSR, then the write-back to the destination will NOT be performed and the flags will NOT be updated.

The following are valid examples using the predicate\_cc optional token:

```
alu[gpr12, 0x7f, AND~, gprb2], predicate_cc
cam_write_state[ ], predicate_cc
asr[ ], predicate_cc
```

## 2.1.4 Condition Codes

The FPC architecture defines four Condition Codes, which are updated by some instructions during normal pipeline execution (specifically, in the P4 stage); as in many other microprocessors, the Condition Codes are mainly used by branch instructions. The defined Condition Codes are: Negative (N), Zero (Z), Carry Out (C), and Overflow (V). [Table 2.6](#) presents a relationship between the instruction set and the Condition Codes; that is, it shows specifically which instructions update the Condition Codes.



### Note

The FPC features one single set of Condition Codes which are shared among all active contexts. This means that when a given context swaps out, its updates of the Condition Codes are not guaranteed to be preserved for its eventual swapping-in: other enabled

contexts may update the Condition Codes while the context in question is swapped out. However, if only one context is enabled, then the Condition Codes remain constant in between swapping windows (from the time the context swaps out to the time it swaps back in).

The **CondCodeEn** Local CSR can be used to prevent condition code updates for debugging. This allows the debugger to execute instructions without modifying the condition codes.

**Table 2.6. Execution Unit Condition Code Values by Instruction Type**

Instruction Type	Operation	N	Z	V	C	
ALU	A + B	Result[31] == 1	Result[31:0] == 0	Set if a signed overflow occurs. Note 1	Carry Out from Adder[31]	
	A +16 B					
	A +8 B					
	A + B + prev_carry					
	A - B					
	B - A					
ALU, ALU_SHF	B	Result[31] == 1	Result[31:0] == 0	Cleared	Cleared	
	~B					
	A AND B					
	~A AND B					
	A AND ~B					
	A OR B					
	A XOR B					
	Arith_right_shift					
	CRC					
	FFS (B)	B[31]	B == 0			
	pop_count1	Cleared	Result[31:0] == 0			
	pop_count2					
	pop_count3					
Multiply	Last or Last2	Result[31] == 1	Result[31:0] == 0	Set if a signed overflow occurs.	Carry Out from Adder[31]	
LD_FIELD (if load_cc bit is set; otherwise unchanged)		Result[31] == 1	Result[31:0] == 0	Cleared	Cleared	
Command		Unchanged	Unchanged	Unchanged	Unchanged	
CAM			Unchanged			
IMMED			Unchanged			
LD_FIELD w/o load_cc token			Unchanged			

Instruction Type	Operation	N	Z	V	C
BR, BCC, BR=CTX, BR_SIGNAL, BR=BYTE, BR_bclr, BR_bset, CTX_ARB, JUMP, RTN					
Local_CSR_Rd, Local_CSR_Wr					
1. Logically equivalent to Carry from Adder[31] XOR Carry from Adder[30].					

### 2.1.4.1 NO\_CC Token

Most instructions that normally update condition codes may specify this optional token; however, the following instructions cannot be used with this token: ld\_field and ld\_field\_w\_clr. When this token is specified with an instruction, the condition codes are not updated; this is useful for data movement among register files without affecting the condition codes previously computed by other instructions. For example:

```
alu_shf[$w1, gpr, or, gpr1, <>rot31], no_cc
```

After executing this ALU instruction, no condition codes will be changed due to the presence of “no\_cc” token.

The table below list all instructions where this token can be used together with restrictions imposed on several instructions.

**Table 2.7. List of Instructions that support no\_cc token**

Instruction name	Restrictions
alu	
alu_shf	
asr	
byte_align_be	
byte_align_le	
dbl_shf	
crc_be	
crc_le	
ffs	
mul_step	Can be specified only with 24x8_last, 16x16_last, 32x32_last, and 32x32_last2 tokens.
pop_count	Can be used only with pop_count3.

### 2.1.5 Branch Defer (defer[n])

Any instruction that makes a branch decision may cause one or more instructions in the execution pipeline to be aborted. An instruction that makes a branch decision does so based on the result of an operation that occurs

in either the P1, P2, P3, or P4 instruction pipeline stage. The specific pipeline stage where the decision is made depends on the instruction.

The purpose of a deferred branch is to reduce or eliminate aborted instructions in the execution pipeline. In a deferred branch, an instruction that follows a branch decision is allowed to execute before the branch takes effect (i.e., the effect of the branch is “deferred” in time). If useful work can be found to fill the wasted cycles after the branch instruction, the branch latency can be hidden.

Deferred branches are supported using the “defer” optional token within an instruction. The Assembler’s optimizer can automatically insert the deferred token and re-arrange the instructions or the programmer can do it manually.

### Example 2.1. Defer[n] Branch Taken

```
alu[temp2,temp1,-,temp3] ; always executed
bgt[branch_taken_label#], defer[3] ; (assume branch taken)
alu[temp,temp,+,temp2] ; always executed (would have been aborted)
alu[temp,--,B,temp2] ; always executed (would have been aborted)
alu[--,temp,-,temp3] ; always executed (would have been aborted)
alu[temp,--,B,temp1] ; not executed (branch was taken)
```

[Table 2.8](#) shows the legal defer options for all branch types. The “no defer” column gives the number of instruction cycles lost on a taken branch if no defer token is used. Each deferred instruction reduces that penalty by one. The “defer [n]” columns indicate if that defer amount is allowed for the given branch type.

**Table 2.8. Branch Defer Summary**

Branch Type	Penalty on Branch Taken with no defer	Defer allowed?		
		defer [1]	defer [2]	defer [3]
Unconditional Branch	2	Yes	Yes	No
Context Swap <sup>a</sup> , <sup>b</sup>	2	Yes	Yes	No
Conditional Branch when condition was set earlier than prior instruction	2	Yes	Yes	No
Conditional Branch when condition was set in prior instruction	3	Yes	Yes	Yes
Jump	4	Yes	Yes	Yes
RTN	4	Yes	Yes	Yes
Branch on Input State	4	Yes	Yes	Yes
Branch on Signal	4	Yes	Yes	Yes
BR_Bit	4	Yes	Yes	Yes
BR_Bit	4	Yes	Yes	Yes

<sup>a</sup> Context Swap applies to I/O reference instructions that support the ctx\_swap optional token and the ctx\_arb instruction. These instructions are special cases of branch instructions since they put a thread to sleep and branch to the instruction for the next context.

<sup>b</sup> When a prior instruction is Branch on input state, Branch on signal, BR\_Bit, or BR\_Bit, then the Penalty on Branch Taken with No Defer = 3.

## 2.1.6 I/O Instruction Format

The NFP-6xxx FPC instruction set contains a class of instructions that are referred to as I/O instructions. The I/O instructions include:

- CLS
- Crypto
- ILA
- MEM
- NBI
- PCIe
- SRAM

All these instructions generate a command that is issued to an ARM, Cryptography Unit, Cluster Local Scratch, ILA Unit, Memory Unit, Packet Processing Unit, PCI Express interface, SRAM Controller, and all have the common properties described in this section.

### 2.1.6.1 CPP Addressing Mode

The NFP-6xxx architecture provides 5 addressing modes from most general to the most resource demanding to the least resource demanding. When performance is critical consider the available addressing modes, otherwise use 40-bit addressing mode.

40-bit Addressing mode is the most general but it's also the most resource intensive in that it uses both transfer registers (used to transfer the data) and GPRs (used to specify the address).

32-bit Address mode is similar to 40-bit, but top 8 bits are zero thus directing the action to the local island (Island 0).

Immediate Address mode saves GPRs, thus limits the size of the address that can be used with the instruction.

Immediate Data mode saves a transfer register thus limits the size of data burst to 1.

Immediate Address/Data mode saves on transfer registers and two GPRs, thus the least resource demanding of the group. Immediate Address/Data is mainly used for debugging and that you can not set token or length.

Table 2.9 below shows the address modes and corresponding instruction format

**Table 2.9. CPP Addressing Modes**

Addressing Mode	Description	Format
40-bit	Create 40 bit address with src_op1 + src_op2. Left shift either operand to create 40 bit address.	target[action, xfer, src_op1, <<8, src_op2, ref_cnt] target[action, xfer, src_op1, src_op2, <<8, ref_cnt]
32-bit	Create 32 bit address with src_op1 + src_op2.	target[action, xfer, src_op1, src_op2, ref_cnt]
Immediate Address	Use 16 bit immediate in src_op1. In this case only the CTM can be accessed because IMEM and EMEM requires setting the upper address bits.	target[action, xfer, src_op1, ref_cnt]  Example: mem[write, \$xfer[0], 0x80FF, 2]
Immediate Data	Address is in Previous ALU 16-bit value. This is the only mode where prev_alu is used for address.	target[action, immed22, --, 1]  Where immed22 is 22-bit immediate data value.  Example:  alu[--, --, b, addr]  mem[write, 0x200000, --, 1]
Immediate Address/Data	Address is immediate 16 bit value and data is 14 bit immediate value. Note: Used primarily for debugging and one can not set token or length.	target[action, immd14, immAddr16]

### 2.1.6.2 Source Operands (src\_op1, src\_op2)

Two source operand parameters are required to form an address for I/O instructions (arm, crypto, cls, ila, mem, nbi, pci, and sram). The address is formed by adding src\_op1 + src\_op2.

Either src\_op1 or src\_op2, but not both may be modified with  $<<8$  to indicate that it is to be shifted up by 8 bits before the add operation, allowing for the description of a 40-bit address ( $\text{src\_op1} << 8 + \text{src\_op2}$ ) or ( $\text{src\_op1} + \text{src\_op2} << 8$ ).

The aggregate field is pushed to the target in the initial CPP command. Generally one of the registers is used as the base address of the data and the second as an offset to a particular data element.

The rules specified in [Table 2.4](#) are imposed when specifying src\_op1 and src\_op2.

### 2.1.6.3 Reference Count (ref\_cnt)

The reference count specifies the burst size of the reference. The reference count specified in the instruction is usually 1 to 8, where larger counts can be specified using an indirect reference (indirect\_ref). Also reference counts from 1 up to 32 can be specified using an indirect reference. How ref\_cnt value is interpreted is

dependent on the I/O instruction and the command. When ref\_cnt is 1 and the command operates on 8 bytes (a word), the data is moved to or from two FPC transfer registers.

**Table 2.10. Reference Count Sizes**

Ref_cnt Increments	I/O Instructions (Commands)
4-byte word	ARM (Registers), CLS, ILA, MEM (most atomic operations), PCIe, SRAM
8-byte word	ARM, Crypto, MEM (read, write), NBI
Other	CLS / MEM non-ALU atomics, e.g. CAM/TCAM operations

### Note



In most cases the ref\_cnt specified in the micro-code instruction encodes to the CPP LENGTH field as:

$$\text{LENGTH} = (\text{ref\_cnt} - 1)$$

Here ref\_cnt is indirectly working with CPP LENGTH field. But once indirect reference (indirect\_ref) is used to override the CPP LENGTH field, you are no longer working with ref\_cnt but with CPP LENGTH directly.

For the rare cases, CPP LENGTH field is directly loaded with ref\_cnt value (Immediate).

$$\text{LENGTH} = \text{ref\_cnt}$$

See Cluster Local Scratch, MEM(Memory Lock) and MEM(Packet Engine Operations) sections of the PRM **ref\_cnt descriptions** field for identifying these commands that use ref\_cnt immediate.

### Note



Any instructions that specify ref\_cnt as “Not Required” or existing code that uses “--” is encouraged to use the following syntax:

cmd[mnemonic, xfer, address, offset]

instead of

cmd[mnemonic, xfer, address, offset, --]

## 2.1.6.4 Optional Tokens (opt\_tok)

Most of the I/O instructions support optional tokens. The table below lists I/O optional tokens along with a short description. Tokens specific to an instruction are defined in the instruction definition.

**Table 2.11. I/O Command Token Descriptions**

Token	Description
ctx_swap[sig_name]	Put the thread to sleep and wake it when the specified signal event (sig_name) is asserted. Specify either ctx_swap or sig_done, not both.
defer[m]	Execute m instruction(s) following this instruction before branching or switching contexts. The value of m is dependent on the instruction and can range from 1 to 3. Only used with ctx_swap (not with sig_done).
ignore_data_error	Signal the FPC thread upon completion if a data error occurs on a read operation. If this token is not specified, the FPC thread is not signalled.
indirect_ref	Use the ALU output of the previous instruction to redefine the instruction. The format of the ALU output data of the previous instruction is dependent on the I/O instruction and is defined in the individual I/O instruction definitions. There are two indirect_ref modes: V1 mode and V2 mode where V2 indirect_ref mode is the default.
ind_targets[fpc1, fpc2, ...]	Informs the assembler that the target of an I/O instruction is in a different FPC (FPC1, FPC2, ....). As a result, any generated signals and transfer registers are considered to be remote and are looked up in the targets. To use this optional token, the indirect reference must override the FPC number.
no_pull	Traditional SRAM is replaced with the Memory Units VQDR target, allowing existing NFP-32xx software to port to NFP-6xxx FPA. The no_pull opt_tok specified with select SRAM atomic commands provides the immediate data via the indirect_ref LENGTH and byte_mask fields, instead of using the transfer out register. By not using a transfer register for the data modifier, the MEM unit does not have to pull any data from the FPC transfer out register, which in turn speeds up the execution of the no_pull atomic operations to the MEM unit.
sig_done[sig_name]	Signal the FPC when the reference completes using the specified signal (sig_name) and continues executing. Specify either ctx_swap or sig_done, not both.
sig_remote	Signal the remote FPC when the data is pushed (ReflectWrite) or pulled (ReflectRead); this will use the signal_ref from the initiating command.

#### 2.1.6.4.1 cpp\_commands source reference

##### 2.1.6.4.1.1 CPP Command Fields

The cpp\_command bus fields are shown below. The cpp\_command fields are described and over-ridden by the indirect\_ref opt\_tok explained below. Default sources for the cpp\_command fields are shown in the table below when indirect\_ref is not used in the instruction command.

*Refer to Databook “Distributed Switch Fabric-Command Push/Pull (DSF-CPP) Bus” for details on the CPP bus and CPP commands. Also explains how data\_master*

**Table 2.12. CPP Command - cpp\_command Source Fields**

Signal	Width	Description
target	4	CPP target for the command.

Signal	Width	Description
action	5	Action recognized by the CPP target indicating what should be performed.
token	2	Subtype of action recognized by the CPP target, indicating the variant of command.
length	5	Length of the command, dependent on the action/token, interpreted by the CPP target.  <i>Default: length[2:0] from instruction, and length[4:3]=2'b00.</i>
address	40	Address that the command should operate on, or other encoding, dependent on the action/token, interpreted by the CPP target. The address field is specified in src_op1 and src_op2 of the command instruction and does not need an override field.
byte_mask	8	Byte mask or further options for a command, dependent on the action/token, interpreted by the CPP target.  <i>Default: 8'b11111111.</i>
master_island	6	Island number of data master and signal master. Note that the data master and signal master must be in the same island.  <i>Default: Island number of the current FPC.</i>
data_master	4	Push/Pull ID within the island of data master to pull/push data from/to.  Pull ID[9:0] = {IslandID[5:0], MasterID[3:0]}, where MasterId is the data_master and valid values are listed in the Databook “Table 3.4 Pull IDs”.  <i>Default: Master ID of the current FPC.</i>
data_ref	14	Reference within the data master as to where to push/pull from; nominally this is byte-addressed, as it is increased by 8 for successive pushes.  <i>Default: The transfer register address from the instruction, data_ref[9:0] = global transfer register byte address for the current context, data_ref[13:10]=4'b0000.</i>
signal_master	4	These bits indicate which master within the master island should be signaled for the command.  <i>Default: Master number of the current FPC.</i>
signal_ref	7	Reference within the signal master as to which signal should be indicated with the commands pull or push. This consists of a 3-bit signal context followed by a 4-bit signal number.  <i>Default: From the instruction.</i>

## 2.1.6.4.2 Introduction to Indirect References

The I/O instructions support an indirect reference optional token. When the indirect\_ref optional token is specified, the output of the ALU from the previous instruction is used to modify one or more parameters within an instruction. There are two Indirect Reference modes: V1 mode and V2 mode. The V1 mode supports 13 formats of indirect data override with limitations. V2 mode has the most flexibility in overriding data fields with override bit per field. The format of the indirect data is specified in the instruction definition. All Reserved fields (noted as RES) must be set to 0 otherwise unpredictable behavior may result.

### 2.1.6.4.2.1 Introduction to V1 Indirect\_ref Format

The V1 indirect\_ref format (the indirect\_ref format used on NFP-32xx) encodes the upper bits (31..28) of the previous ALU value into 13 pre-defined override formats. Given the override format the ALU value lower 28 bits override the corresponding data fields. For format details refer to [Section 2.1.6.4.5](#)

The limitations of V1 Indirect\_ref format include:

- Since there are 13 fixed combinations of overrides, there is not always a format available to override (all and only) the required fields. When one of the 13 fixed formats is available which contains all the required fields and some others, then it can be used at the expense of having to override the extra fields with the values already encoded in the command.
- Some of the formats contain 8-bit data reference fields which are too small to override the transfer register byte addresses.
- Commands that use transfer registers and signals usually require the same context or FPC to be pushed/pulled and signalled. In these common cases both data master and signal master must be overridden which consumes more FPC cycles.

### 2.1.6.4.2.2 Introduction to V2 Indirect\_ref Format

The V2 indirect\_ref format (the indirect\_ref format first introduced with the NFP-6000) is intended to replace all pre-V2 indirect\_ref formats. The new indirect\_ref format reduces FPC cycle overhead while providing the programmer more flexibility and usability.

The previous ALU result and one Indirect Reference CSR is used to encode the override bits and fields. All the override bits and some of the most commonly used fields are specified by the previous ALU result. For format details refer to [Section 2.1.6.4.6](#)

### 2.1.6.4.2.3 Indirect Reference Notes

The following notes pertain to indirect references in general:

- User can't mix two indirect\_ref formats. Only one can be used since which format is used is controlled by CSR MiscControl bit 3 setting:
  - 0x0 Selects Normal Indirect Reference Mode. (V2 Indirect Reference mode)
  - 0x1 Selects Legacy Indirect Reference Mode. (V1 Indirect Reference mode)
- FPC Number, Context Number, and Transfer Register Number can all be overridden separately.

- There are two aspects of the transfer that are affected when the FPC Number is overridden: which FPC's transfer register(s) is the source/destination of data and which FPC is signaled.
- When the Transfer Register Number is overridden, it is treated as an absolute register number. If Context Number is also overridden, it determines where the signal is delivered, but not the Transfer Register Number.
- When the Transfer Register Number is not overridden and Context Number is, Context Number specifies where the signal is delivered as well as the Context Number of the transfer register.

### 2.1.6.4.3 Changing the Reference Count Using Indirect References

There are two methods to specify ref\_cnt in an instruction. One method is if the ref\_cnt is known to be static and is less than or equal to 8. Here opt\_tok indirect\_ref is not required. The instruction might look like:

```
mem[write, $xfer, src_op1, src_op2, ref_cnt] // always a ref_cnt transfer
where ref_cnt legal values are 1..8, where 1..8 is encoded to three bits as 0..7.
```

In the second method when ref\_cnt is changed during run-time operation and/or is greater than 8. In this case the opt\_tok indirect\_ref is required that will override the ref\_cnt for the instruction executed. Thus max\_nn keyword must be specified in the instruction where nn is a decimal constant between 1 and 32. The max\_nn keyword is only valid if the indirect\_ref optional token is specified. It allows the programmer to manually indicate to the assembler the maximum number of 32-bit/64-bit words that will be specified indirectly by the instruction. The override LENGTH value must be ref\_cnt-1. So if 12 transfer registers are required, then override is 11. **Refer to the description of each command for the allowed maximum number for max\_nn"**

V1 Override Length field (ref\_cnt) is accomplished using the following code:

```
// Select Length only Indirect Reference Format template
alu[tmp, --, B, 0x2, <<28]

// Set LENGTH field value in Indirect Reference Format template bits [3:0]
alu[tmp, tmp, OR, num_words]

// max_12 says can't exceed 12 xfer registers
mem[write, $xfer, src_op1, src_op2, max_12], indirect_ref, sig_done[s]
```

First 2 instructions setup LENGTH field using V1 mode (Override length only format) to provide runtime ref\_cnt value. Refer to [Section 2.1.6.4.5](#)

V2 Override Length field (ref\_cnt) is accomplished using the following code:

```
// Set OV_LEN and LENGTH fields in Previous ALU Result
alu[--, --, b, (((LEN-1)<<1) | 1), <<7]
mem[write, $xfer, src_op1, src_op2, max_12], indirect_ref, sig_done[2]
```

Refer to [Section 2.1.6.4.6](#)

Note that the override field LENGTH specified in Indirect Reference Formats for both V1 and V2 modes has the same meaning as ref\_cnt, i.e. specifying the number of transfer registers used with the instruction.

Note that some instructions may encode other data in the LENGTH field (not indicating number of transfer registers)

There is an issue concerning the assembler when redefining the ref\_cnt (reference count) field using an indirect reference: since the indirect reference data is specified at runtime, the assembler's register allocator has no way of knowing how many registers will be used for the I/O instruction. To handle this situation, the reference count field in the instruction parameter list can take values of the form:

max\_nn

where nn is a decimal constant between 1 and 32.

The max\_nn keyword is only valid if the indirect\_ref optional token is specified. It allows the programmer to manually indicate to the assembler the maximum number of 32-bit/64-bit words that will be specified indirectly.

Note that the register allocator needs to know the maximum number of registers used so it can allocate the correct number of registers. A program that specifies a maximum number and uses fewer registers will still work correctly. However, a program that specifies a maximum number and uses more registers will not work correctly and is considered a programming error.

Another issue involves the fact that the maximum reference count that can be encoded in the instruction is 8, whereas the maximum reference count that can be specified indirectly is 32. A count value must be specified in the instruction (stored in the control store) during assembly time regardless of whether the indirect\_ref optional token is specified; since the instruction only supports a three-bit count field, the assembler will use the lesser of nn and 8.

Also, it is possible to specify the indirect\_ref and not change the ref\_cnt value in the indirect reference data, however this is considered a programming error and will not be detected by the assembler.

Note that if the maximum count is less than or equal to 8, the programmer can still use max\_nn although there would be no difference if they used nn directly.

Additionally, the reference count field can take the value "max". This is equivalent to "max\_nn", except that the count is taken by counting how many registers follow the given register in .xfer\_order order.

The following are valid examples of this:

```
sram[read, $xfer, a1, a2, 6], indirect_ref
mem[read, $xfer, a1, a2, max_6], indirect_ref
mem[read, $xfer, a1, a2, max_12], indirect_ref
```

In these examples the reference counts encoded in the instruction are: 6, 6, 12.

The following are invalid examples:

```
sram[read, $xfer, a1, a2, 12], indirect_ref
```

```
mem[read, $xfer, a1, a1, max_12]
```

The problem with the first of these is that “12” is too large to be encoded in the instruction; the problem with the second is that “indirect\_ref” is not specified.

The REF CNT field in the indirect reference specifies the number of words to be transferred to or from a set of transfer registers, where words are 4 bytes or 8 bytes in length depending upon the target. Since the transfer registers are selected by specifying the starting transfer register and a reference count, it is possible to specify a REF CNT and the transfer register that results in a data transfer that extends beyond a thread’s relatively-addressed transfer register set. The programmer can ensure that a transfer register does not extend into the next thread by specifying a contiguous series of transfer registers using the .xfer\_order directive, specifying the first register in the series and ensuring that the REF\_CNT does not exceed the number of registers specified in the series.

#### 2.1.6.4.4 Indirect References to Another FPC

Through an indirect reference, one FPC can direct the I/O operation to another FPC. For example, FPC-0 can issue an mem/read operation and have the read results (and signal) go to FPC-1. To do this, the assembler must be informed as to the actual target FPC. Note that it is also possible to have multiple targets, in the case where the actual FPC is selected at run-time. To do this, add the optional token “ind\_targets[fpc1, fpc2, …]” to the I/O instruction; this is only valid if indirect\_ref is also specified.



##### Note

When using indirect\_ref token overwriting the FPC field, this optional token “ind\_targets[fpc1, fpc2, …]” must be added to direct the I/O operation to another FPC which has to be in the list of possible targets specified within the argument list of the ind\_targets optional token.

When the “ind\_targets” token is provided, it is assumed that the indirect reference is going to override the FPC. This causes two things to happen: any generated signal is considered to be remote and is looked up in the targets (if there are multiple targets, then it must have the same address in all specified FPCs). Secondly, the register specified in the I/O operation is assumed to be remote and is similarly looked up. Even if the transfer register address is being overridden, a valid remote register must be specified. If the transfer register address is not overridden, the transfer register used is for the same context as the issuing context. If the issuing FPC is running an odd context and the remote FPC is in 4-context mode, then the results are unpredictable. On the other hand, if the transfer register address is overridden, then the context can be explicitly specified.



##### Note

In order to direct the I/O operation to another FPC using the indirect\_ref and ind\_targets optional token, transfer registers address and signal number on the possible targets have to be the same in order for the I/O to be targeted correctly to the possible FPC targets.

When the ind\_targets token is provided, then there is no need to “complete” the I/O operation because the I/O operation is not referencing any of the local transfer registers or signals. As such, it is not allowed to use the ctx\_swap token with targets.

It is required that the number of contexts in the remote and local FPC are the same. If the number of contexts in the remote FPC is not the same as in the local FPC, the behavior is undefined.

The example code below shows how one could setup FPC-0 to issue a CTM memory read and have the read data pushed to FPC-1 read transfer registers using V2 Indirect Reference.

```

;Using V2 Indirect Reference format
;FPC-0:

.reg remote $rem_xfer[2]
.reg $xfer[2]
.sig remote rem_sig
.sig s1
.xfer_order $xfer
.reg addr
.reg tmp
.reg data1
.reg data2

.init addr 0x80000000 ; MU Direct Access Self Island CTM Memory
.init tmp (((__ISLAND & 0x3F) << 24) | 5 << 20) ; island and master (5 == FPC 1)

.if (ctx() == 0)
    ; Load a pattern for CTM memory write below.
    immed_w0[data1, 0x4321]
    immed_w1[data1, 0x8765]
    immed_w0[data2, 0xcba9]
    immed_w1[data2, 0xfed]
    alu[$xfer[0], --, B, data1]
    alu[$xfer[1], --, B, data2]
    mem[write, $xfer[0], addr, <<8, 0, 1], ctx_swap[s1]

    ; Read CTM memory containing the patern written above
    mem[read, $xfer[0], addr, <<8, 0, 1], ctx_swap[s1]

    ; The next few lines of code setup for mem[read] and push data to FPC-1.
    ; Write to CSR to override fields that don't directly fit in prev_alu.
    ; Override fields are the Island and Master number that was saved above in tmp.
    local_csr_wr[CMD_INDIRECT_REF_0, tmp]

    ; Load in prev_alu OVE_MASTER field a value of 0x2 to overload island,
    ; master data/signal fields. Here signal and master are the same value.
    alu[--, --, b, 4]

    ; Perform CTM memory read operation from FPC-0 and push data to Island-32 FPC-1
    ; ind_targets will direct I/O operation to Island-32 FPC-1
    mem[read, $rem_xfer[0], addr, <<8, 0, 1], ind_targets[__nfp_meid(32,1)], \
        sig_done[rem_sig], indirect_ref

.endif
ctx_arb[kill]

;FPC-1

.reg visible $rem_xfer[2]
.sig visible rem_sig

```

```
.if(ctx() == 0)
    ctx_arb[rem_sig]
    ; When FPC-1 is woken by FPC-0 signal "rem_sig" the following data is available:
    ; FPC-1 $rem_xfer[0] will contain value of 0x87654321
    ; FPC-1 $rem_xfer[1] will contain value of 0x0fedcba9
.endif

ctx_arb[kill]
```

The example code below shows how one could setup FPC-0 to issue a CTM memory read and have the read data pushed to FPC-1 read transfer registers using V1 Legacy Indirect Reference. Note that this code is for legacy support and is limited to within an one island. It is recommended to use V2 Indirect Reference over the limited V1 Legacy Indirect Reference.

```
;Using V1 Indirect Reference format
;FPC-0
.reg remote $x
.sig remote sig
#define FPC 1

; The data in CTM memory is valid data from previous initialization.

; Override the master number. The Island ID default is 0, thus this
; V1 indirect reference will not work for other Islands. There
; is no support to specify an Island ID using V1 indirect reference.
; Override Data Master and Xfer register (NFP V! (Legacy Mode) Indirect Ref Format)
alu[--,--,b,(0x20 | FPC), <<26]

; Read CTM memory and signal remote FPC after data is pushed into
; FPC-1 read transfer register.
mem[read, $x, addr1, addr2, 1], sig_done[sig], ind_targets[FPC]



---


;FPC-1
.reg visible $x
.sig visible sig

ctx_arb[sig]
; $x contains CTM memory data value read from FPC-0 mem[read] operation
```

## 2.1.6.4.5 NFP-32xx (Legacy) Indirect Reference Formats

In the NFP-32xx Legacy Mode indirect formats, the previous ALU value is divided into two fields. In one case, the uppermost 3-bits of the previous ALU output are used as the "enumeration" field and the lower 29-bits of the previous ALU output are used as the "indirect override" field. In all other cases, the uppermost 4-bits of the previous ALU output are used as the "enumeration" field, and the lower 28-bits are used as the "indirect override" field. In either case, the indirect override field carries a variable number of overriding command parameters (such as byte\_mask, length, etc.). The "enumeration" field can specify up to 16 formats; each one being a fixed combination of parameters. Currently there are 13 pre-defined formats, each one with a particular combination of parameters. Taken as a set, the 13 formats cover a wide range of commonly overridden parameters. However, given the length of the ALU output port (32-bit wide), no one format can carry the entire set of command parameters, and it may be possible that none of the pre-defined formats fit some particular software needs. In order to solve these latter issues, the NFP-6xxx Indirect Reference Format is available that

covers all entire set of command parameters. Refer to section [Section 2.1.6.4.6](#) for the recommended NFP-6xxx Indirect Reference Formats details.

The tables below show the 13 pre-defined formats as available in the NFP-32xx Legacy Mode of the NFP-6xxx architecture.

There are limitations in using the NFP-32xx Legacy Mode Indirect Reference Formats as shown below. It is recommended to use the NFP-6xxx Indirect Reference Formats to make use of the entire resources on the device. The limitations are shown below:

Data master - Is the master receiving push or pull data. In the NFP-32xx indirect reference format an 8-bit Data master field contains two basic fields:

- Bits[3:0] - {1'b1, bit[2:0]} specifies the ME ID up to 8 MEs. The NFP-6xxx architecture supports up to 12 MEs.
- Bits[7:4] - Cluster ID, indicating the ME or Non-ME given the Cluster ID value. However in the NFP-6xxx architecture one cannot override the ISLAND ID using the NFP-32xx override format. In the NFP-6xxx architecture the data\_master is separated from the Island ID (data\_master = 4 bits and island ID is 6 bits). If one wants to override the Island ID one must use the NFP-6xxx override format.

**Table 2.13. NFP Legacy Mode Indirect Reference Formats**

Override all fields																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	x	Byte mask				signal CTX		signal and data master		Byte address of Xfer register or immediate data (see Note 1)								Length												

Override Xfer register only																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	Reserved				Xfer register or immediate data (see Note 1)												Reserved											

Override Xfer register and length																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	Reserved				Xfer register or immediate data (see Note 1)												Length											

Override length only																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	Reserved												Length						Length									

Override byte mask only																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	Reserved																				Byte mask							

Override ME and context																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	1	Signal master										CTX	Reserved																	

Override ME, context, and signal																																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	0	Signal master										CTX	Signal number	Reserved																	

Override ME and Xfer register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Data master										Xfer register or immediate data (see Note 1)										Reserved							

Override ME and Xfer register 2																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	1	Signal and data master										Xfer register or immediate data (see Note 1)										Reserved							

Override ME, Xfer register, and context																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	Data master										Xfer register or immediate data (see Note 1)										Reserved								CTX

Override ME, Xfer register, and context 2																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	1	Signal and data master										Xfer register or immediate data (see Note 1)										Reserved								CTX

Override ME, Xfer register, context, and length																																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	1	1	Signal and data master										Signal CTX	Reserved	Xfer register or immediate data (see Note 1)										Length							

**Note:**

1. The data reference value for an Xfer register must specify the byte address of the transfer register. Overrides with 8 bit fields will only allow specifying the transfer registers of the first two contexts.

### 2.1.6.4.6 NFP-6xxx (Normal) Indirect Reference Formats

The NFP-6xxx Indirect Reference mode reduces the FPC cycle overhead while maintaining ease of use and flexibility. The previous ALU result covers the most commonly used override fields and the remaining override fields are specified by the Indirect CSR.

The software requirements of the NFP-6xxx Indirect Reference mode include:

- Override any combination of fields, each with its own enable bit. While the NFP-32xx Indirect Reference mode overrides only a fixed combinations of fields for a total of 13 formats.
- Override only the fields that are required thus eliminating unnecessary FPC overhead with overridden fields with same values as the instruction should have.
- Optionally override data\_master and signal master from a single field to save FPC cycles. While in the NFP-32xx Indirect Reference mode both the data master and signal master must be overridden consuming more FPC cycles.

#### 2.1.6.4.6.1 Sources for Override Fields

The following sources are used for the override bits and fields:

- PREV\_ALU - The ALU result from the previous instruction is always used when the indirect\_ref options token is specified for a command.
- CMD\_INDIRECT\_REF\_0 - The Indirect CSR register is used for override fields that do not fit in PREV\_ALU. The Indirect CSR only needs to be written if fields in it are enabled by override bits in PREV\_ALU.

#### 2.1.6.4.6.2 Previous ALU Result (PREV\_ALU)

The PREV\_ALU is the ALU result value comprised of various override bits, DATA16 and LENGTH. The override bits are identified in PREV\_ALU table by OV\_\*. (see [Table 2.14](#) )

##### 2.1.6.4.6.2.1 Previous ALU Result

**Table 2.14. Previous ALU Result - PREV\_ALU**

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

DATA16

LENGTH

OV\_SIG\_NUM

OV\_SIG\_CTX

RESERVED

**Table 2.15. Previous ALU Result Description**

Bits	Width	Field name	Description
31:16	16	DATA16	This is primarily for the data reference, but can also override the byte mask when the data reference is not used. The <a href="#">OVE_DATA</a> field enumerates how DATA16 is used.
15	1	RESERVED	Clear to 0.
14	1	OV_SIG_CTX	If set, overrides the signal context: <ul style="list-style-type: none"> <li>• cpp_command.signal_ref[6:4] = <a href="#">CMD_INDIRECT_REF_0.SIGNAL_CTX[2:0]</a></li> </ul>
13	1	OV_SIG_NUM	If set, overrides the signal number: <ul style="list-style-type: none"> <li>• cpp_command.signal_ref[3:0] = <a href="#">CMD_INDIRECT_REF_0.SIGNAL_NUM[3:0]</a></li> </ul>
12:8	5	LENGTH	Used when <a href="#">OV_LEN</a> is set.
7	1	OV_LEN	If set, overrides the length: <ul style="list-style-type: none"> <li>• cpp_command.length[4:0] = <a href="#">PREV_ALU.LENGTH[4:0]</a></li> </ul>
6	1	OV_BM_CSR	If set, overrides the byte mask from <a href="#">CMD_INDIRECT_REF_0</a> : <ul style="list-style-type: none"> <li>• cpp_command.byte_mask[7:0] = <a href="#">CMD_INDIRECT_REF_0.BYTE_MASK[7:0]</a></li> </ul>
5:3	3	OVE_DATA	Controls how <a href="#">DATA16</a> is used for overrides: <ul style="list-style-type: none"> <li>0x0 - <a href="#">DATA16</a> is unused.</li> <li>0x1 - Overrides the full data reference:               <ul style="list-style-type: none"> <li>• cpp_command.data_ref[13:0] = <a href="#">PREV_ALU.DATA16[13:0]</a></li> </ul> </li> <li>0x2 - Overrides 16-bit immediate data:               <ul style="list-style-type: none"> <li>• cpp_command.data_master[3:0] = {2'b00, <a href="#">PREV_ALU.DATA16[15:14]</a> }</li> <li>• cpp_command.data_ref[13:0] = <a href="#">PREV_ALU.DATA16[13:0]</a></li> </ul> </li> <li>0x3 - Overrides only the data context:               <ul style="list-style-type: none"> <li>• cpp_command.data_ref[9:7] = <a href="#">ALU_PREV.DATA16[9:7]</a></li> </ul> </li> <li>0x4 - Overrides both the data and signal contexts from <a href="#">DATA16</a> :               <ul style="list-style-type: none"> <li>• cpp_command.data_ref[9:7] = <a href="#">ALU_PREV.DATA16[9:7]</a></li> <li>• cpp_command.signal_ref[6:4] = <a href="#">ALU_PREV.DATA16[9:7]</a></li> </ul> </li> <li>0x5 - Overrides only the per-context offset of the data reference:               <ul style="list-style-type: none"> <li>• cpp_command.data_ref[6:0] = <a href="#">ALU_PREV.DATA16[6:0]</a></li> </ul> </li> <li>0x6 - Overrides the byte mask from <a href="#">DATA16</a> :</li> </ul>

Bits	Width	Field name	Description
			<ul style="list-style-type: none"> <li>• cpp_command.byte_mask[7:0] = <a href="#">PREV_ALU.DATA16[7:0]</a></li> </ul> <p>0x7 - <a href="#">DATA16</a> is unused.</p> <p>Note that the signal context can be overridden separately using <a href="#">OV_SIG_CTX</a> .</p>
2:1	2	OVE_MASTER	<p>Controls how <a href="#">CMD INDIRECT REF_0.ISLAND[5:0]</a> and <a href="#">CMD INDIRECT REF_0.MASTER[3:0]</a> are used for overrides:</p> <p>0x0 - <a href="#">CMD INDIRECT REF_0.ISLAND[5:0]</a> and <a href="#">CMD INDIRECT REF_0.MASTER[3:0]</a> are unused.</p> <p>0x1 - Overrides the island and the data master:</p> <ul style="list-style-type: none"> <li>• cpp_command.master_island[5:0] = <a href="#">CMD INDIRECT REF_0.ISLAND[5:0]</a></li> <li>• cpp_command.data_master[3:0] = <a href="#">CMD INDIRECT REF_0.MASTER[3:0]</a></li> </ul> <p>0x2 - Overrides the island, data master and signal master:</p> <ul style="list-style-type: none"> <li>• cpp_command.master_island[5:0] = <a href="#">CMD INDIRECT REF_0.ISLAND[5:0]</a></li> <li>• cpp_command.data_master[3:0] = <a href="#">CMD INDIRECT REF_0.MASTER[3:0]</a></li> <li>• cpp_command.signal_master[3:0] = <a href="#">CMD INDIRECT REF_0.MASTER[3:0]</a></li> </ul> <p>0x3 - Overrides the island:</p> <ul style="list-style-type: none"> <li>• cpp_command.master_island[5:0] = <a href="#">CMD INDIRECT REF_0.ISLAND[5:0]</a></li> </ul> <p>Note that the signal master can be overridden separately using <a href="#">OV_SM</a> .</p>
0	1	OV_SM	<p>If set, overrides the signal master:</p> <ul style="list-style-type: none"> <li>• cpp_command.signal_master[3:0] = <a href="#">CMD INDIRECT REF_0.SIGNAL_MASTER[3:0]</a></li> </ul>

#### [2.1.6.4.6.3 Indirect CSR \(CMD\\_INDIRECT\\_REF\\_0\)](#)

The CMD\_INDIRECT\_REF\_0 is used if override fields do not fit in PREV\_ALU. The override fields are shown in Indirect CSR - CMD\_INDIRECT\_REF\_0 register table in [Section 2.1.6.4.6.3.1](#)

**Refer to NFP-6xxx or NFP-4xxx Databook "NFP-6xxx Pull IDs" or "NFP-4xxx Pull IDs" table for list of Island and Master IDs for all Masters in the NFP-6xxx of NFP-4xxx device.**

#### 2.1.6.4.6.3.1 Indirect CSR

**Table 2.16. Indirect CSR - CMD\_INDIRECT\_REF\_0**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED	ISLAND								MASTER	SIGNAL_MASTER				SIGNAL_CTX	SIGNAL_NUM				RESERVED	BYTE_MASK											

**Table 2.17. Indirect CSR Description**

Bits	Width	Field name	Description
31:30	2	RESERVED	Reserved
29:24	6	ISLAND	The <a href="#">OVE_MASTER</a> field controls how ISLAND is used.
23:20	4	MASTER	The <a href="#">OVE_MASTER</a> field controls how MASTER is used.
19:16	4	SIGNAL_MASTER	Used when <a href="#">OV_SM</a> is set.
15:13	3	SIGNAL_CTX	Used when <a href="#">OV_SIG_CTX</a> is set.
12:9	4	SIGNAL_NUM	Used when <a href="#">OV_SIG_NUM</a> is set.
8	1	RESERVED	Reserved
7:0	8	BYTE_MASK	Used when <a href="#">OV_BM_CSR</a> is set.

#### 2.1.6.5 Event Signals

The I/O instructions provide the option of having the FPC signaled when data is pushed or pulled from the transfer register. The I/O instruction/commands can be classified into the following types based on the number of signals that can be requested:

- Single signal provided when data is pulled from the write transfer register
- Single signal provided when data is pushed to the read transfer register
- Two signals provided: one for each of the above cases
- Two signals provided: one to flag completion of a data transfer and one to indicate an error condition
- No signal provided because data is neither pulled nor pushed.

Event Signals can be set in nine different ways:

1. When data is written into Transfer\_In Registers (part of A\_Push\_ID input)
2. When data is taken from Transfer\_Out Registers (part of A\_Pull\_ID input)
3. When data is taken from Transfer\_Out Registers (part of B\_Pull\_ID input)
4. On InterThread\_Sig\_In input

5. On NN\_Sig\_In input
6. On Prev\_Sig\_In input
7. On write to Same\_ME\_Signal Local CSR
8. By Internal Timer

Any or all Event Signals can be set by any of the above sources. The external inputs are the first 7 in the list above. (The “InterThread\_Sig\_In” inputs to the FPC can be connected in such a way that any FPC doing an external command to write a specific register address will assert those inputs. This mechanism can be used for a configuration where many FPCs, or other processors, can signal each other.)

When a Context goes to Sleep state (executes a ctx\_arb instruction, or a Command instruction with ctx\_swap token), it specifies which Event Signal(s) it requires to be put in Ready state. Ctx\_arb instruction also specifies if the logical AND or logical OR of the Event Signal(s) is needed to put the Context into Ready state.

When a Context Event Signals arrive, it goes to Ready state, and then to Executing state. In the case where the Event Signal is linked to moving data into or out of Transfer registers (numbers 1 through 4 in the list above), the code can safely use the Transfer register as the first instruction (for example, using a Transfer\_In register as a source operand will get the new read data). The same is true when the Event Signal is tested for branches (br\_signal or br\_!signal instructions).

Signals are generated and tested using the instructions listed in [Table 2.18](#).

**Table 2.18. Instructions and Optional Tokens that Use Signals**

Option		Description
Instruction	Optional Token	
Any I/O Instruction	sig_done[sig_name]	<ol style="list-style-type: none"> <li>1. Request Signal.</li> <li>2. Thread continues to execute.</li> </ol>
	ctx_swap[sig_name]	<ol style="list-style-type: none"> <li>1. Request Signal.</li> <li>2. Put thread to sleep.</li> <li>3. Wake when signaled.</li> <li>4. Clear signal.</li> </ol> <p>Note: ctx_swap is not used with instructions that require two signals.</p>
br_signal br_!signal		<ol style="list-style-type: none"> <li>1. Test Signal.</li> <li>2. Clear signal if set.</li> <li>3. Branch appropriately. See Note 1.</li> </ol>
ctx_arb		<ol style="list-style-type: none"> <li>1. Put thread to sleep.</li> <li>2. Wake when signaled.</li> <li>3. If ctx_arb[], all, then the signal(s) is(are) cleared. The ctx_arb instruction supports waking on a list of signals. See Note 1.</li> </ol>

**Notes:**

- These instructions use signals requested by an I/O instruction that specify the sig\_done[sig\_name] optional token.

The assembler supports a register allocator that manages the allocation of signals for the programmer. The programmer uses symbolic names for signals rather than explicit numbers and the assembler assigns one or more signal numbers from a pool of fifteen signal (1-15). The hardware requires that I/O instructions that require two signals be assigned consecutive signals and that the first signal is an even number. The assembler hides this restriction from the programmer except for the special case of the br\_signal and br\_!signal instructions. These instructions can only test one signal at a time so the programmer must provide two branch instructions to test both signals. Use the syntax signal\_name[0] and signal\_name[1] to access the first and the second signal, respectively. [Table 2.19](#) lists the number of signals generated for each instruction and instruction command.

This document uses the following terminology in the instruction set definitions (see [Section 2.2](#)) to refer to signals names that are mapped to two signals:

- sig\_name: One signal is provided
- sig\_name2: Two signals are provided.

For example ctx\_swap[sig\_name] or ctx\_swap[sig\_name2].

**Table 2.19. Signal Restrictions for Each I/O Instruction [command]**

Instruction	Command	Valid Signal References	Comments
CLS	(various)	2 to 14, pair beginning with even signal numbers, or 1 to 15 for individual signals, depending on the command	Refer to the <i>Netronome NFP-6xxx Databook</i> to obtain details with respect to signal usage of each command.
Crypto	read, write, write_fifo	1 to 15	
CT	read, write, write_fifo	1 to 15	
	reflect_read_sig_both, reflect_read_sig_init, reflect_read_sig_remote, reflect_write_sig_both, reflect_write_sig_init, reflect_write_sig_remote	1 to 15	
	ring_get, ring_put	1 to 15	
	xpb_read, xpb_write	1 to 15	
	read, read_int, write, write_int	1 to 15	
ILA	read_check_error, write_check_error	2 to 14, pair beginning with even signal numbers	Refer to the <i>Netronome NFP-6xxx Databook</i> to obtain details with respect to signal usage of each command.
	packet_ready_drop, packet_ready_unicast,	1 to 15	

<b>Instruction</b>	<b>Command</b>	<b>Valid Signal References</b>	<b>Comments</b>
	packet_ready_multicast_dont_free, packet_ready_multicast_free_on_last, read, write		
MEM (atomic / queue)	(various)	2 to 14, pair beginning with even signal numbers, or 1 to 15 for individual signals, depending on the command	Refer to the <i>Netronome NFP-6xxx Databook</i> to obtain details with respect to signal usage of each command.
MEM (bulk)	read, write	1 to 15	
PCIe	read, read_int, read_pci, read_rid, write, write_int, write_pci, write_rid	1 to 15	
SRAM	read, write	1 to 15	The “name” signal will be even and it indicates that data has been pulled from the transfer register, while “name” + 1 signal indicates that the swapped or premodified data has been returned to the transfer register.
	test_and_incr, test_and_decr set, clr, add	1 to 15	
	Regular version: test_and_set, test_and_clr, test_and_add, swap	2 to 14, pair beginning with even signal numbers	
	No-pull version: test_and_set, test_and_clr, test_and_add, swap	1 to 15	
	incr, decr	Not Used	
	csr_rd, csr_wr	1 to 15	
	rd_qdesc_head, rd_qdesc_tail	1 to 15	
	wr_qdesc, wr_qdesc_count	Not Used	
	enqueue, enqueue_tail	Not Used	
	dequeue	1 to 15	
	get	1 to 15	
	put	2 to 14, pair beginning with even signal numbers	
	journal	1 to 15	
	fast_journal	Not Used	

## 2.1.7 Coding Restrictions

This section lists coding restrictions.

### 2.1.7.1 Branch or I/O Command in Defer Slot

A branch, jump, context swap or any I/O command cannot be placed in a defer slot.

#### Example

```
br[A#], defer[1]
br[B#] ; illegal
```

#### Example

```
ctx_arb[sig_name], defer[1]
jump[offset, A#]; illegal
```

#### Example

```
alu[i, i, +, 1]
BEQ[A#], defer[2]
alu[i, i, +, 1]
BLT[B#] ; illegal
```

#### Example

```
jump[offset, A#], defer[3]
alu[i, i, +, 1]
alu[i, i, +, 1]
ctx_arb[1] ; illegal
```

### 2.1.7.2 Condition Codes after Swap

Condition Codes are not stored during a context swap.

#### Example

```
ctx_arb[voluntary]
BEQ[A#] ; illegal
```

#### Example

```
ctx_arb[1]
```

```
|BEQ[A#] ; illegal
```

**Example**

```
sram[write, $data, base, offset, 1], ctx_swap[sig_1]  
BEQ[A#] ; illegal
```

**Example**

```
sram[write, $data, base, offset, 1], ctx_swap[sig_1]  
br=byte[i,1,0x0, A#]; legal
```

### 2.1.7.3 CAM after Conditional P3 Branch

An operation that modifies any state of the CAM cannot immediately follow a conditional P3 branch. P3 branches include:

- BR=BYTE,
- BR!=BYTE,
- BR\_BCLR,
- BR\_BSET,
- BR\_SIGNAL,
- BR\_!SIGNAL,
- BR\_INP\_STATE,
- BR\_!INP\_STATE,
- BR\_INP\_LSTATE,
- BR\_!INP\_LSTATE.

CAM state includes Tags, State bits, and LRU/MRU logic. (Note that a CAM can be placed in the defer slot of a Conditional P3 branch.)

**Example**

```
br=byte[i,1,0x0, A#]; p3 branch  
CAM_Write_State[entry , 0x2]; illegal
```

**Example**

```
br=byte[i,1,0x0, A#]; p3 branch  
alu[i,i,+1]  
CAM_Write_State[entry , 0x2]; legal
```

**Example**

```
br=byte[i,1,0x0, A#], defer[1]; p3 branch
```

```
CAM_Write_State[entry , 0x2]; legal
```

**Example**

```
br_signal [1, A#] ; p3 branch  
Lookup_CAM[status, lookup_value]; illegal
```

**Example**

```
jump [offset, A#] ; p3 jump  
Lookup_CAM[status, lookup_value]; legal
```

## 2.1.7.4 Signal Pair with Swap

A command that returns a signal pair cannot use ctx\_swap token.

**Example**

```
mem[test_add, $val, base, offset, 1], ctx_swap[test_sig]; illegal (double signal)
```

**Example**

```
mem[test_add, $val, base, offset, 1], sig_done[test_sig]; legal (no ctx_swap)  
ctx_arb[test_sig]
```

**Example**

```
cls[read, $val, base, offset, 1], ctx_swap[read_sig]; legal (single signal)
```

## 2.1.7.5 BCC after Conditional P3 Branch

A BCC instruction cannot be executed immediately after a branch conditional instruction that makes a branch decision in the FPC P3 pipe stage.

**Example**

```
br=byte[i, 1, 0x0, A#]; p3 branch  
BNE[A#] ; illegal
```

**Example**

```
br_bset[reg, bit, A#]; p3 branch  
BLT[A#] ; illegal
```

**Example**

```
br!signal[1, A#]; p3 branch  
BEQ[A#] ; illegal
```

### Example

```
br_inp_state[state, A#]; p3 branch  
BGT[A#] ; illegal
```

### Example

```
jump[offset, A#]; p3 branch  
BLT[A#] ; legal
```

### Example

```
br=byte[i, 0, 0x0, A#]; p3 branch  
br=byte[i, 1, 0x0, B#]; legal  
br=byte[i, 2, 0x0, C#]; legal  
br=byte[i, 3, 0x0, D#]; legal
```

### Example

```
br=byte[i, 1, 0x0, A#]; p3 branch  
alu[i, i, +, 1]  
BEQ[A#] ; legal
```

### Example

```
br=byte[i, 1, 0x0, A#],defer[1]; p3 branch  
alu[i, i, +, 1]  
BEQ[A#] ; legal
```

### Example

```
alu[i, i, +, 1]  
br=byte[i, 1, 0x0, A#]; p3 branch  
nop  
BEQ[A#] ; legal
```

## 2.1.7.6 LOCAL\_CSR\_RD Cannot Be in Last Defer Slot

A LOCAL\_CSR\_RD cannot be placed in the last defer slot of a branch, ctx\_arb, ctx\_swap, jump, or rtn.

### Example

```
ctx_arb[1], defer[1]
```

```
|local_csr_rd[ACTIVE_LM_ADDR_0]; illegal|
```

**Example**

```
|ctx_arb[1], defer[2]  
local_csr_rd[ACTIVE_LM_ADDR_0]; legal  
immed[temp, 0]|
```

**Example**

```
|br[label#] defer[2]  
alu[i, i, +, 1]  
local_csr_rd[NN_GET]; illegal|
```

**Example**

```
|rtn[address], defer[3]  
alu[i, i, +, 1]  
local_csr_rd[Active_CTX_sts]; legal  
immed[temp, 0]|
```

**Example**

```
|rtn[address], defer[3]  
alu[i, i, +, 1]  
nop  
local_csr_rd[Active_CTX_sts]; illegal|
```

## 2.1.7.7 LOCAL\_CSR\_WR to ACTIVE\_LM\_ADDR, or CAM\_LOOKUP

LOCAL\_CSR\_WR to ACTIVE\_LM\_ADDR, or CAM\_LOOKUP (with lm\_addr#[num] token) cannot be placed in the defer slots of a CTX\_ARB instruction (or command instruction with ctx\_swap token).

**Example**

```
|ctx_arb[1], defer[1]  
local_csr_wr[ACTIVE_LM_ADDR_0, value]; illegal|
```

**Example**

```
|ctx_arb[1]  
local_csr_wr[ACTIVE_LM_ADDR_0, value]; legal|
```

**Example**

```
|ctx_arb[1], defer[1]|
```

```
local_csr_wr[ACTIVE_LM_ADDR_1_BYTE_INDEX, value]; illegal
```

**Example**

```
ctx_arb[1], defer[1]
local_csr_wr[INDIRECT_LM_ADDR_1, value]; legal because not ACTIVE_LM_ADDR
```

**Example**

```
ctx_arb[1], defer[2]
alu[i, i, +1]
local_csr_wr[ACTIVE_LM_ADDR_0, value]; illegal
```

**Example**

```
ctx_arb[1], defer[1]
cam_lookup[@gpr_a,@gpr_b], lm_addr0[value] ; illegal
```

## 2.1.7.8 LOCAL\_CSR\_RD Must Be Followed by an IMMED Op

LOCAL\_CSR\_RD instructions must be followed by an IMMED instruction.

**Example**

```
local_csr_rd[NN_get]
local_csr_wr[NN_put, 0x2]; illegal
```

**Example**

```
local_csr_rd[ACTIVE_LM_ADDR_0]
local_csr_rd[NN_PUT]; illegal
```

**Example**

```
local_csr_rd[ACTIVE_CTX_STS]
alu[i, i, +1]; illegal
```

**Example**

```
local_csr_rd[ACTIVE_CTX_STS]
immed[temp, 0]; legal
```

## 2.1.7.9 I/O Command Op after LOCAL\_CSR\_WR

If an I/O instruction is issued within the CSR Usage or Read Latency of a local\_csr\_write, the CSR should not be used or read by the I/O instruction. Instructions prior to the I/O instruction may use or read the CSR.

However, instructions after the I/O instruction may not use or read the CSR until the Usage or Read Latency rules have been met

### Example

```
local_csr_wr[ active_lm_addr_1, reg_a18]
mem[write , $data[0], *l$index1, offset, ref_cnt], sig_done[sig_2]; illegal
```

### Example

```
local_csr_wr[ active_lm_addr_0, reg_a18]
immed_w1[ reg_a31, 0 ]
sram[read , $sxfer15, *l$index0, offset, ref_cnt], ctx_swap[sig_2]; illegal
```

### Example

```
local_csr_wr[ active_lm_addr_0, reg_a18]
alu [ reg_a1, --, B, reg_a2 ]
immed_w0[ reg_a2, 0 ]
sram[read , $sxfer15, *l$index0, offset, ref_cnt], sig_done[sig_2]; illegal
```

### Example

```
local_csr_wr[ t_index, reg_a18]
immed [ reg_a1, 0 ]
sram[read , $sxfer15, reg_a1, offset, ref_cnt], sig_done[sig_2]
alu [*$index++, --, B, reg_a10]; illegal
```

### Example

```
local_csr_wr[ t_index, reg_a18]
immed [ reg_a1, 0 ]
sram[read , $sxfer15, *$index, offset, ref_cnt], sig_done[sig_2]; illegal
```

### Example

```
local_csr_wr[ t_index, reg_a18]
immed [ reg_a1, 0 ]
sram[read , $sxfer15, *l$index0, offset, ref_cnt], sig_done[sig_2]
local_csr_rd [t_index]; illegal
immed [tmp, --]
```

### Example

```
local_csr_wr[ active_lm_addr_0, reg_a18]
nop;
nop;
```

```
nop;  
sram[read , $sxfer15, *l$index0, offset, ref_cnt], sig_done[sig_2]; legal
```

### Example

```
local_csr_wr[ active_lm_addr_0, reg_a18]  
alu [ reg_a1, *l$index0, + reg_b1 ] ; legal  
immed_w0[ *l$index0, 0 ] ; legal  
immed_w1[ *l$index0, 1 ] ; legal  
sram[read , $sxfer15, *l$index0, offset, ref_cnt], sig_done[sig_2]; legal
```

### Example

```
local_csr_wr[ t_index, reg_a18]  
immed [ reg_a1, 0 ]  
sram[read , $sxfer15, reg_a1, offset, ref_cnt], sig_done[sig_2]  
nop  
alu [*$index++, --, B, reg_a10]; legal
```

### Example

```
Local_csr_wr[ t_index, reg_a1]  
Immed [reg_a1,0]  
Alu[*$index++, --, B, reg_a11] ;legal  
Sram[read , $sxfer10, reg_a1, offset, ref_cnt], sig_done[sig_2]
```

## 2.1.7.10 LOCAL\_CSR\_WR to CTX\_WAKEUP\_EVENTS

LOCAL\_CSR\_WR to CTX\_WAKEUP\_EVENTS should be directly followed by a CTX\_ARB[--] or should be inside the defer shadow of a CTX\_ARB[--].

### Example

```
local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, value]  
ctx_arb[--]; legal
```

### Example

```
ctx_arb[--], defer[1]  
local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, value]; legal
```

### Example

```
local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, value]  
alu[temp, temp, +, 1]; unrelated instruction  
ctx_arb[--]; illegal
```

## 2.1.8 NFP-6xxx FPC Permitted Coding Sequences

The following coding sequences are permitted on the NFP-6xxx FPC. They are supplied for clarification, as they may have been illegal in the FPC instruction sets of earlier processors (e.g. Intel IXP processors).

### 2.1.8.1 Swap after P3 Branch

P3 branches (including jumps) can be followed by a context swap.

#### Example 1

```
br=byte[i, 1, 0x0, A#]; p3 branch  
ctx_arb[1] ; legal
```

#### Example 2

```
jump[offset, A#]; p3 branch  
ctx_arb[1] ; legal
```

#### Example 3

```
br!signal[1, A#]; p3 branch  
ctx_arb[1] ; legal
```

#### Example 4

```
br_bset[reg, bit, A#]; p3 branch  
ctx_arb[1] ; legal
```

## 2.1.8.2 Memory Command after P3 Branch

P3 branches (including jumps) can be followed by a command memory op.

#### Example

```
br_bset[reg, bit, A#]; p3 branch  
sram[write, $data, base, offset, 1], ctx_swap[sig_1]; legal
```

#### Example

```
jump[offset, A#] ; p3 branch  
cls[write, $data, base, offset, 2]; legal
```

## 2.1.8.3 Swap after Voluntary Swap

A context swap instruction can immediately follow a voluntary context swap.

### Example 1

```
ctx_arb[1]
ctx_arb[2] ; legal
ctx_arb[voluntary] ; legal
ctx_arb[3] ; legal
```

### Example 2

```
ctx_arb[1]
ctx_arb[2] ; legal
ctx_arb[voluntary] ; legal
alu[i,i,+,1]
ctx_arb[3] ; legal
```

### Example 3

```
ctx_arb[voluntary]
sram[write, $data, base, offset, 1], ctx_swap[sig_1]; legal
```

## 2.1.8.4 A LOCAL\_CSR\_WR in Defer Slot

A LOCAL\_CSR\_WR can be placed in the last defer slot of a branch, ctx\_arb, ctx\_swap, jump, or rtn.



### Note

There is a separate rule whereby it is illegal to write the ACTIVE\_LM\_ADDRs in the defer slot of a CTX\_ARB instruction (or command instruction with ctx\_swap token); see [Section 2.1.7.7](#). Also, some CSRs such as USTORE\_ERROR\_STATUS are read-only.

### Example 1

```
ctx_arb[1], defer[1]
local_csr_wr[NN_get, value]; legal
```

### Example 2

```
ctx_arb[1], defer[2]
Alu[i,i,+,1]
local_csr_wr[NN_put, 0x0]; legal
```

### Example 3

```
alu[i,i,+,1]
BEQ[label#], defer[1]
local_csr_wr[Same_ME_Signal, sig_value]; legal
```

#### Example 4

```
br[label#], defer[2]
alu[i,i,+,1]
local_csr_wr[Same_ME_Signal, sig_value]; legal
```

#### Example 5

```
rtn[address], defer[3]
alu[i,i,+,1]
immed[temp,2]
local_csr_wr[Active_ctx_future_count, fut_count]; legal
```

### 2.1.8.5 LOCAL\_CSR\_WR Can Be Followed by a LOCAL\_CSR\_RD or LOCAL\_CSR\_WR

#### Example 1

```
local_csr_wr[NN_get, value]
local_csr_wr[NN_put, 0x2]; legal
```

#### Example 2

```
local_csr_wr[Same_ME_Signal,sig_value]
local_csr_rd[NN_put] ; legal
immed[tmp,0]
```

## 2.2 Instruction Set

### 2.2.1 ALU

Perform an ALU operation on one or two source operands and deposit the result into the destination register. Update all ALU condition codes according to the result of the operation.

#### Instruction Format

```
alu[dest, A_op, alu_op, B_op], opt_tok
```

#### Parameter Descriptions

Parameter	Description
dest	Unrestricted destination that gets written with the result of the operation.

Parameter	Description	
A_op, B_op	Unrestricted source operand.	
alu_op	<b>ALU Operation</b>	<b>ALU operation Description</b>
	+	A operand + B operand.
	+16	A operand + (0xFFFF & B). B operand truncated to the least significant 16 bits (upper 2 bytes zeroed).
	+8	A operand + (0xFF & B). B operand truncated to the least significant 8 bits (upper 3 bytes zeroed).
	+carry	A operand + B operand + previous carry-in (carry-in equals previous carry-out).
	-carry	A operand - B operand - inverse of carry.
	-	A operand - B operand.
	B-A	B operand - A operand.
	B	B operand (A operand is ignored).
	~B	Inverted B operand (A operand is ignored).
	AND	A operand AND B operand (Bit-wise AND).
	~AND	Inverted A operand AND B operand (Bit-wise AND).
	AND~	Inverted B operand AND A operand (Bit-wise AND).
	OR	A operand OR B operand (Bit-wise OR)
	XOR	A operand XOR B operand (Bit-wise exclusive OR).
opt_tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a>	
	no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a>	
	predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>	

### Condition Codes Affected

ALU Operation	N	Z	V	C
+	Result[31] == 1	Result[31:0] == 0	Set if a signed overflow occurs. See Note 1.	Carry Out from Adder[31]
+16				
+8				
+carry				
-carry				
-				
B-A				
B				
~B				
			Cleared	Cleared

<b>ALU Operation</b>	<b>N</b>	<b>Z</b>	<b>V</b>	<b>C</b>
AND				
~AND				
AND~				
OR				
XOR				

**Note:**

1. Logically equivalent to Carry from Adder[31] XOR Carry from Adder[30].

## 2.2.2 ALU\_SHF

Perform an ALU operation on one or two operands and deposit the result into the destination register. The B operand is shifted or rotated prior to the ALU operation.

If the programmer uses the ALU command in the place of ALU\_SHF, the assembler will replace it with ALU\_SHF.

### Instruction Format

```
alu_shf[dest, A_op, alu_op, B_op, B_op_shf], opt Tok
```

### Parameter Descriptions

<b>Parameter</b>	<b>Description</b>	
dest	Restricted destination that gets written with the result of the operation.	
A_op, B_op	Restricted source operand.	
alu_op	<b>ALU Operation</b>	<b>ALU operation Description</b>
	B	B operand (A operand is ignored).
	~B	Inverted B operand (A operand is ignored).
	AND	A operand AND B operand (Bit-wise AND)
	~AND	Inverted A operand AND B operand (Bit-wise AND).
	AND~	Inverted B operand AND A operand (Bit-wise AND).
	OR	A operand OR B operand (Bit-wise OR).
B_op_shf	<b>Shift operation</b>	<b>Shift operation Description</b>
	<<n	Left shift n bits, where n = 1 through 31.
	<<indirect	Left shift by an amount specified in the lower 5 bits of the A operand of the previous instruction (the previous instruction must be one of the following ALU or ALU_SHF instructions – “AND”,

Parameter	Description								
	“AND~”, “XOR”, “OR”, “B” or “~B”). The lower 5 bits of the A operand should be n, where n is the desired left-shift amount. The A operand cannot be a constant. For cases where it should be a constant, a constant shift rather than an indirect shift should be used.								
	>>n	Right shift n bits, where n = 1 through 31.							
	>>indirect	Right shift by the amount specified in the lower 5 bits of the A operand of the previous instruction.							
	<<rotn	Left rotate n bits, where n = 1 through 31.							
	>>rotn	Right rotate n bits, where n = 1 through 31.							
	<b>Notes:</b>								
	<ol style="list-style-type: none"> <li>1. Indirect rotates are not allowed. However, an indirect rotate can be emulated by copying the desired register value to be rotated into a register on the opposite bank and then performing a dbl_shf instruction.</li> <li>2. There should be no spacing between the shift symbol (&lt;&lt;, &gt;&gt;) and the variable or keyword that follows it.</li> </ol>								
opt_tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a> no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a> predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>								

### Condition Codes Affected

ALU Operation	N	Z	V	C
B				
~B				
AND				
~AND				
AND~				
OR				

### 2.2.3 ASR

Perform an arithmetic shift right on a register. This is a two-instruction sequence. The first instruction must be a shift and/or logical instruction that puts a value in the MSb (bit 31) of the result (this instruction must be one of the following ALU or ALU\_SHF instructions – B, ~B, AND, ~AND, AND~, OR, XOR). The ASR instruction shifts a source register right, replicating that MSb into vacated bit positions.

## Instruction Format

```
asr[dest, src, shf_amt], opt_tok
```

## Parameter Descriptions

Parameter	Description	
dest	Restricted destination that gets written with the result of the operation.	
src	Restricted source operand.	
shf_amt	>>n	Shift the src operand right by n bits. Valid values of n are 1 to 31.
	>>indirect	Right shift by the amount specified in the lower 5 bits of the A operand of the previous instruction.
opt Tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a>	
	no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a>	
	predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>	

## Condition Codes Affected

N	Z	V	C
Result[31] == 1	Result[31:0] == 0	Cleared	Cleared

## Example

```
Example 1: Sign-extend the low byte of r0 which is 0x80. The result is 0xFFFF FF80.
immed_w0[r0,0x80]
immed_w1[r0,0x0]
alu_shf[r0, --, B, r0, <<24] ; bit 31 of result determines sign
asr[r0, r0, >>24]
```

## Example

```
Example 1: Sign-extend the low byte of r0 which is 0x80 using run-time shift value.
The result is 0xFFFF FF80.
immed_w0[r0,0x80]
immed_w1[r0,0x0]
alu[r1, --, B, 24] ; Set up the shift amount
;
alu[--, r1, OR, 1] ; a_op of this instr is r1 = 24
alu_shf[r0, --, B, r0, <<indirect] ; Bit 31 of result determines sign
; Result = 0x8000 0000
alu[--, r1, OR, r0] ; a_op of this instr = r1 = 24
asr[r0, r0, >>indirect]; shift >> 24 & sign extend 0x8000 0000
```

```
; result is 0xffff ff80
```

## 2.2.4 BCC (BRANCH CONDITION CODE)

Branch to an instruction at a specified label based on the condition codes set by a previous instruction. The condition codes supported are Sign (N), Zero (Z), Overflow (V), and Carry (C) out.

### Instruction Format

```
bcc[label#], opt_tok
```

**Table 2.20** describes the supported BCC instructions. The terms less, greater, and equal are used for comparison of signed numbers while higher, lower, and same are used for unsigned numbers. Subtracting two operands using the ALU instruction performs comparisons (for example A-B is equivalent to compare A to B). A Branch if Higher function can be performed by reversing the operands of the subtraction operation and using BLO and a Branch Lower or Same can be performed by reversing the operands of the subtraction operation and using BHS. Since there is more than one way to interpret a particular state of the Condition Codes, the assembler provides more than one mnemonic for some states.

**Table 2.20. Branch on Condition Code Instructions**

Mnemonic (BCC)	Description	Condition Code	Data Type
BEQ	Br if equal	Z == 1	Both
BNE	Br if not equal	Z == 0	Both
BMI	Br sign set (minus)	N == 1	Signed
BPL	Br sign clear (plus)	N == 0	Signed
BCS	Br if carry set	C == 1	Both
BHS	Br if higher or BCC same		Unsigned
BCC	Br if carry clear	C == 0	Both
BLO	Br if lower		Unsigned
BVS	Br if overflow set	V == 1	Signed
BVC	Br if overflow clear	V == 0	Signed
BGE	Br if greater or equal	N == V (XNOR)	Signed
BLT	Br if less than	N != V (XOR)	Signed
BGT	Br if greater than	(Z == 0) AND (N == V)	Signed
BLE	Br if less than or equal	(Z == 1) OR (N != V)	Signed

### Parameter Descriptions

Parameter	Description
label#	Symbolic label corresponding to the address of an instruction.

Parameter	Description
opt_tok	defer[n] (n= 1 to 3); refer to <a href="#">Section 2.1.5</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.5 BR

Branch unconditionally.

#### Instruction Format

br[label#], opt_tok
---------------------

#### Parameter Descriptions

Parameter	Description
label#	Symbolic label corresponding to the address of an instruction.
opt_tok	defer[n] (n= 1 to 2); refer to <a href="#">Section 2.1.5</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.6 BR\_BCLR, BR\_BSET

Branch to the instruction at the specified label when the specified bit of the register is clear or set.

#### Instruction Formats

br_bclr[reg, bit_position, label#], opt_tok
br_bset[reg, bit_position, label#], opt_tok

#### Parameter Descriptions

Parameter	Description
reg	Restricted source operand.

Parameter	Description
bit_position	A number specifying a bit position in a 32-bit word. Bit 0 is the least significant bit. Valid bit_position values are 0 through 31.
label#	Symbolic label corresponding to the address of an instruction.
opt_tok	defer[n] (n= 1 to 3); refer to <a href="#">Section 2.1.5</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.7 BR=BYTE, BR!=BYTE

Branch to the instruction at the specified label if a specified byte in a 32-bit word matches or mismatches the byte\_compare\_value.

#### Instruction Formats

br=byte[reg, byte_no, byte_compare_value, label#], opt Tok
br !=byte[reg, byte_no, byte_compare_value, label#], opt Tok

#### Parameter Descriptions

Parameter	Description
reg	Restricted source operand.
byte_no	A number specifying a byte in the register to be compared with byte_compare_value. Valid byte_no values are 0 through 3. A value of 0 refers to the byte in bit position [7:0].
byte_compare_value	Value used for comparison. Valid byte_compare_values are 0 to 255.
label #	Symbolic label corresponding to the address of an instruction.
opt Tok	defer[n] (n= 1 to 3); refer to <a href="#">Section 2.1.5</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.8 BR\_CLS\_STATE, BR\_!CLS\_STATE

Branch if the state of the specified cluster local scratch state name is equal to 1 (BR\_CLS\_STATE) or equal to 0 (BR\_!CLS\_STATE).

## Instruction Formats

br_cls_state[state_name, label#], opt_tok
br_!cls_state[state_name, label#], opt_tok

## Parameter Descriptions

Parameter	State Number	State Name	Description	
state_name	0	cls_ring0_status	Indicates that cluster local scratch ring is full.	
	1	cls_ring1_status		
	2	cls_ring2_status		
	3	cls_ring3_status		
	4	cls_ring4_status		
	5	cls_ring5_status		
	6	cls_ring6_status		
	7	cls_ring7_status		
	8	cls_ring8_status		
	9	cls_ring9_status		
	10	cls_ring10_status		
	11	cls_ring11_status		
	12	cls_ring12_status		
	13	cls_ring13_status		
	14	cls_ring14_status		
	15	cls_ring15_status		
label#	A symbolic label corresponding to the address of an instruction.			
opt_tok	defer [1] : Execute the instruction following this instruction before performing the branch operation.			
	defer [2] : Execute the two instructions following this instruction before performing the branch operation.			
	defer [3] : Execute the three instructions following this instruction before performing the branch operation.			

## 2.2.9 BR=CTX, BR!=CTX

Branch to the instruction at the specified label based on whether the current context is the specified context number.

## Instruction Formats

br=ctx[ctx, label#], opt_tok
------------------------------

br!=ctx[ctx, label#], opt_tok
-------------------------------

### Parameter Descriptions

Parameter	Description
label#	Symbolic label corresponding to the address of an instruction.
ctx	Context number. Valid ctx values are: <ul style="list-style-type: none"> <li>• 4 context mode: 0, 2, 4, 6</li> <li>• 8 context mode: 0 through 7.</li> </ul>
opt Tok	defer[n] (n= 1 to 2); refer to <a href="#">Section 2.1.5</a> .

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.10 BR\_INP\_STATE, BR\_!INP\_STATE

Branch if the state of the specified state name is equal to 1 (BR\_INP\_STATE) or equal to 0 (BR\_!INP\_STATE). A state is information generated external to the FPC and sent as an input to the FPC.

### Instruction Formats

br_inp_state[state_name, label#], opt Tok
br_!inp_state[state_name, label#], opt Tok

### Parameter Descriptions

Parameter	State Number	State Name	Description
state_name	0	NN_Empty	No valid data in NN_Ring from previous neighbor FPC.
	1	NN_FULL	NN_Ring in the neighbor FPC is full.
	2	CTM_Ring0_Status	Indicates that the CTM Ring is full or empty, based on the setting in Ring_#_Base[StatusType] register.
	3	CTM_Ring1_Status	
	4	CTM_Ring2_Status	
	5	CTM_Ring3_Status	
	6	CTM_Ring4_Status	
	7	CTM_Ring5_Status	
	8	CTM_Ring6_Status	

Parameter	State Number	State Name	Description
	9	CTM_Ring7_Status	
	10	CTM_Ring8_Status	
	11	CTM_Ring9_Status	
	12	CTM_Ring10_Status	
	13	CTM_Ring11_Status	
	14	CTM_Ring12_Status	
	15	CTM_Ring13_Status	
label#	A symbolic label corresponding to the address of an instruction.		
opt Tok	defer[n] (n= 1 to 3); refer to <a href="#">Section 2.1.5</a> .		

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.11 BR\_SIGNAL, BR\_!SIGNAL

Branch if the specified signal (in the CTX\_SIG\_EVENTS register) is asserted or deasserted. If the signal is asserted, these instructions will clear it.

#### Instruction Formats

br_signal[signal_name, label#], opt Tok
br_!signal[signal_name, label#], opt Tok

#### Parameter Descriptions

Parameter	Description
label#	Symbolic label corresponding to the address of an instruction.
signal_name	Symbolic label that specifies the signal. The two signals of a doubled signal is specified as sig_name and sig_name with a “+1” suffix (sig_name+1).
opt Tok	defer[n] (n= 1 to 3); refer to <a href="#">Section 2.1.5</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.12 BYTE\_ALIGN\_BE, BYTE\_ALIGN\_LE

This instruction requires multiple instructions and is used to concatenate data in two registers and put any four bytes into the destination register. The byte offset is specified by the value in the BYTE\_INDEX local CSR. The BYTE\_ALIGN\_BE instruction supports big endian data and the BYTE\_ALIGN\_LE instruction supports little endian data.

One byte\_align instruction is required to load the first four bytes from the src operand into one of two special temporary registers (not visible to the programmer). The BYTE\_ALIGN\_BE instruction uses the prev\_B temporary register and the BYTE\_ALIGN\_LE uses the prev\_A temporary register. Prev\_A and Prev\_B are loaded only during BYTE\_ALIGN instructions.

Successive instructions use the src operand to specify the next four bytes, concatenate the eight bytes, and select the desired result that is then placed into the dest register. The src operand is then placed into the appropriate temporary register to be used by the next byte\_align instruction.

The number of cycles required to perform a byte alignment on n 4 byte words takes n+2 cycles. (which include the instruction that loads the BYTE\_INDEX local CSR and initialize the temp register).

### Instruction Formats

<code>byte_align_le[dest, src], opt_tok</code>
<code>byte_align_be[dest, src], opt_tok</code>

### Parameter Descriptions

Parameter	Description
dest	Restricted destination that gets written with the result of the operation.
src	Restricted source operand.
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a></p> <p>predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

### Condition Codes Affected

N	Z	V	C
Result[31]==1	Result[31:0] == 0	Cleared	Cleared

### 2.2.12.1 Big Endian Align – Byte Index = 2

The initial data is shown in [Table 2.21](#). The NOP instructions are required to cover the read to use latency.

**Table 2.21. Initial Register Contents**

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	0x00	0x01	0x02	0x03
1	0x04	0x05	0x06	0x07
2	0x08	0x09	0x0A	0x0B
3	0x0C	0x0D	0x0E	0x0F

Instructions	Prev B	A Operand	B Operand	Result (destx)
local_csr_wr[byte_index, 2]				
nop				
nop				
nop				
Byte_align_be[--, r0]	--	--	0x00 01 02 03	--
Byte_align_be[dest1, r1]	0x00 01 02 03	0x00 01 02 03	0x04 05 06 07	0x02 03 04 05
Byte_align_be[dest2, r2]	0x04 05 06 07	0x04 05 06 07	0x08 09 0A 0B	0x06 07 08 09
Byte_align_be[dest3, r3]	0x08 09 0A 0B	0x08 09 0A 0B	0x0C 0D 0E 0F	0x0A 0B 0C 0D

**Note:**

1. A Operand comes from Prev\_B register during byte\_align\_be instructions.

## 2.2.12.2 Little Endian Align – Byte Index = 2

The initial data is shown in [Table 2.22](#) . The NOP instructions are required to cover the read to use latency.

**Table 2.22. Initial Register Contents**

Register	Byte 3 [31:24]	Byte 2 [23:16]	Byte 1 [15:8]	Byte 0 [7:0]
0	0x03	0x02	0x01	0x00
1	0x07	0x06	0x05	0x04
2	0x0B	0x0A	0x09	0x08
3	0x0F	0x0E	0x0D	0x0C

Instruction	A Operand	B Operand	Prev A	Result (destx)
local_csr_wr[byte_index,2]				
nop				
nop				
nop				
Byte_align_le[--, r0]	0x03 02 01 00	--		--

Instruction	A Operand	B Operand	Prev A	Result (destx)
Byte_align_le[dest1, r1]	0x07 06 05 04	0x03 02 01 00	0x03 02 01 00	0x05 04 03 02
Byte_align_le[dest2, r2]	0x0B 0A 09 08	0x07 06 05 04	0x07 06 05 04	0x09 08 07 06
Byte_align_le[dest3, r3]	0x0F 0E 0D 0C	0x0B 0A 09 08	0x0B 0A 09 08	0x0D 0C 0B 0A

**Note:**

1. B Operand comes from Prev\_A register during byte\_align\_le instructions.

### 2.2.12.3 Big Endian Align Using Index Address Mode to Local Memory

Another mode of operation is to use the index local CSRs with post-increment to select the source and destination registers. The ACTIVE\_LM\_ADDR\_0 local CSR is used to index into local memory and the T\_INDEX\_BYTEx\_INDEX local CSR is used to index the transfer registers and specify the byte alignment. The data in \$xfer0 to \$xfer3 is shown in [Table 2.21](#). The NOP instructions are required to cover the read-to-use latency.

Instruction	Prev B	A Operand (Note 2)	B Operand	Result (LM)
alu[lm_index_byte_align, b_align, OR, lm_index0,<<2]	--	--	--	--
local_csr_wr [ACTIVE_LM_ADDR_0_ BYTE_INDEX, lm_index_byte_align]	--	--	--	--
local_csr_wr[T_INDEX,& \$xfer0]	--	--	--	--
nop				
nop				
byte_align_be[--, *\$index++]	--	--	0x00 01 02 03	--
byte_align_be[*L\$index0[0],* \$index++]	0x00 01 02 03	0x00 01 02 03	0x04 05 06 07	0x02 03 04 05
byte_align_be[*L\$index0[1],* \$index++]	0x04 05 06 07	0x04 05 06 07	0x08 09 0A 0B	0x06 07 08 09
byte_align_be[*L\$index0[2],* \$index++]	0x08 09 0A 0B	0x08 09 0A 0B	0x0C 0D 0E 0F	0xA 0B 0C 0D

**Notes:**

1. b\_align = byte align value (2 in this example) and \$xfer0 is starting register. Refer to T\_INDEX, ACTIVE\_LM\_ADDR\_0\_BYTEx\_INDEX local CSR description. The & is a preprocessor directive that returns the address of the register.
2. A Operand comes from Prev\_B register during byte\_align\_be instructions.

## 2.2.12.4 Big Endian Align Using Index Address Mode to GPR

A common function is to read a L2 and L3 header into transfer registers, determine the offset of the L3 header based on the L2 protocol field, and then move the L3 header from the transfer registers to GPRs properly aligned. This is accomplished using the T\_INDEX local CSR with post-increment. The T\_INDEX\_BYTE\_INDEX local CSR is used to write both the T\_INDEX and BYTE\_INDEX local CSRs in a single write operation.

Instruction	Prev B	A Operand	B Operand	Result
alu[reg_addr, b_align, OR, & \$xfer0,<<2]	--	--	--	--
local_csr_wr [T_INDEX_BYT_E_IND EX,reg_addr]	--	--	--	--
byte_align_be[--, *\$index++]	--	--	0x00 01 02 03	--
byte_align_be[gpr0,*\$index++]	0x00 01 02 03	0x00 01 02 03	0x04 05 06 07	0x02 03 04 05
byte_align_be[gpr1,*\$index++]	0x04 05 06 07	0x04 05 06 07	0x08 09 0A 0B	0x06 07 08 09
byte_align_be[gpr2,*\$index++]	0x08 09 0A 0B	0x08 09 0A 0B	0x0C 0D 0E 0F	0x0A 0B 0C 0D

**Note:**

1. b\_align = gpr containing byte align value (2 in this example) and \$xfer0 is starting register. The & is a preprocessor directive that returns the address of the register.

## 2.2.13 CAM\_CLEAR

Clears all entries in the CAM by writing 0x00000000 to the tag, clearing all the state bits, and putting the Least Recently Used (LRU) into an initial state where entry CAM 0 is LRU, ..., CAM entry 15 is Most Recently Used (MRU).

The CAM is not reset by FPC reset. Software must either do a cam\_clear prior to using the CAM to initialize the LRU and clear the tags to 0, or explicitly write all entries with cam\_write.

### Instruction Format

cam_clear
-----------

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.14 CAM\_LOOKUP

Search the 16 entry CAM for a 32-bit tag equal to the value specified in the src\_reg. All entries are compared in parallel, and the result of the lookup is a 9 bit value which is written into the specified destination register in bits [11:3], with all other bits of the register 0. The result can also optionally be written into either of the LM\_Addr registers.

The 9-bit result consists of 4 State bits (dest\_reg[11:8]), concatenated with a 1-bit Hit/Miss indication (dest\_reg[7]), concatenated with 4-bit entry number (dest\_reg[6:3]). All other bits of dest\_reg are written with 0. Possible results of the lookup are:

- miss (0) – lookup value is not in CAM, entry number is LRU (Least Recently Used) entry (which can be used as a suggested entry to replace), and State bits are 0000.
- hit (1) – lookup value is in CAM, entry number is entry which has matched, State bits are the value from the entry which has matched. The entry is marked as MRU (Most Recently Used).

An optional token allows the result to also be written into either of the LM\_Addr registers.

The LRU (Least Recently Used) is maintained in a time-ordered CAM entry usage list. When an entry is loaded, or matches on a lookup, it is marked as MRU (Most Recently Used). Note that a lookup that misses does not modify the LRU list.



### Note

The following rules must be followed to when using the CAM.

1. CAM is not reset by a FPC reset. Software must either do a CAM\_clear prior to using the CAM to initialize the LRU and clear the tags to 0, or explicitly write all entries with CAM\_write.
2. No two tags can be written to have same value. If this rule is violated, the result of a lookup that matches that value will be unpredictable, and the LRU state is unpredictable.

The value 0x00000000 can be used as a valid lookup value. However, note that CAM\_clear instruction puts 0x00000000 into all tags. To avoid violating rule 2 after doing CAM\_clear, it is necessary to write all entries to unique values prior to doing a lookup of 0x00000000.

### Instruction Format

```
cam_lookup[dest_reg, src], opt_tok
```

### Parameter Descriptions

Parameter	Description
dest_reg	Unrestricted destination that receives the result of the CAM lookup. Refer to <a href="#">Table 2.23</a> for result format.

Parameter	Description
src	Unrestricted source operand that holds the value to lookup in the CAM.
opt_tok	<p>gpr_wrbboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>lm_addr#[num]: Load the result of the lookup into LM_Addr (# = 0 or 1), as well as dest. Bits[6:3] of CAM result go to LM_Addr[9:6]. Num specifies a 2-bit value to load into LM_Addr[11:10]. LM_Addr[5:0] is set to 0. The write latency for loading the LM_Addr (# = 0 or 1) is 3 cycles which is the same as if a local_csr_wr instruction were used.</p> <p>predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

**Table 2.23. CAM\_LOOKUP Result**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	state	hit/miss	CAM Entry Number	0	0	0			

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.15 CAM\_READ\_TAG

Read the tag for the specified CAM entry into dest\_reg.

#### Instruction Format

```
cam_read_tag[dest_reg, entry], opt_tok
```

#### Parameter Descriptions

Parameter	Description
dest_reg	Unrestricted destination that receives the tag data from the CAM entry.
entry	Unrestricted source operand that specifies the CAM entry number to read. Valid values are 0 to 15.
opt_tok	gpr_wrbboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a>

Parameter	Description
	predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.16 CAM\_READ\_STATE

Read the State bits for the specified CAM entry. The value is placed into bits [11:8] of dest\_reg, with all other bits 0.

#### Instruction Format

```
cam_read_state[dest_reg, entry], opt_tok
```

#### Parameter Descriptions

Parameter	Description
dest_reg	Unrestricted destination that receives the State bits from the CAM entry. Dest_Reg gets State in bits [11:8], all other bits are 0.
entry	Unrestricted source operand that specifies the CAM entry number to read. Valid values are 0 to 15.
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>predicate_cc: Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

**Table 2.24. CAM\_READ\_STATE Result**

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

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.17 CAM\_WRITE

Write a 32-bit value in src\_reg to the tag of the specified CAM entry. The entry is marked as MRU (Most Recently Used).



### Note

No two tags can be written to have same value. If this rule is violated, the result of a lookup that matches that value will be unpredictable, and LRU state is unpredictable.

### Instruction Format

```
cam_write[entry_reg, src_reg, state_value], opt_tok
```

### Parameter Descriptions

Parameter	Description
entry	Unrestricted source operand that specifies the CAM entry number to write. Valid values are 0 to 15.
src_reg	Unrestricted source operand that holds the data to write to the CAM entry.
state_value	Constant that specifies the State bit value.

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.18 CAM\_WRITE\_STATE

Write the value into the State bits for the specified MEv2 CAM entry. The Tag value is not changed. This instruction does not modify the Least-Recently-Used (LRU) list.

### Instruction Format

```
cam_write_state[entry, state_value], opt_tok
```

### Parameter Descriptions

Parameter	Description
entry	Unrestricted source operand that specifies the CAM entry number to modify. Only the State bits are affected.

Parameter	Description
state_value	Constant that specifies the State bit value. Valid values are 0 to 0xF.

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.19 CRC\_LE, CRC\_BE

Enable the CRC datapath to compute a CRC. The CRC is calculated using a remainder that resides in the CRC\_Remainder Local CSR and the source data. The result is written to the CRC\_Remainder Local CSR and the unmodified source data can be written to a destination register. This allows a CRC to be calculated and the result moved from one register to another (example: transfer in to transfer out) in the same instruction.

The CRC\_Remainder Local CSR is typically initialized prior to doing the CRC instruction(s). CRC instruction cannot be used consecutively. At least one intervening instruction must be done between two CRC instructions, and 5 intervening instruction before reading the CRC result. The first example shows how a CRC can be calculated over a block of data by interleaving the CRC instructions.

The `crc_le` instruction (little endian) is used when the data is in little endian mode so the bytes are swapped before the CRC is performed. The `crc_be` instruction (big endian) is used when the data is in big endian format and no bytes are swapped before the CRC is performed. Refer to the following examples.

### Instruction Formats

<code>crc_le[crc_type, dest_reg, src], opt Tok</code>
<code>crc_be[crc_type, dest_reg, src], opt Tok</code>

### Parameter Descriptions

Parameter	Description	
crc_type#	crc_ccitt	CRC-CCITT polynomial is: $x^{16} + x^{12} + x^5 + 1$ .
	crc_32	CRC-32 polynomial is: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ .
	crc_iscsi	CRC-iscsi polynomial is: $x^{32} + x^{28} + x^{27} + x^{26} + x^{25} + x^{23} + x^{22} + x^{20} + x^{19} + x^{18} + x^{14} + x^{13} + x^{11} + x^{10} + x^9 + x^8 + x^6 + 1$ .
	crc_10	CRC-10 polynomial is: $x^{10} + x^9 + x^5 + x^4 + x + 1$ .
	crc_5	CRC-5 polynomial is: $x^5 + x^2 + 1$ .
	none	Pass the input data through into CRC_Remainder without any CRC computation. This can be used to do bit and byte swapping on the input data.

Parameter	Description			
dest	All crc types	Unrestricted destination that gets written with the unmodified src operand.		
src	All crc types	Unrestricted source operand.		
opt_tok	crc_ccitt,  crc_32,  crc_iscsi,  crc_10 crc_5	One of the following tokens can be used to limit which bytes of data are used in the computation. If no token is used, all four bytes are used. These are used in the beginning and end of a CRC computation to skip leading and trailing byte positions. The meaning of the token can be thought of in two ways: 1. It specifies the bytes, in order of left to right, of the post-swapped big endian data. 2. It specifies the bytes of the pre-swapped data in the order implied by the endianness.		
		opt_tok	Bytes in source used for CRC calculation	
			Big endian	Little Endian
	bytes_0_3	0,1,2,3	3,2,1,0	
	bytes_0_2	0,1,2,-	-,2,1,0	
	bytes_0_1	0,1,-,-	-,-,1,0	
	byte_0	0,-,-,-	-,-,-,0	
	bytes_1_3	-,1,2,3	3,2,1,-	
	bytes_2_3	-,-,2,3	3,2,-,-	
	byte_3	-,-,-,3	3-, -, -	
	All CRC types	bit_swap: Swaps the bits in each individual byte so that bit 7 is swapped with bit 0, bit 6 is swapped with bit 1, bit 5 is swapped with bit 2, and bit 4 is swapped with bit 3		
	gpr_wrboth	For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a>		
	no_cc	Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a>		
	predicate_cc	Qualify the dest_reg write-back and the updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>		

### Condition Codes Affected

N	Z	V	C
Result[31] == 1	Result[31:0] == 0	Cleared	Cleared

### Example: Loading, Calculating and Reading the CRC

```
; Assume data to CRC is in $transfer_in[3:0]
; Current remainder has been put into gpr_n
```

```
; Compute CRC while moving data to transfer_out[3:0]
local_csr_wr[crc_remainder, gpr_n] ;restore remainder
crc[crc_ccitt, $trans_out_0, $trans_in_0]
nop
crc[crc_ccitt, $trans_out_1, $trans_in_1]
nop
crc[crc_ccitt, $trans_out_2, $trans_in_2]
nop
crc[crc_ccitt, $trans_out_3, $trans_in_3]
nop
nop
nop
nop
nop ; five intervening instructions before reading result
local_csr_rd[crc_remainder] ;get the new remainder
immed[gpr_n, 0x0000]
```

#### Example: crc\_le[ ] with bytes\_0\_1

```
case 1#:
;data from alu out: M L C S
crc_le[], bytes_0_1
;data after swapping: S C L M
;CRC computed on: S C
```

#### Example: crc\_be[ ] with bytes\_2\_3

```
case 2#:
;data from alu out: M L C S
crc_be[], bytes_2_3
;data after swapping: M L C S
;CRC computed on: C S
```

#### Example: crc\_le[] with bytes\_2\_3

```
case 3*:
;data from alu out: M L C S
crc_le[], bytes_2_3
;data after swapping: S C L M
;CRC computed on: L M
```

#### Example: crc\_be[] with bytes\_0\_1

```
case 4#:
;data from alu out: M L C S
crc_be[], bytes_0_1
;data after swapping: M L C S
;CRC computed on: M L
```

## 2.2.20 CTX\_ARB

Swap the currently running context out to let another context execute. Wake up the swapped out context when the specified signal(s) is activated.

Please refer to the *Event Manager Peripheral* and the *Interrupt Manager Peripheral* sections of the *Netronome Network Flow Processor 6xxx: Databook* .

## Instruction Format

```
ctx_arb[Event_Signal_Mask], op_tok
```

### Parameter Descriptions

Parameter	Mask option	Description
Event_ Signal_ Mask	signal list	A list of signal names separated by commas. The thread is put to sleep and woken based on the state of the signals and the ANY and ALL optional tokens.
	Voluntary	Voluntary is a keyword that indicates that the thread should be put to sleep and woken when all the other threads have had a chance to run.
	Kill	Kill is a keyword that indicates the current thread should be put to sleep and never woken.
	bpt	<p>bpt is a keyword that indicates the current thread should be put to sleep, no thread should be woken, and that an ATTN signal should be sent. This is typically used for breakpoints. The actions taken are as follows:</p> <ul style="list-style-type: none"> <li>• Clear the CTX_Enable for all contexts in the FPC.</li> <li>• Put the currently executing Context into Sleep state.</li> <li>• Sets the CTX_Enable[Breakpoint] bit in the FPC local CSR .</li> </ul> <p>Note that the CTX_WAKEUP_EVENTS is cleared for the context that is running, therefore software running on the ARM or PCIe attached host must set the voluntary bit to restart the context.</p>
	--	The value “--” indicates that the instruction does not contain a list of Event Signals; that list must instead be loaded using the instruction local_csr_wr[CTX_Wakeup_Events, src]. This can be the instruction that immediately precedes the ctx_arb[--] instruction, or in the defer shadows of the ctx_arb instruction. The local_CSR_wr can be to either CTX_Wakeup_Events_Active or CTX_Wakeup_Events_Indirect CSRs.
opt Tok	All mask options	defer[n] (n = 1 to 2)
	signal list, --	ALL: Activate the context when all of the listed signal(s) is received. This also clears the contexts

Parameter	Mask option	Description
		sig_events and wakeup_events when the context is woken. If no token is used, ALL is assumed. ALL and ANY are mutually exclusive.
		ANY: Activate the context when any of the listed signal(s) is received. This also clears the context wakeup_events when the context is put in Ready state, but leave the contexts sig_events unchanged. The thread can then use the br_signal or br_!signal to determine which signal event woke it. ANY and ALL are mutually exclusive.
	signal list, -- , Voluntary	br[label#]: Resume execution at the address specified by the label when the context wake up. If this token is not used executions resumes at the next sequential instruction. The defer optional token can be used with this token.

### Example: CTX\_ARB

```
;example 1: signal list mask option with optional token = any
ctx_arb[test1, test2], any

;example 2: -- mask option
ctx_arb[ -- ],defer[1]
local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, 0x2] ; wake on signal 1

;example 3: branch optional token with --
ctx_arb[--],br[label#],defer[1]
local_csr_wr[ACTIVE_CTX_WAKEUP_EVENTS, 0x2] ; wake on signal 1

;example 4: Voluntary mask option
ctx_arb[voluntary]
```

## 2.2.21 DBL\_SHF

Load a destination register with a 32-bit word that is formed by concatenating the A operands and B operands together (A is most significant, B is least significant), right shifting the 64-bit quantity by the specified amount, and then storing the lower 32 bits.

### Instruction Format

```
dbl_shf[dest, A_op, B_op, shf_cntl], opt_tok
```

### Parameter Descriptions

Parameter	Description
dest	Restricted destination that gets written with the result of the operation.

Parameter	Description
A_op, B_op	Restricted source operand.
shf_cntl	>>1 through >>31: (Right shift 1) through (Right shift 31). >>indirect: Right shift by the indirect value. The indirect value is specified by the previous instruction which must be an ALU or ALU_SHF and the shift amount is specified in the lower 5 bits of the A_op parameter (which must be a register - not a constant).
opt_tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a> no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a> predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

### Condition Codes Affected

N	Z	V	C
Result[31]==1	Result[31:0] == 0	Cleared	Cleared

#### Example: DBL\_SHF >>value

```
If a = 0x87654321 and b = 0xFEDCBA98, then
dbl_shf[c, a, b, >>12]
saves 0x321FEDCB in c.
```

#### Example: DBL\_SHF >>indirect

```
If a = 0x87654321, b = 0xFEDCBA98, shf_value = 12, any_reg = any value (it's not
used) then
alu[--,shf_value,OR,any_reg]
dbl_shf[c, a, b, >>indirect]
saves 0x321FEDCB in c.
```

## 2.2.22 FFS

Find First Set bit in the src register beginning at the LSB. Dest[4:0] gets the encoded value from 0 to 31 indicating the least significant 1 bit in the src. If there are no bits set, the Z condition code is set; otherwise the Z condition code is cleared.

## Instruction Format

```
ffs[dest, src], opt_tok
```

## Parameter Descriptions

Parameter	Description
dest	Unrestricted destination that receives the result of the operation. Dest[4:0] receives a value from 0 to 31 indicating first bit found starting at the LSB in the src. If there are no bits set, the Z condition code is set; otherwise the Z condition code is cleared. All other bits of dest receive 0.
src	Unrestricted operand for the operation (B operand).
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a></p> <p>predicate_cc: Qualify the dest write-back and updating of flags depending on the value of the instruction CC and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

## Condition Codes Affected

N	Z	V	C
Cleared	Set if B == 0	Cleared	Cleared

## 2.2.23 HALT

This instruction puts the current thread to sleep without waking up any other thread. The FPC will assert its ATTN output signal, which in turn can become an event on the Event Bus. This instruction is equivalent to instruction “ctx\_arb[bpt]”. The actions taken are as follows:

- Clear the CTX\_Enable for all contexts in the FPC.
- Put the currently executing Context into Sleep state.
- Sets the CTX\_Enable[Breakpoint] bit in the FPC local CSR .

Note that the CTX\_WAKEUP\_EVENTS is cleared for the context that is running, therefore software running on the ARM processor or PCIe attached host must set the voluntary bit to restart execution.

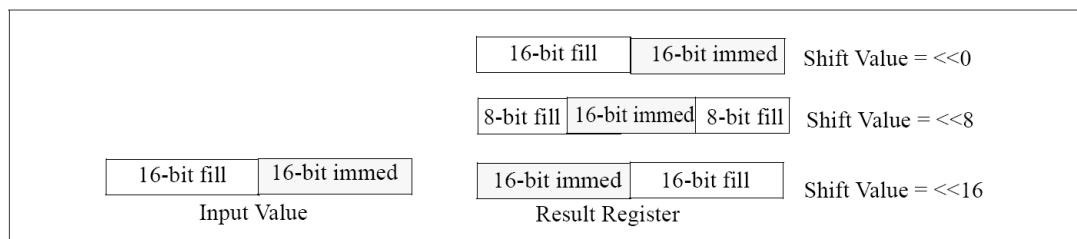
Please refer to the *Event Manager Peripheral* and the *Interrupt Manager Peripheral* sections of the *Netronome Network Flow Processor NFP-6xxx: Databook* .

## Instruction Format

```
HALT
```

## 2.2.24 IMMED

Load immediate 16-bits into the specified register. The immediate data must be specified having the upper 16-bits equal to either all zeros or ones. As shown in [Figure 2.1](#), the immediate data can be stored in the 32-bit word aligned on an 8-bit boundary based on the optional shift parameter. The fill data is either all zeros or ones and is based on the specified upper 16-bits.



**Figure 2.1. Load Immediate**

### Instruction Format

```
immed[dest, immed_data, shf_ctrl], opt_tok
```

### Parameter Descriptions

Parameter	Description
dest	Unrestricted operand. If the register is specified as a transfer register the result is always placed into the transfer out register.
immed_data	32-bit data having the upper bits all zeroes or all ones and the lower 16-bits user defined.
shf_ctrl	0, <<0, or, no character: No shift. <<8 : Left shifts one byte. <<16: Left shifts two bytes.
opt_tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a> predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

### Condition Codes Affected

N	Z	V	C
Not Affected			

### **Example: IMMED**

```
;Negative number (-256 = 0xFFFF FF00)
immmed[dest_reg10, -256, <<0];Result: 0xffff ff00
immmed[dest_reg11, -256, <<8];Result: 0xffff 00ff
immmed[dest_reg12, -256, <<16];Result: 0xff00 ffff
```

### **Example: IMMED**

```
;Hexadecimal number - zero fill
immmed[dest_reg04, 0xff00, <<0] ;Result: 0x0000 ff00
immmed[dest_reg05, 0xff00, <<8] ;Result: 0x00ff 0000
immmed[dest_reg06, 0xff00, <<16] ;Result: 0xff00 0000
```

## **2.2.25 IMMED\_B0, IMMED\_B1, IMMED\_B2, IMMED\_B3**

The specified dest\_reg is read as a source, one byte of immediate data is loaded into the specified byte, and the result is written into the destination, while preserving all the other bits of the source value. These instructions perform a read-modify-write operation on a specified destination register.

If a transfer register is specified as the dest\_reg, these instructions perform a read and modify from a read transfer register and write the result into a transfer out register.

If a Neighbor register is specified, the FPCs Next Neighbor register is read, modified and then stored into the Next Neighbor FPC.

B0 refers to the least significant byte.

### **Instruction Formats**

immmed_b0[dest_reg, byte_data], opt_tok
immmed_b1[dest_reg, byte_data], opt_tok
immmed_b2[dest_reg, byte_data], opt_tok
immmed_b3[dest_reg, byte_data], opt_tok

### **Parameter Descriptions**

<b>Parameter</b>	<b>Description</b>
dest_reg	Unrestricted operand that receives the result of the operation.
immmed_data	Immediate byte to be loaded into dest_reg.
opt_tok	<p>gpr_wrbboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to</p>

Parameter	Description
	the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.26 IMMED\_W0, IMMED\_W1

The specified dest\_reg is read as a source, one word of immediate data is loaded into the specified word, and the result is written into the destination, while preserving all the other bits of the source value. These instructions perform a read-modify-write operation on a specified destination register.

If a transfer register is specified as the dest\_reg, these instructions perform a read and modify from a read transfer register and writes the result into a write transfer register.

If a Neighbor register is specified, the FPC's Next Neighbor register is read, modified and then stored into the Next Neighbor FPC.

W0 refers to the least significant word. W1 refers to the most significant word.

### Instruction Formats

immed_w0[dest_reg, immed_data], opt_tok
immed_w1[dest_reg, immed_data], opt_tok

### Parameter Descriptions

Parameter	Description
dest_reg	Unrestricted operand that receives the result of the operation.
immed_data	Immediate 2 bytes to be loaded into dest_reg. Valid immed_data values are 0 to 0xFFFF.
opt_tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a> predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.27 JUMP

Unconditional branch to an address that is formed during runtime execution by the addition of the register and label# values.

### Instruction Format

```
jump[src, label#], opt_tok
```

### Parameter Descriptions

Parameter	Description
src	Unrestricted operand that holds the value added to the address of the specified label#. Immediate data is not supported.
label#	Symbolic label corresponding to the base address for the jump.
opt Tok	defer[n] (n= 1 to 3) refer to <a href="#">Section 2.1.5</a> . targets [label1, label2, ...labeln] This is always required. It is a list of labels that represents all possible locations to which the jump could occur. All possible targets must be specified in the target list, otherwise the code produced by the assembler may not run as expected. The list is for the assembler to follow execution of the code and properly encode offsets.

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Example: Jump

```
;Possible Results:  
;if offset = 0, result = 1  
;if offset = 2, result = 2  
;if offset = 4, result = 3  
immed[offset, 2]  
jump[offset, base0#], targets [base0#, base1#, base2#]  
continue#:  
br[continue#]  
base0#:;base0 + 0  
immed[result, 1]  
br[continue#]  
base1#:;base0 + 2  
immed[result, 2]  
br[continue#]
```

```
base2#:;base0 + 4
immed[result, 3]
br[continue#]
```

## 2.2.28 LD\_FIELD, LD\_FIELD\_W\_CLR

Load one or more bytes positions within a register with the shifted value of another operand. Data in the bytes that are not loaded remain unchanged or are cleared. LD\_FIELD performs a read-modify-write on a destination register. LD\_FIELD\_W\_CLR performs a write to a destination register.

If a transfer register is specified as the dest\_reg for LD\_FIELD, these instructions perform a read and modify from a read transfer register and writes the result into a write transfer register.

In some cases LD\_FIELD instruction is useful feature when doing 1 byte modifications to CLS or other memory that come in and go out via xfer registers.

If a Neighbor register is specified for LD\_FIELD, the FPC's Next Neighbor register is read, modified and then stored into the Next Neighbor FPC.

### Instruction Formats

ld_field[dest_reg, byte_ld_enables, src_op, opt_shf_cntl], op_tok
ld_field_w_clr[dest_reg, byte_ld_enables, src_op, opt_shf_cntl], op_tok

### Parameter Descriptions

Parameter	Description
dest_reg	Restricted operand (Context-Relative) that receives the result of the operation. Note that this operand is also used as a source. Note that *n \$index++ cannot be used as destination for ld_field_w_clr instruction.
byte_ld_enables	A 4-bit mask that specifies which byte(s) are affected by the instruction. Each set bit enables the corresponding byte of the destination operand 32-bit word to be loaded or cleared. There must be at least 1 set bit in this mask. For example, 0101 loads the 1st and 3rd bytes while the other bytes remain unchanged.
src_op	Restricted source (Context-Relative). If a GPR, this register must be on the opposite bank as the destination register. Refer to <a href="#">Table 2.4</a> for source register selection rules.
opt_shf_cntl	Shift or rotate the source_op contents using the syntax shown below. <<n: Left shift n bits, where n = 1 through 31. <<indirect: Left shift by an amount specified in the lower 5 bits of the A operand of the previous instruction (the previous instruction must be one of the following ALU or ALU_SHF instructions -- A AND B, A AND ~B, A XOR B, A OR B). The lower 5 bits of the A operand should be n, where n is the desired left shift amount.

Parameter	Description
	>>n: Right shift n bits, where n = 1 through 31.
	>>indirect: Right shift by the amount specified in the lower 5 bits of the A operand of the previous instruction.
	<<rotn: Left rotate n bits, where <<rot is a keyword and n = 1 to 31.
	>>rot : Right rotate n bits, where >>rot is a keyword and n = 1 to 31.
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>load_cc : Load ALU condition codes based on result formed.</p> <p>predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

### Condition Codes Affected

Optional token	N	Z	V	C
load_cc	Result[31] == 1	Result[31:0] == 0	Cleared	Cleared
not load_cc	Not Affected			

## 2.2.29 LOAD\_ADDR

Load a register with an address of the location specified by label#.

### Instruction Format

```
load_addr[dest, label#], opt_tok
```

### Parameter Descriptions

Parameter	Description
dest	Unrestricted destination that receives the result of the operation.
label#	Symbolic label corresponding to the address of an instruction.
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

## Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.30 LOCAL\_CSR\_RD

Read the specified FPC CSR register. The read data is accessed by replacing the immediate data source operand of the next immed instruction with the FPC CSR read data. If the very next instruction does not contain an immediate data source operand field, then the opportunity to access the CSR data read from the previous instruction is lost.

### Instruction Format

```
local_csr_rd[local_csr]
```

### Parameter Descriptions

Parameter	Description	
local_csr	Specifies the Local CSRs.	
<b>CSR Names</b>		
USTORE_ADDRESS	INDIRECT_LM_ADDR_1	
USTORE_ADDRESS_1	INDIRECT_LM_ADDR_1_BYTE_INDEX	
ALU_OUT	ACTIVE_LM_ADDR_1	
CTX_ARB_CNTL	ACTIVE_LM_ADDR_1_BYTE_INDEX	
CTX_ENABLES	BYTE_INDEX	
CC_ENABLE	T_INDEX	
CSR_CTX_POINTER	T_INDEX_BYTE_INDEX	
INDIRECT_CTX_STS	INDIRECT_FUTURE_COUNT_SIGNAL	
ACTIVE_CTX_STS	ACTIVE_FUTURE_COUNT_SIGNAL	
ACTIVE_CTX_SIG_EVENTS	NN_PUT	
INDIRECT_CTX_SIG_EVENTS	NN_GET	
INDIRECT_CTX_WAKEUP_EVENTS	TIMESTAMP_HIGH, TIMESTAMP_LOW	
ACTIVE_CTX_WAKEUP_EVENTS	CRC_REMAINDER	
INDIRECT_CTX_FUTURE_COUNT	NEXT_NEIGHBOR_SIGNAL	
ACTIVE_CTX_FUTURE_COUNT	PREV_NEIGHBOR SIGNAL	
INDIRECT_LM_ADDR_0	SAME_ME_SIGNAL	
INDIRECT_LM_ADDR_0_BYTE_INDEX	PROFILE_COUNT	

Parameter	Description		
	ACTIVE_LM_ADDR_0	PSEUDO_RANDOM_NUMBER	
	MAILBOX_0	USTORE_ERROR_STATUS	
	MAILBOX_1	ACTIVE_LM_ADDR_0_BYTE_INDEX	
	MAILBOX_2		
	MAILBOX_3		

#### Condition Codes Affected

N	Z	V	C
Not Affected			

#### Example: LOCAL\_CSR\_RD

Here is an example of how to access the read data:

```
local_csr_rd[timestamp_low]
immed[gpr_n,0]
```

### 2.2.31 LOCAL\_CSR\_WR

Write specified FPC CSR register with the data in the specified source register. There is always a delay between local\_csr\_wr and the value changed.

#### Instruction Format

```
local_csr_wr[local_csr, src]
```

#### Parameter Descriptions

Parameter	Description	
local_csr	Specifies the Local CSRs.	
	<b>CSR Names</b>	
	CTX_ARB_CNTL	BYTE_INDEX
	CTX_ENABLES	T_INDEX
	CC_ENABLE	T_INDEX_BYTE_INDEX
	CSR_CTX_POINTER	INDIRECT_FUTURE_COUNT_SIGNAL
	INDIRECT_CTX_STS	ACTIVE_FUTURE_COUNT_SIGNAL
	ACTIVE_CTX_STS	NN_PUT
	INDIRECT_CTX_SIG_EVENTS	NN_GET

Parameter	Description	
	ACTIVE_CTX_SIG_EVENTS	TIMESTAMP_HIGH, TIMESTAMP_LOW
	INDIRECT_CTX_WAKEUP_EVENTS	NEXT_NEIGHBOR_SIGNAL
	ACTIVE_CTX_WAKEUP_EVENTS	PREV_NEIGHBOR_SIGNAL
	INDIRECT_CTX_FUTURE_COUNT	SAME_ME_SIGNAL
	ACTIVE_CTX_FUTURE_COUNT	CRC_REMAINDER
	INDIRECT_LM_ADDR_0	PROFILE_COUNT
	INDIRECT_LM_ADDR_0_BYTE_INDEX	PSEUDO_RANDOM_NUMBER
	ACTIVE_LM_ADDR_0	USTORE_ADDRESS
	ACTIVE_LM_ADDR_0_BYTE_INDEX	USTORE_ADDRESS_1
	INDIRECT_LM_ADDR_1	USTORE_ERROR_STATUS
	INDIRECT_LM_ADDR_1_BYTE_INDEX	MAILBOX_0
	ACTIVE_LM_ADDR_1	MAILBOX_1
	ACTIVE_LM_ADDR_1_BYTE_INDEX	MAILBOX_2
		MAILBOX_3
src	Unrestricted source operand for the operation.	

#### Condition Codes Affected

N	Z	V	C
Not Affected			

### 2.2.32 MUL\_STEP

Used in a multi-instruction operation to multiply two unsigned numbers. Multiplication is done 8- bits per step. The following examples show the sequence to use to multiply different size operands. Operands of less than 32 bits must have 0s into unused leading bits. Note that the final add sets the Condition Codes.



#### Note

For mul\_step, 32x32\_last2, the N and Z condition codes reflect the value of the 32 most significant bits of the multiply result. The programmer can use the ALU “B” operation to test the N and Z condition codes of the 32 least significant bits of the multiply result.

opt\_tok is allowed with mult\_step instructions that carries a destination.

#### Instruction Formats

mul_step[A_op, B_op], tok
---------------------------

```
mul_step[dest, --], tok, opt_tok
```

## Parameter Descriptions

Parameter	Description
A_op	Unrestricted source operand (multiplicand). Used when one of the “step” or “start” tokens are specified.
B_op	Unrestricted source operand (multiplier).Used when one of the “step” or “start” tokens are specified.
dest	Unrestricted destination operand. Used when one of the “last” tokens are specified. Note that *n \$index++ cannot be used as destination.
--	Must always be “--” when the “Last” tokens are specified
tok	24x8_start      Indicates the beginning of a new multiply sequence. Load a_source and 16x16_start      b_source into the multiplier array. A separate token is provided for each type of 32x32_start      multiply; 24x8, 16x16, or 32x32.  24x8_step1 16x16_step1 16x16_step2 32x32_step1      Number of the step. 1 step for 24x8, 2 steps for 16x16, 4 steps for 32x32. 32x32_step2 32x32_step3 32x32_step4  24x8_last 16x16_last      Final add to produce the low 32-bits of the product. 32x32_last  32x32_last2      Final add to produce the upper 32-bits of a 64-bit product.
opt Tok	gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a>  no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a>  predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a>

## Condition Codes Affected

Optional token	N	Z	V	C
Last	Result[31] == 1	Result[31:0] == 0	Undefined	Undefined
Last2				
Others		Not Affected		

## Example: 8 x 24 multiply (8 bit number is multiplier)

```
mul_step[multiplicand,multiplier], 24x8_start
mul_step[multiplicand,multiplier], 24x8_step1
mul_step[dest,--], 24x8_last
```

#### Example: 16 x 16 multiply

```
mul_step[multiplicand,multiplier], 16x16_start
mul_step[multiplicand,multiplier], 16x16_step1
mul_step[multiplicand,multiplier], 16x16_step2
mul_step[dest,--], 16x16_last
```

#### Example: 32 x 32 multiply with 32 bit result

```
mul_step[multiplicand,multiplier], 32x32_start
mul_step[multiplicand,multiplier], 32x32_step1
mul_step[multiplicand,multiplier], 32x32_step2
mul_step[multiplicand,multiplier], 32x32_step3
mul_step[multiplicand,multiplier], 32x32_step4
mul_step[dest,--], 32x32_last
```

**Note:** Overflow above 32 bit result will not be detected; if the programmer is not ensured of that based on input operands, use the 64 bit version in next Example.

#### Example: 32 x 32 multiply with 64 bit result

```
mul_step[multiplicand,multiplier], 32x32_start
mul_step[multiplicand,multiplier], 32x32_step1
mul_step[multiplicand,multiplier], 32x32_step2
mul_step[multiplicand,multiplier], 32x32_step3
mul_step[multiplicand,multiplier], 32x32_step4
mul_step[dest_low,--], 32x32_last
mul_step[dest_high,--], 32x32_last2
```

## 2.2.33 NOP / NOP\_VOLATILE

Consume one microcycle without performing any operation and without setting any FPC state. A `nop` can be removed or replaced by the assembler optimizer. A `nop_volatile` will not be removed or replaced by the assembler optimizer.

### Instruction Format

nop
nop_volatile

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.34 POP\_COUNT

Find number of 1 bits in the SRC register. To find the number of 1s in a partial field, precede the pop\_count instruction with an ALU AND instruction to mask off undesired bits. If there are no bits set, the Z condition code is set; otherwise the Z condition code is cleared. This function is implemented in three consecutive instructions, pop\_count1, pop\_count2, and pop\_count3 that must be executed consecutively, with each specifying the same source operand and only the third instruction specifying the destination operand. If opt\_tok is used with three consecutive pop\_counts it would be applied to the last pop\_count instruction that has a destination.

### Instruction Formats

pop_count1[src]
pop_count2[src]
pop_count3[dest, src], opt_tok

### Parameter Descriptions

Parameter	Description
dest	Unrestricted destination that receives the result of the operation. dest[5:0] receives a value from 0 to 32 indicating the number of 1 bits in the src. If there are no bits set, the Z condition code is set; otherwise the Z condition code is cleared. All other bits of dest receive 0.
src	Unrestricted operand for the operation (B operand). The source has to be the same for all three instructions.
opt_tok	<p>gpr_wrboth: For GPR destinations, write both, the A and B banks, at the address specified in GPR dest. Refer to <a href="#">Section 2.1.3.1</a></p> <p>no_cc: Turn off the setting of condition codes. Refer to <a href="#">Section 2.1.3.2</a></p> <p>predicate_cc: Qualify the dest_reg write-back and updating of flags depending on the value of the selected predicate condition code prior to the instruction execution and the IndPredCC register setting. Refer to <a href="#">Section 2.1.3.3</a></p>

### Condition Codes Affected

N	Z	V	C
Cleared	Set if src == 0	Cleared	Cleared

## 2.2.35 RTN

Unconditional branch to the address contained in the lower 14 bits of the specified register. Typically used to return from a branch or jump instruction.

## Instruction Format

```
rtn[reg], opt_tok
```

## Parameter Descriptions

Parameter	Description
reg	Unrestricted source that contains the return address. The return address is typically loaded into the register using the load_addr instruction.
opt Tok	defer[n] (n= 1 to 3) refer to <a href="#">Section 2.1.5</a> .

## Condition Codes Affected

N	Z	V	C
Not Affected			

## Example: jump and rtn

```
load_addr[rtn_reg, rtn_label#]
br[sub_routine#]
rtn_label#:
.subroutine
sub_routine#:
; ----- do some work here
rtn[rtn_reg]
.endsub
```

## 2.2.36 ARM

### 2.2.36.1 ARM (Read and Write)

Read or write from/to the address of the shared memory block or peripherals inside the ARM11 subsystem.

## Instruction Format

```
arm[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

## Parameter Descriptions

Parameter	Command	Description
cmd	read	Read 64-bit words from the ARM11 subsystem. This operation only supports full 64-bit word reads.

Parameter	Command	Description
	write	Write 64-bit words to the ARM11 subsystem. The write operation only supports full 64-bit word writes with no byte mask.
xfer	read	Read transfer registers.
	write	Write transfer registers.
src_op1, src_op2	Both commands	<p>Restricted source operands for ARM address. The address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a></p> <p>SRAM: Address[15:3] is address.</p> <p>Peripheral: Address[23:20] is peripheral address (!=0), Address[19:2] is address.</p> <p>The upper 128 64B addressable locations of the shared memory are reserved for the operation of the ARM11 subsystem. Accessing the reserved areas of the shared memory while the ARM core is enabled, is not recommended, but is permitted by the hardware.</p>
ref_cnt	Both commands	Reference count in 8-byte words. Valid values are 1 for Peripheral and 1-16 for SRAM.
opt_tok	Both commands	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.37 Cluster Local Scratch (CLS)

### 2.2.37.1 CLS (Atomic Operations)

Issue a memory reference to perform an atomic operation on data in Cluster Local Scratch memory. An atomic operation is one in which a read, modify, and write operation is performed on the memory location and it is assured that another operation will not be allowed to access the data while the atomic operation is in progress.

#### Instruction Format

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

## Parameter Descriptions

Parameter	Command	Description
cmd	add	Add the 32-bit value(s) in the transfer register(s) to the 32-bit value(s) at the specified address.
	add_imm	Add 16-bit immediate to the 32-bit value at the specified address. The two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a>
	add_imm_sat	Refer to addsat_imm.
	add_sat	Add the 32-bit value(s) in the transfer register(s) to the 32-bit value(s) at the specified address. Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	addsat	Refer to add_sat.
	addsat_imm	Add 16-bit immediate value to the 32-bit value at the specified address. The two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a> . Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	add64	Add the 64-bit value(s) in the transfer register pairs to the 64-bit value(s) at the specified address.
	add64_imm	Add 16-bit immediate to the 64-bit value at the specified address.
	clr	Clear the bit(s) at the specified address according to a bit mask provided in the transfer register(s). A 1 in the bit position of the bit mask signifies that the bit should be cleared.
	clr_imm	Clear the bit(s) at the specified address according to a bit mask provided by the immediate data. A 1 in the bit position of the bit mask signifies that the bit should be cleared. Only operates at a single 32-bit word. See <a href="#">Note 1</a> (with bit 15 zero).
	cmp_rw	Compare memory (word-by-word) against bytes from the pull-from-write-xfer registers indicated by the byte_mask; if all indicated bytes in a word match, replace the memory with the pull-from-write-xfer register word. Return the unmodified memory contents.
	decr	Single instruction to decrement by 1 a 32-bit value in memory. Implemented as add_imm command with an immediate of 0xff, ie. -1.
	decr64	Single instruction to decrement by 1 a 64-bit value in memory. Implemented as add64_imm command with an immediate of 0xff, ie. -1.
	incr	Single instruction to increment by 1 a 32-bit value in memory. Implemented as sub_imm command with an immediate of 0xff, ie. -1.

Parameter	Command	Description
	incr64	Single instruction to increment by 1 a 64-bit value in memory. Implemented as sub64_imm command with an immediate of 0xff, ie. -1.
	meter	Perform a metering operation. Address[18] select RFC Type: 0 == RFC2698, 1 == RFC2697/4115. Address[17:16] selects color (0: Green, 1: Yellow, 2: Red), Write transfer register contains packet size in bytes, and read transfer register contains meter command color result in bits[1:0].
	set	Set the bit(s) at the specified address according to a bit mask provided in the transfer register(s). A 1 in the bit position of the bit mask signifies that the bit should be set.
	set_imm	Set the bit(s) at the specified address according to a bit mask provided by the immediate data. A 1 in the bit position of the bit mask signifies that the bit should be set. Only operates at a single 32-bit word. See <a href="#">Note 1</a> (with bit 15 zero).
	statistic	Perform a Statistics operation. Add value as 32-bit data to statistic value at address specified.
	statistic_imm	Perform a Statistics operation. Add value as 16-bit immediate data to statistic value at address specified.
	sub	Subtract unsigned 32-bit value(s) in the transfer register(s) from the 32-bit value(s) at the specified address.
	sub_imm	Subtract unsigned 16-bit immediate from the 32-bit value at the specified address. The two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a> .
	sub_imm_sat	Refer to subsat_imm.
	sub_sat	Refer to subsat.
	sub64	Subtract unsigned 64-bit value(s) in the transfer registers from the 64-bit value(s) at the specified address. 64-bit source values are constructed by pairing 32-bit words from transfer registers (least significant word first).
	sub64_imm	Subtract unsigned 16-bit immediate from the 64-bit value at the specified address. The two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a> .
	subsat	Subtract unsigned 32-bit value(s) in the transfer register(s) from the 32-bit value(s) at the specified address. Two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a> . Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	subsat_imm	Subtract unsigned 16-bit immediate from the 32-bit value at the specified address. Two most significant bits of the

Parameter	Command	Description
		immediate indicates sign extension. Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	swap	exchange the value in the transfer register with the value in memory.
	test_add	Add the 32-bit value in the write transfer register to the value in memory, write back to memory and return the pre-modified value to read transfer register.
	test_add64	Add the 64-bit value in the write transfer registers to the 64-bit value in memory, write back to memory and return the pre-modified value to read transfer registers.
	test_add_imm	Add the 5-bit immediate value to the 32-bit value in memory, write back to memory and return the pre-modified value to read transfer registers.
	test_add64_imm	Add the 5-bit immediate value to the value 64-bit data in memory, write back to memory and return the pre-modified value to read transfer registers.
	test_addsat	Add the value in the transfer register to the value in memory and return the pre-modified value. Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	test_addsat_imm	Add the 5-bit immediate to the value in memory and return the premodified value to the transfer register. Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	test_and_add_imm_sat	Refer to test_addsat_imm.
	test_and_add_sat	Refer to test_addsat.
	test_and_clr	Refer to test_clr.
	test_and_clr_imm	Refer to test_clr_imm.
	test_and_compare_write	Refer to test_compare_write.
	test_and_set	Refer to test_set.
	test_and_set_imm	Refer to test_set_imm.
	test_and_sub_imm_sat	Refer to test_subsat_imm.
	test_and_sub_sat	Refer to test_subsat.
	test_clr	Same as clr command but also returns the premodified memory contents.
	test_clr_imm	Same as clr_imm command but also returns premodified memory contents.
	test_compare_write	Read memory and compare (word-by-word) the indicated bytes with the transfer register (pulled data); if all the indicated bytes match, then replace the memory word with the transfer register (word = 32 bits). The mask value indicates which bytes must match for the memory to be

Parameter	Command	Description
		<p>overwritten by the transfer register data. Byte mask bits must not be equal to 0x0 or 0xF. Byte mask is specified using an indirect reference, where each bit in the mask corresponds to a byte in the 32-bit word. The push-back of the pre-modified data in memory is unconditional.</p> <ul style="list-style-type: none"> <li>• mask bit:</li> </ul> <ul style="list-style-type: none"> <li>• 0: Memory contents does not need to match.</li> <li>• 1: Memory contents needs to match the corresponding byte of transfer register data.</li> </ul> <p>Note: For data to be written at least one byte in the 32-bit word in transfer register needs to match the data in memory. If a zero byte mask is used it will only return the premodified memory contents and no data will be written.</p>
	test_set	Read memory, OR with supplied data, and write back to specified memory location. Return the premodified memory value.
	test_set_imm	Set the bit(s) at the specified address (according to a bit mask provided by the immediate data) and return the premodified value. A 1 in the bit position of the bit mask signifies that the bit should be set. Only operates at a single 32-bit word. See <a href="#">Note 1</a> (with bit 15 zero).
	test_sub	Subtract the 32-bit value in the specified transfer register from the specified memory location, and return the premodified memory value.
	test_sub64	Subtract the 64-bit value in the specified transfer register from the specified memory location, and return the premodified memory value.
	test_sub_imm	Subtract the 5-bit value in the specified transfer register from the specified memory location, and return the premodified memory value.
	test_sub64_imm	Subtract the 5-bit value in the specified transfer register from the specified memory location, and return the premodified memory value.
	test_subsat	Subtract the 32-bit value in the specified transfer register from the specified memory location, and return the premodified memory value. Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.
	test_subsat_imm	Subtract 5-bit immediate from the 32-bit value at the specified address. Two most significant bits of the immediate indicates sign extension. See <a href="#">Note 1</a> . Saturation limits apply, refer to <a href="#">Note 2</a> for saturation limits definition.

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
	xor	Perform bitwise exclusive-or of the 32-bit value(s) in the transfer registers to the 32-bit value(s) at the specified address.
xfer	<p><i>Logical and arithmetic non-immediate instructions without readback:</i></p> <p>add, add64, add_sat, addsat, clr, set, statistic, sub, sub64, sub_sat, subsat</p> <p><i>Logical and arithmetic non-immediate instructions with readback:</i></p> <p>meter swap, test_add, test_add64, test_addsat, test_and_add_sat, test_and_clr, test_and_compare_write, test_and_set, test_and_sub_sat, test_clr, test_compare_write, test_set, test_sub, test_sub64, test_subsat, xor</p> <p><i>All immediate instruction without readback:</i></p> <p>add64_imm, add_imm, add_imm_sat, addsat_imm, clr_imm, set_imm, statistic_imm, sub64_imm, sub_imm, sub_imm_sat, subsat_imm</p> <p><i>Arithmetic and Relational immediate instructions with readback:</i></p> <p>test_add_imm, test_add64_imm, test_addsat_imm, test_and_add_imm_sat, test_and_clr_imm, test_and_set_imm, test_and_sub_imm_sat, test_clr_imm, test_set_imm, test_sub64_imm,</p>	Write transfer registers hold data modifiers.  Write transfer registers hold data modifiers and transfer read registers hold the premodified data.  Must be omitted; should be “--”.  Transfer read registers to hold the premodified data.

Parameter	Command	Description
	test_sub_imm, test_subsat_imm	
	decr, decr64, incr, incr64	Must be omitted; should be “--”.
src_op1, src_op2	<p><i>Non-Immediate instructions:</i></p> add, add_sat, addsat, clr, cmp_rw, meter, set, statistic, sub, sub_sat, subsat, swap, test_add, test_addsat, test_and_add_sat, test_and_clr, test_and_set, test_and_sub_sat, test_clr, test_set, test_sub, test_subsat, xor	Restricted source operands that construct base address to a preferred 40-bit I/O target. SRAM address specified in command address[15:2] where address[2:0] is 2b00. <a href="#">Section 2.1.6.2</a>
	<p><i>Immediate instructions:</i></p> add_imm, add_imm_sat, addsat_imm, clr_imm, set_imm, statistic_imm, sub_imm, sub_imm_sat, subsat_imm, test_add_imm, test_addsat_imm, test_and_add_imm_sat, test_and_clr_imm, test_and_compare_write, test_and_set_imm, test_and_sub_imm_sat, test_clr_imm, test_compare_write, test_set_imm, test_sub_imm, test_subsat_imm	Restricted source operands that construct base address to a preferred 40-bit I/O target. SRAM address specified in command address[15:2] where address[2:0] is 2b00. <a href="#">Section 2.1.6.2</a>
	add64, add64_imm, incr, incr64, decr, decr64, sub64, sub64_imm, test_add64, test_add64_imm, test_sub64, test_sub64_imm	Restricted source operands that construct base address to a preferred 40-bit I/O target. The address must be 64-bit aligned. SRAM address specified in command address[15:3] where address[3:0] is 3b00.
ref_cnt	add, add_sat, addsat, clr, cmp_rw, sub, sub_sat, subsat, swap, test_add, test_add64, test_addsat, test_and_add_sat, test_and_compare_write, test_and_sub_sat, test_compare_write, test_sub, test_sub64, test_subsat	Reference count in 4-byte words. Valid values are 1-32. Values above 8 must be specified using indirect reference.

Parameter	Command	Description
	set, test_and_clr, test_and_set, test_clr, test_set, xor	Reference count in 4-byte words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
	add64, sub64, test_add64	Reference count in 4-byte words. Valid values are 2, 4, 6, 8, 10, 12, 14 and 16. Values above 8 must be specified using indirect reference.
	<i>Arithmetic and Relational immediate instructions without readback:</i>  add_imm, add64_imm, add_imm_sat, add_sat_imm, clr_imm, set_imm, sub_imm, sub64_imm, sub_imm_sat, subsat_imm	16-bit immediate data value for operation. Values from 1 to 7 can be specified directly in the instruction with ref_cnt. An indirect reference must be used for specifying a full 16-bit value. Note: ref_cnt is directly applied to the command operation. See <a href="#">Note 3</a> .
	<i>Arithmetic and Relational immediate instructions with readback:</i>  test_add_imm, test_add64_imm, test_addsat_imm, test_and_add_imm_sat, test_and_clr_imm, test_and_set_imm, test_and_sub_imm_sat, test_clr_imm, test_set_imm, test_sub64_imm, test_sub_imm, test_subsat_imm	5-bit immediate data value for operation. Values from 1 to 7 can be specified directly in the instruction with ref_cnt. An indirect reference must be used for specifying a full 5-bit value. Note: ref_cnt is directly applied to the command operation. See <a href="#">Note 4</a> .
opt Tok	set_imm, clr_imm, add_imm, add64_imm, sub_imm, sub64_imm	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
	incr, incr64, decr, decr64	Must be omitted; should be “--”.
	All other commands	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> . ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> . indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> . ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

#### Notes:

The I/O instructions support an indirect reference optional token. When the indirect\_ref optional token is specified, the output of the ALU from the previous instruction is used to modify one or more parameters within

an instruction. The format of the indirect data is specified in the instruction definition. All Reserved fields (noted as RES) must be set to 0 otherwise unpredictable behavior may result.

1. Add/subtract/set/clr operations with immediate can optionally sign extend the immediate value. Only the 14 least significant bits of the 16-bit immediate is used for the actual value. The remaining two most significant bits indicate sign extension. A value of:
  - 00 indicates no sign extension
  - 01 indicates sign extend to 32-bit/64-bit value for 32/64-bit operation respectively
  - 10 indicates no sign extension, but duplicate immediate in both high and low 32-bit words (applicable to 64-bit operations only)
  - 11 means sign extend to 32-bit word and duplicate value in both high and low 32-bit words (applicable to 64-bit operations only).
2. Any instruction that has sat keyword in the instruction performs arithmetic operation and restricts the computation to the limit of the target storage value to (32 or 64 bits).

For example: `cls[add_sat, $xfer[0], *l$index0, 0x0, 1];`

Where `$xfer[0] = 0x0010` and address `*l$index0` contains `0xFFFF3`. The instruction will add `0x0010` to `0xFFFF3` and will not wrap to `0x0003`, but will saturate to `0xFFFF`. If instruction was subtract `0x0010` from `0x0005` the result will not wrap to `0xFFFF5`, but will saturate to `0x0000`.

32-bit arithmetic operations - `0xFFFF FFFF FFFF FFF0 + 0x10` results in `0xFFFF FFFF FFFF`. And `0x0000 000F - 0x10` results in `0x0000 0000`.

64-bit arithmetic operations - `0xFFFF FFFF FFFF FFFF FFF0 + 0x10` results in `0xFFFF FFFF FFFF FFFF`. And `0x0000 0000 0000 000F - 0x10` results in `0x0000 0000 0000 0000`.

3. For immediate atomic operations, the immediate data is specified by the `ref_cnt` parameter (immediate atomics always operate on a single word, so the length of the operation is implicit). Values in the range 1-7 can be encoded directly in the instruction. Values in the range 1-31 can be specified by overriding the CPP command length via an indirect reference.

Specifying a `ref_cnt` of 0 is special. It indicates that a 16-bit immediate data value is constructed using the data master and data reference fields of the CPP transaction. The lower 14 bits are taken from the data reference and the upper 2 bits are taken from the least significant bits of the data master.

The recommended V2 Indirect Reference mode for specifying the immediate data is to use `PREV_ALU_OVE_DATA = 2` and providing the data in `PREV_ALU DATA16` field. Refer to section [Section 2.1.6.4.6.2](#) for further details.

When using V1 indirect reference formats the immediate data can be specified using the following indirect\_ref format.

Override immediate using data master and data reference																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	Reserved				Immediate data																Reserved							

4. For immediate atomic operations, the immediate data is specified by the ref\_cnt parameter (immediate atomics always operate on a single word, so the length of the operation is implicit). Values in the range 1-7 can be encoded directly in the instruction. Values in the range 1-31 can be specified by overriding the CPP command length via an indirect reference.

Note that immediate atomic operations requiring result data (i.e., "test\_and\_" commands) can not use the override for data master/data reference to specify the immediate value as these are required for specifying the read transfer registers for the results. Therefore they are limited to using the 'length' field for immediate data.

### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.37.2 CLS (CAM Lookup)

Treat area of Cluster Local Scratch SRAM as Content Addressable Memory and perform lookup on it.

### Instruction Format

cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
--

### Parameter Descriptions

Parameter	Command	Description
cmd	cam_lookup32	Fetch 32-bit values from SRAM and compare with 32-bit input data.
	cam_lookup32_add	Fetch 32-bit values from SRAM and compare with 32-bit input data. If no match was found insert input data into first zero entry in the CAM.
	cam_lookup24	Fetch 32-bit values from SRAM and compare lower 24-bits with input data.
	cam_lookup24_add	Fetch 32-bit values from SRAM and compare lower 24-bits with input data. If no match was found insert input data (full 32-bit) into first zero entry in the CAM.
	cam_lookup24_add_extend	Fetch 32-bit values from SRAM and compare lower 24-bits with input data. If match is found then bit[31] of the matching memory data location is set indicating that the entry is now locked. The pre-modified memory data top byte is returned in the push data[15:8] so that the software can learn if location was already locked. If match is not found and there is room for the entry then new entry is added with data[31] set indicating that the entry is now locked. If match is not found and there is no free entry then set bit[31] of the first memory location used for the CAM command. It will also push back the previous value of bit[31] of the first memory location in bit[15] of the

Parameter	Command	Description
		push data (which is normally zero) along with 0xff in push data[7:0].
	cam_lookup24_add_lock	Fetch 32-bit values from SRAM and compare lower 24-bits with input data. If match is found then bit[31] of the matching memory data location is set indicating that the entry is now locked. The pre-modified memory data top byte is returned in the push data[15:8] so that the software can learn if location was already locked. If match is not found and there is room for the entry then new entry is added with data[31] set indicating that the entry is now locked. If match is not found and there is no free entry then no memory update is performed and (0x000000ff) is returned in push data.
	cam_lookup24_add_inc	Fetch 32-bit values from SRAM and compare lower 24-bits with input data. If match was found increment the counter in the top 8-bits of the matching entry with one (without saturation). If no match was found insert input data (full 32-bit) into first zero entry in the CAM.
	cam_lookup16	Fetch 16-bit values from SRAM and compare with lower 16-bits of input data.
	cam_lookup16_add	Fetch 16-bit values from SRAM and compare with lower 16-bits of input data. If no match was found insert 16-bit input data into first zero entry in the CAM.
	cam_lookup8	Fetch 8-bit values from SRAM and compare with lower 8-bits of input data.
	cam_lookup8_add	Fetch 8-bit values from SRAM and compare with lower 8-bits of input data. If no match was found insert 8-bit input data into first zero entry in the CAM.
xfer	cam_lookup32, cam_lookup32_add, cam_lookup24, cam_lookup24_add, cam_lookup24_add_extend, cam_lookup24_add_inc, cam_lookup24_add_lock	One or two write transfer registers containing input data. One read transfer register containing result. See <a href="#">Note 1</a> .
	cam_lookup16, cam_lookup16_add, cam_lookup8, cam_lookup8_add	One write transfer register containing input data. Two read transfer registers containing result. See <a href="#">Note 2</a> .
src_op1, src_op2	All commands	Restricted source operands that constructs base address to CAM memory. Address[25:0] holds the CAM memory address where address[2:0] must be zero. Refer to <a href="#">Section 2.1.6.2</a>

Parameter	Command	Description
ref_cnt	cam_lookup32, cam_lookup24, cam_lookup16, cam_lookup8	Size of CAM in 64-bit words. Valid values are 1-32. Values above 8 must be specified using indirect reference.
	cam_lookup32_add, cam_lookup24_add, cam_lookup16_add, cam_lookup8_add, cam_lookup24_add_extend, cam_lookup_add_lock	Size of CAM in 64-bit words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
	cam_lookup24_add_inc	Size of CAM in 64-bit words + 16. Valid values are 1-16 (+ 16). Value must be specified using indirect reference.
opt_tok	All commands	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

**Notes:**

1. 32-bit and 24-bit CAM lookups have a single 32-bit result delivered in a read transfer register:

Result of 32-bit and 24-bit CAM lookups																																
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Match mask																Entry bits								Miss	Offset							

Bits	Bit width	Field Name	Description
31:16	16	Match mask	Bitmask of the last 24-bit or 32-bit word match, where bit[16] is the first CAM entry, bit[17] is the second and so on to bit[31].
15:8	8	Entry bits	Bits [31:24] of CAM matching entry. For 24-bit lookups this provides for 8-bit of client data in the CAM entry. For cam_lookup24_add_inc this gives the 8-bit count value associated with the CAM entry. On CAM match CAM entry Bits

Bits	Bit width	Field Name	Description
			[31:24] is stored in read transfer register before CAM entry Bits [31:24] is incremented.
7	1	Miss	Set if CAM match did not occur. Cleared if match occurred.
6:0	7	Offset	If match occurred, offset to the CAM entry which matched. If match did not occur, but insert was requested and successful, offset to the CAM entry which was inserted. If match did not occur, and no entry was inserted, the value 0x7F.

2. 16-bit and 8-bit CAM lookups have a 64-bit result delivered in two read transfer registers (least significant word first):

Result of 16-bit and 8-bit CAM lookups																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Match mask (hi)																															

Result of 16-bit and 8-bit CAM lookups																																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Match mask (lo)																															Miss	Offset	

Bits	Bit width	Field Name	Description
63:8	56	Match mask	Bitmask of the last $n$ byte comparisons, with bit [63] always indicating the result of the last byte comparison and the following bits indicating previous comparisons. For example, with a CAM lookup of six 64-bit words, bit [16] indicates result of first byte comparison, bit [17]

Bits	Bit width	Field Name	Description
			indicates result of next comparison, etc.
7	1	Miss	Set if CAM match did not occur. Cleared if match occurred.
6:0	7	Offset	If match occurred, offset to the CAM entry which matched. If match did not occur, but insert was requested and successful, offset to the CAM entry which was inserted. If match did not occur, and no entry was inserted, the value 0x7F.

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Example of `cls[cam_lookup24]`

The CLS CAM supports various sizes of input match data and CAM table sizes. CAM table resides Cluster Local Scratch SRAM memory. NFP-6xxx memory architecture provides hardware acceleration engines for CAM operations.

CAM table requires pre-initializing all table entries to zero, and search for CAM entry of zero is invalid. Lower 24 bits of CAM memory must be initialized to zero because these are the CAM compare bits.

The CAM operations are inherently atomic, so multiple FPC threads may perform CAM operations - and they will happen atomically. No external atomic mechanism is needed.

On a CAM match the CAM entry index is returned and can be used as an offset into another table that contains the actual application data structure. One use case is to use a CAM table to front-end a hash table. For every CAM index there is a corresponding hash bucket into a hash table. One can, for instance, place 32-bits of hash value of a certain tuple-key as CAM value and look it up at runtime. If matched, the index number of matched CAM entry can be used to access corresponding hash table entry.

Managing the CAM table is up to the application. One use case is when CAM entries are static and worker thread adds and deletes CAM entries with `cls[cam_lookup24_add_inc]` and `cls[clr]`. When each lookup operation results in a match, then the upper 8 bits in CLS CAM memory is incremented by one. This can be used as LRU count and when CAM is full and need to add entry this count value can determine which CAM entry could be removed. When CAM memory location is zero (available location) then if CAM lookup results in a miss then the match data (pull data) is added.

In order to remove old/stale entries from the CAM memory, one will need to write zero atomically to CAM entry of interest. So any `cls`-atomic operation can be used (`cls[clr]` for example). Depending on the application, there

can be different reasons to purge entries in CAM; for example to make room for a new entry in an already full CAM, one can adopt Least Recently Used algorithm where timestamp is maintained in a separate data structure and may be used to find the LRU and purge it. In other applications, an entry may be explicitly deleted due to certain delete event.

The `cls[cam_lookup24_add_inc]` operation has nice feature where if lookup of a CAM tag fails, it is added, provided there is space. If there is no space, CAM result returns `0xFF` which then can be used by the ME thread in operation to purge an LRU and then add its tag to CAM table.

#### Example 1. Atomic CAM lookup on CLS CAM of sixteen 24 bit entries.

With `cls[cam_lookup24_add_inc, $xfer0, clsAdr, 0, 8]` command the CAM entries total eight 32-bit entries, where lower 24 bits are compared. This command accepts a 24-bit match input value loaded in `$xfer0[31:0]` register. The lookup engine pulls data from transfer register and performs atomic lookup on the CAM memory lower 24 bits. If no match exists and one or more CAM entries are unused (value is `0x000000`) then the command match value (lower 24 bits) is written to the first available entry in the CAM table.

The `cls[cam_lookup24_add_inc]` example below demonstrates the use case where a worker thread performs CAM Table management using LRU algorithm on CAM table in Cluster Local Scratch Memory `0x8000_4000`. The code will handle three cases:

- Match
- No match and add to CAM Table
- No match, apply LRU and add to CAM Table

If the CAM table is full one can adopt Least Recently Used (LRU) algorithm where timestamp is maintained in separate data structure and is used to find the LRU and purge it with `cls[clr]` command. Or an entry may be explicitly deleted due to certain delete event. None the less the CAM table is managed in an atomic fashion because all CAM operations are inherently atomic.

The LRU algorithm code is not shown below but has a place holder in the code for showing the three logical use cases. It is shown by `lru()` macro below.

```
// Here a worker thread has received a packet and its job is to perform CAM lookups and
// manage CAM table.

// A tag byte from a packet is used as search key in CAM table
// The CAM table is pre-initialized with value of 0x0 and is not shown.
// CAM Table is arranged in CLS memory at starting address 0x80004000
// The number of 32 bit entries for this example is 16.
// Each CAM entry's upper 8 bits will hold the LRU count value that will atomically
// increment by the lookup operation when a match occurs.
// The read XFER register $xfer[31:16] will contain the bit map of the
// match entry. First entry is $xfer[16], second entry is $xfer[17], ... sixteenth entry is
// $xfer[31]

.sig s1
.reg r_mem_addr_ulw
.reg r_mem_addr_llw
.reg r_camAddr_llw
.reg r_cam_entry
.reg tag
```

```

.reg r_byteMask
.reg $xrefClr0 $xrefClr1 $xrefClr2 $xrefClr3
.set $xrefClr0 $xrefClr1 $xrefClr2 $xrefClr3
.xfer_order $xrefClr0 $xrefClr1 $xrefClr2 $xrefClr3
.reg $xref0 $xref1 $xref2 $xref3
.set $xref0 $xref1 $xref2 $xref3
.xfer_order $xref0 $xref1 $xref2 $xref3

r_mem_addr_ulw = 0x80000000
r_mem_addr_llw = 0x4000

r_byteMask = 0x00000000
r_camAddr_llw = 0x4000

// CAM search on tag byte from packet received
CAM_LOOKUP#:

;Load tag byte into transfer register that will be used in lookup instruction.
alu[r_cam_entry, --, B, tag]
alu[$xref0, --, B, r_cam_entry]
; perform lookup on CAM table in CLS address 0x80004000
cls[cam_lookup24_add_inc, $xref0, r_mem_addr_ulw, r_mem_addr_llw], sig_done[s1]
ctx_arb[s1]

; Check read transfer register for result of CAM lookup.
; If return value is 0xff then no match is found and no free entries in CAM.
; Delete CAM entry determined by LRU algorithm - write zero to CAM entry (CLS memory)
; If return value is 0xy, where y is between 0 and f (for up to 16 CAM entries)
; then lookup did not find match but was added to CAM Table and y is CAM entry index.
; If return value is not 0xff then match is found in CAM table.
; Return value is index where match was found,
; first entry is 0x00, second entry is 0x04, third entry is 0x08

alu[--, $xref0, -, 0xff]
bne[MATCH_OR_CAM_ADD#]
; No match found and no entry in CAM Table, make room using LRU algorithm.
; LRU algorithm determines which CAM entry to remove.
; LRU macro details not shown for this example.
; LRU could use CAM entry upper 8 bits to determine which entry to remove.
; LRU might return a 4-byte mask and CLS memory offset from CAM base address that
; will be used to free CAM entry.
lru(r_byteMask, r_camAddr_llw)
alu[$xrefClr0, --, B, r_byteMask]

; Clear entry in CLS memory, 4-byte mask is loaded in $xrefClr0 transfer register.
cls[clr, $xrefClr0, r_mem_addr_ulw,, r_camAddr_llw], sig_done[s1]
ctx_arb[s1]

; Now add new entry to CAM Table now that CAM entry is free (set to zero in CLS CAM memory)
br[CAM_LOOKUP#]

MATCH_OR_CAM_ADD#:
; check if tag was added to CAM entry and update LRU data structure.
; $xref[7] is set then miss occurred and tag added to CAM.
br_bset[$xref0, 7, CAM_ADD#]

; MATCH only
; $xref0[6:0] contains the entry where MATCH data was found.
; Use $xref0[6:0] (CAM table entry index) for hash table lookup.

```

```

; Each time match is found CAM entry upper 8 bits is incremented by 1 and
; $xref[15:8] will hold the CAM entry upper 8 bits before incremented.
; $xref[31:16] contains the last match entry in bit map format, if 16-bit value
; is preferred for hash table lookup.
; Update LRU table (details not shown here)
br[DONE#]

CAM_ADD#:
; No match and add new tag to CAM table.
; Update LRU table (details not shown here)
; xref0[6:0] indicate offset where entry was added (word increments: 0, 4, 8, etc)
; Use $xref0 in byte lower 7 bits to update hash table with new entry added to CAM table.
DONE#:

```

### 2.2.37.3 CLS (CSR Read and Write)

Move data between FPCs and Cluster Local Scratch CSRs. CSR accesses are to the event manager, autopush logic, ring configurations, hash logic, TRNG, and interrupt manager. All CSR accesses must be 64-bit aligned read/write operations.

#### Instruction Format

<code>cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok</code>
---

#### Parameter Descriptions

Parameter	Command	Description
cmd	read, read_be	Read from cluster local scratch CSR in big endian (LWBE) format.
	read_le	Read from cluster local scratch CSR in little endian format.
	write, write_be	Write to cluster local scratch CSR in big endian (LWBE) format.
	write_le	Write to cluster local scratch CSR in little endian format.
xfer	read, read_be, read_le	Read transfer register.
	write, write_be, write_le	Write transfer register.
src_op1, src_op2	All commands	Restricted source operands that constructs base address to a 32-bit I/O target. <a href="#">Section 2.1.6.2</a>
ref_cnt	All commands	Reference count in 4-byte words (must be 1).
opt_tok	All commands	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.37.4 CLS (Hash Operations)

Create a 64-bit hash index over a set of transfer register values. Each Cluster Local Scratch supports 8 different hash indexes stored in CSRs. The hash index to use is selected using bits [18:16] of the address given in the command. Bits [15:0] of the address are used to specify the location in SRAM of a mask to apply to the transfer registers before calculating the hash index. The mask should be properly initialized by the programmer before invoking the hash command.

For each 64-bit data value (pair of two transfer registers) the data is masked with the corresponding 64-bit mask word fetched from Cluster Local Scratch SRAM, and a new hash\_index is calculated as follows:

$$n = (\text{hash\_index}^{64} + \text{data})$$

$$\text{hash\_index} = (m_{63} \times n^{63} + m_{53} \times n^{53} + m_{36} \times n^{36} + m_4 \times n^4) \bmod (n^{64} + n^{54} + n^{35} + n^{17} + 1)$$

The new 64-bit hash\_index is then stored back into the CSRs. The multiplier values  $m_{63}$ ,  $m_{53}$ ,  $m_{36}$ , and  $m_4$  are taken from the ClsHashMultiply CSR (see Netronome Network Flow Processor NFP-6xxx-xC Databook) and can be used to configure the hash\_index distribution.

### Instruction Format

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	hash_mask	<p>Create a 64-bit hash_index over the transfer registers as follows:</p> <ul style="list-style-type: none"> <li>hash_index_select = address[18:16]</li> <li>hash_mask_address = address[15:0]</li> <li>hash_index = hash_indexes[hash_index_select]</li> <li>For each 64-bit data value in the transfer register pairs:</li> <ul style="list-style-type: none"> <li>data = data AND cls_read64(hash_mask_address)</li> <li>calculate new hash_index as indicated above</li> <li>hash_mask_address = hash_mask_address + 8</li> <li>hash_indexes[hash_index_select] = hash_index</li> </ul> </ul>
	hash_mask_clear	Same as hash_mask command, but also clears the initial value of hash_index before calculating the new value.
xfer	Both commands	Write transfer registers.

Parameter	Command	Description
src_op1, src_op2	Both commands	Restricted source operands for cluster local scratch address. The address is specified by src_op1 + src_op2. The address must be 64-bit aligned. Address bits [15:0] specifies the SRAM data to use for the mask, and address bits [18:16] selects the hash index.
ref_cnt	Both commands	Length of data to produce hash_index over in 32-bit words. Valid values are multiples of 2 in the range 2-32. Values above 8 must be specified using indirect reference.
opt_tok	Both commands	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Example: Create hash of 256-bit value

Firstly, a 32-byte (256-bit) mask needs to be declared and initialized to apply to the data. The mask is initialized with all bits set so that no bits of the data are ignored.

```
; Declare and initialize a 64-bit aligned hash mask
.global_mem hashmask cls 32 8
.init hashmask, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
.init hashmask+16, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
```

Using the initialized hash mask, a 64-bit hash index is then created over the data. The generated hash index can be obtained by reading the CSR corresponding to the selected hash index.

```
.reg addr
.sig hashsig
immed_w0[addr, hashmask]
immed_b2[addr, 3] ; Use hash_index 3

; Declare 256-bits worth of transfer registers. Assume that we somehow
; get our data into these.
.reg $data[8]
.xfer_order $data

; Calculate new hash index, clearing the old one
cls[hash_mask_clear, $data[0], addr, 0, 8], ctx_swap[hashsig]

; Read the new hash_index (ignore the upper 32 bits)
.reg $hash_index
.reg hash_idx_addr
immed_w0[hash_idx_addr, NFP_MECL_HASH_INDEX_LO3]
immed_w1[hash_idx_addr, (NFP_MECL_HASH_INDEX_LO3 >> 16)]
cls[read, $hash_index, hash_idx_addr, 0, 1], ctx_swap[hashsig]
```

## 2.2.37.5 CLS (Lock Queue Operations)

Issue an atomic and exclusive enqueue-and-lock/unlock-and-dequeue operation on a Cluster Local Scratch lock-queue variable. All queue operations must be 64-bit aligned. The lock-queue variable itself must be zeroed before being used as a lock.

There is no transfer data associated with the lock. Instead, a signal pair (sig\_name and sig\_name+1) is used to indicate whether the lock was taken or not. Sending sig\_name is used to indicate that the operation completed. If sig\_name+1 is also sent, it indicates that the operation completed, but failed because the queue was full. Unless the lock operation can be guaranteed to never fail, the programmer should always check sig\_name+1 to verify that the lock operation succeeded.

The lock-queue can hold up to 5 pending signals. Including the initial lock operation on an empty queue, a lock-queue variable can as such support 6 concurrent lock operations without overflowing. All signals are enqueued in FIFO order. When a FPC issues an unlock operation the first signal in the queue is dequeued.

Note that the lock-queue only stores 11 bits of signal information (7 bits for the signal\_ref, and 4 bits for signal\_master) per entry. This implies that the CLS queue operations can only signal FPC threads within the current cluster, even if an indirect\_ref is used to override the signal\_master for the queue\_lock operation.

### Instruction Format

```
cls[cmd, xfer, src_op1, src_op2], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	queue_lock	Attempt to acquire lock and respond with sig_name when lock is acquired. If lock can not be immediately acquired, enqueue the signal to send once it can be acquired. Up to 5 additional signals can be enqueued. If queue is already full, respond with sig_name and sig_name+1.
	queue_unlock	Release an already acquired lock. If the lock-queue has any pending signals enqueued do not release the lock, but instead dequeue the first signal off the queue and trigger it. If there are no pending signals in the queue, then no signals are triggered and the queue is marked as empty.
xfer	queue_lock, queue_unlock	Must always be “--”.
src_op1, src_op2	Both commands	Restricted source operands for cluster local scratch address. The address is specified by src_op1 + src_op2. The address must be 64-bit aligned.
opt Tok	queue_lock	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .
	queue_unlock	None.

## Condition Codes Affected

N	Z	V	C
Not Affected			

### Example: queue\_lock/queue\_unlock

Firstly, a 64-bit variable needs to be declared in cluster local scratch which is aligned at 64-bits and has initial value zero.

```
; Declare and initialize a 64-bit aligned lock variable
.global_mem lockvar cls 8 8
.init lockvar, 0, 0
```

Using the initialized lock variable, a thread can then perform lock/unlock operations using the appropriate cls command.

```
.reg addr
.sig locksig
.immed[addr, lockvar]

; Acquire lock and check for overflow
cls[queue_lock, --, addr, 0], sig_done[locksig]
ctx_arb[locksig], any
br_signal[locksig+1, error#]

; Perform operations requiring lock to be held

; Release lock
cls[queue_unlock, addr, 0]

error#:
; Perform error handling
```

## 2.2.37.6 CLS (Reflect)

Issue a reflect operation via Cluster Local Scratch to access transfer registers of other FPCs in the current cluster. A reflect operation copies write transfer registers from the current FPC to the read transfer registers of a remote FPC, or write transfer registers from a remote FPC to the read transfer registers of the current FPC.

CLS reflect instructions only applies to FPC Transfer registers.

The remote FPC and its transfer registers are specified via other\_data\_master and other\_data\_ref fields encoded in the command address. Address bits [15:12] encode the other\_data\_master and address bits [11:0] encode the other\_data\_ref. Cluster Local Scratch will trigger a signal to the source and/or destination when the transfer registers have been read or written. The signal reference to use is either specified in the command instruction itself with opt\_tok, or if address[31] is set, signal is encoded in address[30:24].

In summary, the address fields is defined below:

- address[39:34] = Island# (40-bit address mode) or 6b0 (32-bit address mode)

- address[31] = if 1 use other signal\_ref (address[30:24]) or if 0 use the signal\_ref given in the command for the signal to the other master.
- address[30:24] = other\_signal\_ref (a signal reference for the other master; only used if address[31] is True)
- address[15:12] = other\_data\_master (ID within the cluster of the other master for data), where other\_data\_master is FPC + 4.
- address[11:0] = other\_data\_ref (data reference for the other master)

When data\_master references another FPC, the address encoding is best described as:

40-bit Address encoding for CLS Reflect Operations							
39:34	33:32	31	30:28	27:24	23:16	15:12	11:0
Island#	2b0	S	Signal CTX	Signal number	8b0	other_data_master	other_data_ref

### Instruction Format

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok // 32-bit address encode
cls[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt_tok // 40-bit address encode
cls[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt_tok // 40-bit address encode
```

### Parameter Descriptions

Parameter	Command	Description
cmd	reflect_from_sig_both	Refer to reflect_write_sig_both.
	reflect_from_sig_dst	Refer to reflect_write_sig_remote.
	reflect_from_sig_src	Refer to reflect_write_sig_local.
	reflect_read_sig_both	Read data from the other_data_master/other_data_ref (remote FPC) as specified in the address and write it to the local FPC Transfer In registers. Signal remote FPC after data_ref is read and local FPC after xfer_registers are written.
	reflect_read_sig_local	Read data from the other_data_master/other_data_ref (remote FPC) as specified in the address and write it to the local FPC Transfer In registers. Signal local FPC (destination) after xfer_registers are written.
	reflect_read_sig_remote	Read data from the other_data_master/other_data_ref (remote FPC) as specified in the address and write it to the local FPC Transfer In registers. Signal remote FPC (source) after data_ref is read.
	reflect_to_sig_both	Refer to reflect_read_sig_both.
	reflect_to_sig_dst	Refer to reflect_read_sig_local.
	reflect_to_sig_src	Refer to reflect_read_sig_remote.
	reflect_write_sig_both	Read data from the local FPC Transfer Out registers and write to the other_data_master/other_data_ref (remote FPC)

Parameter	Command	Description
		as specified in the address. Signal local FPC after transfer registers are read and remote FPC after data_ref is written.
	reflect_write_sig_local	Read data from the local Transfer Out registers and write to the other_data_master/other_data_ref (remote FPC) as specified in the address. Signal local FPC (source) after transfer registers are read.
	reflect_write_sig_remote	Read data from the local FPC Transfer Out registers and write to the other_data_master/other_data_ref (remote FPC) as specified in the address. Signal remote FPC (destination) after data_ref is written.
xfer	reflect_read_sig_both, reflect_read_sig_local, reflect_read_sig_remote, reflect_to_sig_both, reflect_to_sig_dst, reflect_to_sig_src	Read transfer registers.
	reflect_from_sig_both, reflect_from_sig_dst, reflect_from_sig_src, reflect_write_sig_both, reflect_write_sig_local, reflect_write_sig_remote	Write transfer registers.
src_op1, src_op2	All commands	Restricted source operands for cluster local scratch address. The 32-bit address is specified by src_op1 + src_op2 with address encoding as described above.
ref_cnt	All commands	Reference count in 4-byte words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
opt_tok		sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Examples of cls[reflect]

The cluster local scratch implements reflect operations only for masters within a cluster. The number of 32-bit words to reflect is specified in the length field of the CPP command, and the direction (reading or writing the target master) is given by the command itself. There are three signaling options: signal after the pull, signal after

the push, or signal after both. The reflect operations may be initiated by any master with a simple command address decode.

At a minimum address[15:0] must be specified. The bottom 12 bits of address indicate the 12-bit data reference (other\_data\_ref) of the reflection, and the top 4 bits indicate the data master (other\_data\_master within the cluster) for the reflection. Also, if bit [31] of the address is set then bits [30:24] are used as the signal reference (other\_signal\_ref) for the reflection.

A typical usage of the Reflector mode is for an FPC to read or write the Transfer registers of another FPC within the same cluster (since this is CLS command). Other usages include a Master reading FPC mailbox CSRs, since they also can be read through the pull bus.

The following examples will show various ways to use the `cls[reflect_to_xxx]` commands to access other FPC Transfer registers in 32-bit and 40-bit address encode mode. The command type will determine who gets a signal when action is completed.

The various examples below show how FPC 4 can get data from FPC 5 Xfer-out registers and control the signaling on transfer complete. Each ME island's Target15AddressModeConfig CSR register is by default set to recommended 40-bit address mode and setup of this CSR is not shown in examples below.

#### **Example 1. `cls[reflect_read_sig_local]`**

`cls[reflect_read_sig_local]` instruction will copy data from the remote FPC xfer out registers to the local FPC xfer in registers. Once the data is written to local FPC then FPC 4 will receive a signal when data push is completed.

Below \$xfer0 indicates where to push the first word pulled from FPC 5 xfer register. The command address[15:12] field contains the remote FPC (other\_data\_master) to read and is set to 5. The remote\_xfer register is loaded into address[11:0] and holds the starting remote FPC 5 xfer register to read (In this case it is set to 0 for xfer0). The ref\_cnt (5th argument) in `cls[reflect_read_sig_local]` command indicates how many 32-bit words to transfer. Here remote FPC 5 \$xfer0 out register is pulled and pushed to FPC 4 xfer0 in register. FPC 4 is signaled with sig2 when push to FPC 4 xfer0 in register is completed.

```

; -----
; FPC 4
; -----
.sig sig1 sig2
.reg address
.reg remote_xfer
.reg ref_cnt
.reg read write $xfer0 $xfer1 $xfer2 $xfer3
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3
.set $xfer0 $xfer1 $xfer2 $xfer3

alu[address, --, B, 0x5, <<12] ; other_data_master = FPC 5
alu_shf[remote_xfer, --, B, 0] ; Used to select starting remote xfer register, $xfer0
alu_shf[ref_cnt, --, B, 1] ; Used to set number of xfer registers to pull, 1 word
cls[reflect_read_sig_local, $xfer0, address, remote_xfer, ref_cnt], ctx_swap[sig2]

```

#### **Example 2. `cls[reflect_read_sig_both]`**

`cls[reflect_read_sig_both]` instruction will copy data from remote FPC xfer out registers to the local FPC xfer In registers. Once the data is pulled from remote FPC then signal the remote FPC and signal the local FPC on push completion.

Below \$xfer2 indicates start xfer register where to push two words from remote FPC 5 xfer registers. Address[15:12] contains the remote FPC and is set to 5. The remote\_xfer register holds the remote start xfer register to pull (set to 2) and ref\_cnt (5th argument) in the command says to pull two words from remote FPC xfer 2 and 3 registers and pushes the words to local FPC xfer 2 and 3 registers. FPC 4 and FPC 5 are signaled with signal 4 when second push operation completes to FPC 4.

```
; -----
; FPC 4
; -----

.sig sig4
.reg address
.reg remote_xfer remote_me
.reg ref_cnt
.reg read write $xfer0 $xfer1 $xfer2 $xfer3
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3
.set $xfer0 $xfer1 $xfer2 $xfer3

immed[remote_me, 0x5] ; other_data_master = FPC 5
alu[address, --, B, remote_me, <<12] ; Remote FPC 5 encoded in address[15:12]
alu_shf[remote_xfer, --, B, 2] ; Used to select starting remote xfer register, $xfer2
alu_shf[ref_cnt, --, B, 2] ; Used to select number of xfer registers to pull, 2 words
cls[reflect_read_sig_both, $xfer2, address, remote_xfer, ref_cnt], ctx_swap[sig4]
```

### **Example 3. cls[reflect\_read\_sig\_remote]**

cls[reflect\_read\_sig\_remote] instruction will copy data from remote FPC xfer out registers to the local FPC xfer in registers. Once the data is pulled from remote FPC xfer out registers, signal remote FPC.

Below \$xfer0 indicates start xfer register where to push four words from remote FPC 5 xfer registers. The command address[15:12] field contains the remote FPC (other\_data\_master) to read and is set to 5. The remote\_xfer register contains the starting xfer register to pull from (set to xfer4 register). The ref\_cnt register (5th argument) in cls[reflect\_read\_sig\_remote] command below indicates how many words to pull. In this example remote FPC 5 \$xfer4-\$xfer7 registers are pulled and pushed into FPC 4 \$xfer0-\$xfer3 registers. FPC 5 is signalled with signal 6 when four words are pulled from the reflect command.

```
; -----
; FPC 4
; -----

.sig sig6
.reg address
.reg remote_xfer
.reg ref_cnt
.reg read write $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5 $xfer6 $xfer7
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5 $xfer6 $xfer7
.set $xfer0 $xfer1 $xfer2 $xfer3 $xfer4 $xfer5 $xfer6 $xfer7

alu[address, --, B, 0x5, <<12] ; other_data_master = FPC 5
alu_shf[remote_xfer, --, B, 4] ; Used to select starting remote xfer register, $xfer4
alu_shf[ref_cnt, --, B, 4] ; Used to set number of xfer registers to pull, 4 words
cls[reflect_read_sig_remote, $xfer0, address, remote_xfer, ref_cnt], sig_done[sig6]
```

### **Example 4. cls[reflect\_read\_sig\_both]**

Next example shows how cls[reflect\_read\_sig\_both] command address specifies the signal reference and opt Tok to signal both the remote and local FPC in 40-bit addressing encoding.

Below \$xfer4 indicates where to push the single word from remote FPC 5 \$xfer2. address2[15:12] contains the remote FPC and is 5. Because address2[31] is set, signal 11 is specified in address2[27:24], and address[30:29] contains the Signal CTX and is set to 0. The remote\_xfer register contains the remote start xfer register to pull from (set to xfer2 register). The ref\_cnt register holding how many pulls to perform is the 5th argument in cls[reflect\_read\_sig\_both] and is set to 1. One word will be pulled from remote FPC xfer 2 out register and will be pushed to the local FPC xfer 4 in register. FPC 5 is signaled with signal 5 and signal 11 on pull completion. FPC 4 is signaled with signal 5 when one word is pushed from the reflect command.

```

; -----
; FPC 4
; -----

.sig sig1 sig5
.reg address
.reg address2
.reg remote_xfer
.reg remote_me
.reg ref_cnt
.reg cl_num ; Island#
.reg read write $xfer0 $xfer1 $xfer2 $xfer3
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3
.set $xfer0 $xfer1 $xfer2 $xfer3

immed[remote_me, 5]
alu[address, --, B, remote_me, <<12] ; other_data_master = FPC 5

alu[address2, address, OR, 1, <<31] ; Use signal_ref specified in address2[30:24]
alu[address2, address2, OR, 11, <<24] ; Use Signal num 11, specified in address[27:24],
                                         ; and Signal CTX = 0
alu_shf[remote_xfer, --, B, 2]          ; remote $xfer 2 out register
alu[address2, address2, OR, remote_xfer]

; For 40-bit addressing : cl_num (island#) is specified in command's address[39:34] field.
; Refer to Table Local Scratch Recommended Addressing Mode" in NFP6xxx Databook.
immed[cl_num, --, B, 0x20] ; Used to set Island# in cls[] command address[39:34] below.
alu_shf[cl_num_shift, --, B, cl_num, <<26] ; left shift 26 bits first
alu_shf[ref_cnt, --, B, 1] ; Set ref_cnt to how many words to pull, 1 word

; Signal SRC only, where the other FPC (SRC) should get signal 11
; 40-bit addressing: place cl_num (island#) of 0x20 in address bits [39:34]
; cls[reflect_read_sig_both] will signal remote and local FPC with different signal numbers.
cls[reflect_read_sig_both, $xfer4, address2, cl_num_shift, <<8, ref_cnt], sig_done[sig5]

```

## 2.2.37.7 CLS (Ring Operations)

Add entries to or remove entries from a ring in Cluster Local Scratch SRAM. Each Cluster Local Scratch supports 16 rings. The size and location of a ring is configured via CSRs. Entries are added to or removed from a ring in quantities of 32-bit words.

Rings may be used as either a FIFO queue (add entries to tail, remove entries from head), or a LIFO stack (add entries to tail, remove entries from tail). There is also a mode to add entries at an offset beyond the current tail pointer without updating the tail, and to later update the tail pointer without adding any entries. This can be used to, e.g., reorder the packets in a queue.

A ring can be configured to generate events on the event bus when it: underflows, overflows, becomes not empty, or becomes not full (not full meaning less than 3/4 full). No explicit signal will be sent back to the FPC if an add or remove operation failed because the ring overflowed or underflowed. The programmer must guarantee through software means that no such errors occur, or use the event system to trigger such a signal.

The rings are configured through a CLS read and write commands, and must be accessed with 64-bit aligned addresses. CSR address space is 0x10000 to 0xfffff inclusive.

## Instruction Format

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

## Parameter Descriptions

Parameter	Command	Description
cmd	add_tail	See <code>ring_add_to_tail_ptr</code> . (deprecated)
	get	See: <code>ring_get</code>
	get_safe	See: <code>ring_get_freely</code>
	journal	See: <code>ring_journal</code>
	pop	See: <code>ring_pop</code>
	pop_safe	See: <code>ring_pop_freely</code>
	put	See: <code>ring_put</code>
	<code>ring_add_to_tail_ptr</code>	Update tail pointer with an offset. Do not write any words to the ring. The offset is specified using a <code>byte_mask</code> override (See <a href="#">Note 1</a> ). If there are not enough empty slots in the ring, the tail pointer is not updated. This will raise an overflow event if the ring is configured to deliver one. If <code>ref_cnt</code> is 1 or 3 then Ring is specified by <code>address[5:2]</code> . If <code>ref_cnt</code> is 2 or 4 then Ring is specified by <code>address[19:16]</code> and sequence is specified in <code>address[15:2]</code> . Address[20] must be set to 1.
	<code>ring_get</code>	Read 32-bit words from head of ring and update head pointer. If there are not enough entries in the ring, no valid data is returned and the head pointer is not updated. This will raise an underflow event if the ring is configured to deliver one. Ring is specified by <code>address[5:2]</code> . <b>Not for Journals.</b>
	<code>ring_get_freely</code>	Read 32-bit words from head of ring and update head pointer. If there are not enough entries in the ring, the ring will be emptied and zero will be returned in place of the missing entries. No underflow event will be raised. Ring is specified by <code>address[5:2]</code> . <b>Not for Journals.</b>
	<code>ring_journal</code>	Perform a ring put of 32-bit data where index determines which ring. If there are not enough empty slots the ring wraps around, overwriting the oldest items. The head pointer is not updated and no overflow event will be raised.

Parameter	Command	Description
		In effect, a journal only uses the tail pointer and has no concept of overflow. Ring is specified by address[5:2].
	ring_ordered_lock	Permits a ring to be locked for reordering. Compare sequence number with ring head pointer on a specified ring. If match is found then push signal only, otherwise write the signal information to the ring at offset specified by sequence number. Sequence number and ring are specified in the address field. Address[32:20] must be set to 0x1. Address[19:16] specifies which ring to use as a reorder queue. Address[15:2] specifies sequence number to reorder on.
	ring_ordered_unlock	Unlocks a ring after reordering. Increment the head pointer on specified ring. Read the ring at the new head pointer and overwrite with zero, if the data was non-zero then push to the signal data that was read. Sequence number and ring are specified in the address field. Address[32:20] must be set to 0x1. Address[19:16] specifies which ring to use as a reorder queue. Address[15:2] specifies sequence number to reorder on. Ring is specified by address[19:16].
	ring_pop	Read 32-bit words from tail of ring and update tail pointer. If there are not enough entries in the ring, no valid data is returned and the tail pointer is not updated. This will raise an underflow event if the ring is configured to deliver one. Note that a single pop operation will return the words in the order they were added. If the ring contains the words (A, B, C, D, E), then a single pop operation of 3 words will return the values (C, D, E) - in that order. Ring is specified by address[5:2]. <b>Not for Journals</b> .
	ring_pop_freely	Read 32-bit words from tail of ring and update tail pointer. If there are not enough entries in the ring, the ring will be emptied and zero will be returned in place of the missing entries. No underflow event will be raised. Note that a single pop operation will return the words in the order they were added. If the ring contains the words (A, B, C, D, E), then two pop operations of 3 words will return the values (C, D, E) and (A, B, 0) - in that order. Ring is specified by address[5:2]. <b>Not for Journals</b> .
	ring_put	Write 32-bit words to tail of ring and update tail pointer. If there are not enough empty slots in the ring, no words are added and the tail pointer is not updated. This will raise an overflow event if the ring is configured to deliver one. Ring is specified by address[5:2].
	ring_read	Read number of 32-bit data from specified offset within a ring, bounded by the ring size. Address fields specify ring and offset. Address[20] must be set to 1. Ring is specified

Parameter	Command	Description
		by Address[19:16]. Address[15:2] specifies the offset from base for ring to read and is bounded by ring size.
	ring_workq_add_thread	Add a handle to the workq ring. Ring is specified by Address[5:2].
	ring_workq_add_work	Add work to the work ring, where index determines which ring. Adding work to a queue that contains threads will get the first thread and deliver the work to it. Work item is in 32-bit words. Ring is specified by Address[5:2].
	ring_write	Write number of 32-bit data from write transfer registers to a specified offset within a ring, bounded by the ring size. Address fields specify ring and offset. Address[20] must be set to 1. Ring is specified by Address[19:16]. Address[15:2] specifies the offset from base for ring to write and is bounded by ring size.
xfer	get, pop, get_safe, pop_safe, ring_get, ring_get_freely, ring_pop, ring_pop_freely, ring_read, ring_workq_add_thread	Read transfer registers.
	put, journal, ring_journal, ring_put, ring_workq_add_work, ring_write	Write transfer registers.
	add_tail, ring_add_to_tail_ptr, ring_ordered_lock, ring_ordered_unlock	Must be omitted; should be “--”.
src_op1 , src_op2	All commands	Restricted source operands for address. The address is specified by src_op1 + src_op2. Four address bits are used to select the ring for the command. (See <a href="#">Note 2</a> ).
ref_cnt	ring_write	Reference count in 32-bit words. Valid values are 1-32. Values above 8 must be specified using indirect reference.
	get, pop, get_safe, pop_safe, put, journal, ring_get, ring_journal, ring_get_freely, ring_pop, ring_pop_freely, ring_put, ring_read, ring_workq_add_thread, ring_workq_add_work	Reference count in 32-bit words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
	add_tail, ring_add_to_tail_ptr	Reference count valid values are 1-4. Used to encode ring_add_to_tail_ptr command variant. Refer to NFP-6xxx Databook.

Parameter	Command	Description
opt Tok	get, pop, get_safe, pop_safe, put, ring_get, ring_get_freely, ring_journal, ring_pop, ring_pop_freely, ring_put, ring_read, ring_workq_add_thread, ring_workq_add_work, ring_write	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> . ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> . indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> . ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .
	add_tail, ring_add_to_tail_ptr, ring_ordered_lock, ring_ordered_unlock	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

**Notes:**

1. For the add\_tail commands the offset can be specified using the V2 Indirect Reference “Indirect CSR BYTE\_MASK” ([Section 2.1.6.4.6.2](#)).

V1 indirect override:

Override offset using byte mask																																	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
0	1	0	0	Reserved																													

For add\_tail the offset is specified in 32-bit words minus 1 (a value 0 means offset of 1 word).

2. The ring\_add\_to\_tail\_ptr command variants specify which ring to adjust tail pointer; (Index) is specified in address[5:2] or address[19:16]. The address bits that specify Index field is dependent upon the Length field (Length). The amount to be added to the ring is encoded in byte\_mask[4:0]. With byte\_mask of 0 add 1 to the tail pointer, with byte\_mask of 5b11111 add 32 to the tail pointer. See NFP-6xxx Databook.

**Condition Codes Affected**

N	Z	V	C
Not Affected			

## [2.2.37.8 CLS \(SRAM Read and Write\)](#)

Move data between FPCs and Cluster Local Scratch memory. All accesses occur internally using 64-bit words. Read accesses will be automatically aligned and any extra data discarded. Write accesses not operating on aligned 64-bit words will result in read-modify-write operations.

**Instruction Format**

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

**Parameter Descriptions**

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
cmd	read, read_be	Read 32-bit words from cluster local scratch in big endian (LWBE) format. The read data will be automatically byte aligned.
	read_le	Read 32-bit words from cluster local scratch in little endian format. The read data will be automatically byte aligned.
	write, write_be	Write 32-bit words to cluster local scratch in big endian (LWBE) format. Unless the operation is 64-bits and 64-bit aligned, the first and/or last word of the transaction will require a read-modify-write operation.
	write_le	Write 32-bit words to cluster local scratch in little endian format. Unless the operation is 64-bits and 64-bit aligned, the first and/or last word of the transaction will require a read-modify-write operation.
	write8, write8_be	Write bytes to cluster local scratch in big endian (LWBE) format. The write will be translated into 64-bit read-modify-write operations on the first and last 64-bits of data (if required). The intervening 64-bit words will be regular writes.
	write8_le	Write bytes to cluster local scratch in little endian format. The write will be translated into 64-bit read-modify-write operations on the first and last 64-bits of data (if required). The intervening 64-bit words will be regular writes.
xfer	read, read_be, read_le	Read transfer registers.
	write, write_be, write_le	Write transfer registers.
src_op1, src_op2	All commands	Restricted source operands for cluster local scratch address. The address is specified by src_op1 + src_op2.
ref_cnt	read, read_be, read_le, write, write_be, write_le	Reference count in 4-byte words. Valid values are 1-32. Values above 8 must be specified using indirect reference.
	write8, write8_be, write8_le	Reference count in bytes. Valid values are 1-32. Values above 8 must be specified using indirect reference.
opt_tok	All commands	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Condition Codes Affected

<b>N</b>	<b>Z</b>	<b>V</b>	<b>C</b>
Not Affected			

## 2.2.37.9 CLS (TCAM Lookup)

Treat area of Cluster Local Scratch SRAM as a TCAM and perform lookup on it. A TCAM line is a 64-bit aligned SRAM value interpreted in LWBE order. Each TCAM line is compared to data in write transfer register, where TCAM data and TCAM mask is compared to write Transfer register data as follows: (pull\_data & TCAM\_mask == TCAM\_data & TCAM\_mask). When match is found the Match mask and offset fields are updated.

### Instruction Format

```
cls[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	tcam_lookup, tcam_lookup32	Compare whole 32-bit TCAM word with 32-bit input data.
	tcam_lookup24	Compare lower 24-bits of TCAM word with lower 24-bits of input data. The upper 8 bits of TCAM match word and TCAM mask are ignored.
	tcam_lookup16	Split TCAM match word into two 16-bit words and compare each with lower 16-bits of input data.
	tcam_lookup8, tcam_lookup_byte	Split TCAM match word into four 8-bit words and compare each with lower 8-bits of input data.
xfer	tcam_lookup, tcam_lookup32, tcam_lookup24	One write transfer register containing input data. One read transfer register containing result. See <a href="#">Note 1</a> .
	tcam_lookup16, tcam_lookup8	One write transfer register containing input data. Two read transfer registers containing result. See <a href="#">Note 2</a> .
src_op1, src_op2	All commands	Restricted source operands for cluster local scratch TCAM address. The address is specified by src_op1 + src_op2. The address must be 64-bit aligned.
ref_cnt	All commands	Size of TCAM in 64-bit words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
opt Tok	All commands	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Notes:

1. TCAM lookup pulls 32-bit data from write transfer register and mask-compares it with every TCAM line in SRAM, where each TCAM line is 64-bits LWBE order. The lower word (lower addressed 32-bits) is TCAM match data and upper word is TCAM mask. A 24-bit TCAM match ignores bits [31:24] of the match data, and effectively these bits are also masked out in the comparison.

32-bit and 24-bit TCAM lookups have a single 32-bit result delivered in a read transfer register:

Result of 32-bit and 24-bit TCAM lookups																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Match mask																Entry bits								Offset							

Bits	Bit width	Field Name	Description
31:16	16	Match mask	Bitmask of all matches $n$ . Bitmask of all 24-bit or 32-bit word comparisons that satisfied the TCAM match. Bit assignment to TCAM match longword is shown in <a href="#">Table 2.25</a> .
15:8	8	Entry bits	Bits [31:24] of matching entry. For 24-bit lookups this provides for 8-bit of client data in the TCAM match entry.
7:0	8	Offset	If match occurred, offset to the TCAM entry which matched. If match did not occur, the value 0x7F.

2. TCAM lookup pulls 32-bit data from write transfer register and mask-compares bits 0 through 7 with every multiple-quad-byte TCAM line in SRAM, where TCAM line is 64-bits LWBE order. The lower word (lower addressed 32-bits) is TCAM match data and upper word is TCAM mask. The incoming data is replicated to a 32-bit value for the match with the TCAM data. The matching process will include four match bits for each TCAM compare, one per byte. 16-bit TCAM match effectively bonds bytes 0/1 and bytes 2/3.

16-bit and 8-bit TCAM lookups have a 64-bit result delivered in two read transfer registers (least significant word first):

Result of 16-bit and 8-bit TCAM lookups																															
63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
Match mask (hi)																															

Result of 16-bit and 8-bit TCAM lookups																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Match mask (lo)																Offset															

Bits	Bit width	Field Name	Description
63:8	56	Match mask	Bitmask of the last $n$ byte comparisons, with

Bits	Bit width	Field Name	Description
			bit [63] always indicating the result of the last byte comparison and the following bits indicating previous comparisons. For example, with a TCAM lookup of 6 words, bit [40] indicates result of first byte comparison, bit [41] indicates result of next comparison, etc.
7:0	8	Offset	If match occurred, offset to the TCAM entry which matched. If match did not occur, the value 0x7F.

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Example of CLS TCAM lookup32

The CLS TCAM supports various sizes of input match data and TCAM table sizes. TCAM table reside in Cluster Internal SRAM memory. NFP-6xxx memory architecture provides hardware acceleration engines for TCAM operations.

TCAM example below is configured for 512-bit TCAM, eight long word TCAM entries, where each entry is 32-bits of Data and 32-bits of Mask value.

The `cls[tcam_lookup32]` will push to the read transfer register the following:

- `$xfer0[7:0]` = Index of the first match (shifted left by 3). Another way to view this value – it is the offset from TCAM starting memory address. (See [Table 2.25](#) )
- `$xfer0[15:8]` = Contents of lowest matched TCAM entry memory data[31:24]
- `$xfer0[31:16]` = Bit mask of all matches found in TCAM table. See [Table 2.25](#) column "bitmask map to Rd xfer[31:16]" for bit assignment to TCAM offset.

If there is no TCAM match the read transfer register will contain the value of `0x0000_00ff`.

For a particular TCAM lookup where the result is a match, the actual lookup might result in multiple matches given the TCAM configuration of data and mask words. The read transfer register bit[7:0] will contain the lowest TCAM entry offset where a match was found. This value is a long word offset from the start of TCAM memory entry# 0. Valid values will be `0x00, 0x08, 0x10, 0x18, 0x20, 0x28, 0x30` and `0x38`. Read transfer register [31:16] provides a bit map of all TCAM matches, where "bitmask map to Rd xfer[31:16]" column shows which bits map to which TCAM offset. (See [Table 2.25](#) )

In the TCAM Table below the TCAM MASK values determine which bits to mask during the TCAM lookup operation, where each 32-bit TCAM Data word and Match data word (from write transfer register) are compared after applying the TCAM mask. If the mask bit is cleared then the bit is don't care and if set then the bit is used in match operation.

For each long word (data and mask) in TCAM table, a compare is performed with the Match data (pull data), TCAM data and TCAM mask. If both bit-wide AND operations of the Match data and TCAM data against the TCAM mask word evaluate to be TRUE then there is a match. The compare expression is shown below followed by three TCAM long word match operation examples.

If the write transfer register (Match data) contains value of 0xba011ee5 the TCAM data is evaluated as follows:

**Compare Expression: (Match\_data & TCAM\_Mask == TCAM\_Data & TCAM\_Mask)**

```
; Three evaluations - no match and two match examples:  
Match data presented to TCAM is 0xba011ee5.  
(Match data & TCAM Mask == TCAM data & TCAM Mask) ? Match : NO Match  
  
0xba011ee5 & 0xffff00f00 == 0xaaadbef1 & 0xffff00f00  
-> 0xba000e00 != 0xaaa00e00 ; NO match  
  
0xba011ee5 & 0xffff00ff3 == 0xba000ee5 & 0xffff00ff3  
-> 0xba000ee1 == 0xba000ee1 ; Match  
  
0xba011ee5 & 0x8a000f21 == 0xba00beel & 0x8a000f21  
-> 0x8a000e21 == 0x8a000e21 ; Match
```

Read Transfer register will contain the following after a TCAM lookup32:

\$xref[7:0] = 0x20 ; longword offset

\$xref[15:8] = 0xba ; TCAM data[31:24] at offset 0x20

\$xref[31:16] = 0x4100 ; bit mask for all matches in TCAM. See "bitmask map to Rd xfer[31:16]" column in [Table 2.25](#). Where value of 0x4100 says match found for entry4 (offset 0x20) and entry7 (offset 0x38) for the TCAM lookup32.

The fields in the read transfer register is the lookup result and the complexity can be extended by how one constructs the TCAM table given the match data expected.

In the table below TCAM Data and TCAM Mask values are pre-loaded into CLS memory prior to TCAM lookup. The TCAM Match Data value is compared to all eight TCAM entries.

**Table 2.25. TCAM Table layout for cls[tcam\_lookup32] example**

Offset	TCAM Entry#	bitmask map to Rd xfer[31:16]	CLS Address	TCAM DATA value (lower addr word)	TCAM MASK value (upper addr word)
0x00	0	xref[16]	0x4000:4007	0xaadbef1	0xffff00f00
0x08	1	xref[18]	0x4008:400f	0x123dbef0	0x00ffff00
0x10	2	xref[20]	0x4010:4017	0xabcd bef1	0xffff00f00
0x18	3	xref[22]	0x4018:401f	0x5656ab40	0xffffffff00
0x20	4	xref[24]	0x4020:4027	0xba000ee5	0xffff00ff3
0x28	5	xref[26]	0x4028:402f	0x55a78780	0xf0f7f7f0
0x30	6	xref[28]	0x4030:4037	0abcd0010	0xffff00f1f
0x38	7	xref[30]	0x4038:403f	0xba00bee1	0x8a000f21

The code below runs on a thread that receives a 32 bit match data value constructed from a packet. The objective of this code is to search in the TCAM for a match and return the pushed data for determining what to do with the packet. The TCAM has been pre-configured with the values shown above in TCAM Table layout.

```

; TCAM lookup32 example
.sig s1 s2
.reg r_match_data
.reg ind_ref_reg
.reg me_num
.reg address
.reg data
.reg cl_num

.reg read write $xref0 $xref1, $xref2, $xref3
.xfer_order $xref0 $xref1, $xref2, $xref3

; The ME_NUM is in bits [7:3]
; CL_NUM[3:0] is in bit [28:25]
local_csr_rd[ACTIVE_CTX_STS]
immed[me_num,0]

; Derive address from island and ME
alu_shf[cl_num, 0x3F, AND, me_num, >>25]
alu_shf[me_num, 0xF, AND, me_num, >>3]

alu_shf[address,--, B, me_num, <<12]
alu_shf[address, address, OR, cl_num, <<26]

-----
; tcam lookup
;     Read Transfer register $xref0
;     Match returns    [7:0] = match<<3, or TCAM memory offset
;     Match returns    [15:8] = Upper 8 bit of TCAM data entry
;     Match returns    [31:16] = Bit set for each entry that matches (see table above)
-----

; r_match_data was pre-loaded before entering this code block. It contains match
; data to be applied to TCAM lookup32 operation.

```

```

; Move r_match_data containing the 0xba011ee5 into $xfer0 write transfer register
alu[$xref0, --, B, r_match_data]
; Perform TCAM lookup32 across eight long words (64 bits memory words)
; If used tcam_lookup24 in place of tcam_lookup32 below then top 8 bits are ignored.
; Nothing else changes.
cls[tcam_lookup32, $xfer0, address, 0, 8], ctx_swap[sig2]

; Check lookup results
alu[--, $xref0, -, 0xff]
beq[NO_MATCH#]
; Match found, send 32-bit result in $xref0 to statistics handler
; Forward the packet based upon the result data.
; $xref will contain the value of 0x4100ba20
; Apply to hash table
br[DONE#]

NO_MATCH#:
; No match found in TCAM drop the packet or handle this as exception case.

DONE#:

```

What follows are two high level examples of a TCAM lookup32 instruction intended to help the user understand how TCAM operations results should be interpreted.

Example 1: (Note that the example is shown in Big Endian format for clarity)

When a TCAM entry mask is zero, by definition, you will get a match:

Pull\_Data AND Entry Mask=0x0 == Entry Data AND Entry Mask=0x0 is a match (i.e., 0x0 = 0x0)

The Pull Data is 0x00000001.

The data in cls tcam is:

TCAM Entry 0-----TCAM Entry 1

Data + Mask-----Data + Mask

0x0000000000000000	0x0000000000000000	--	TCAM Entry0/1 => Match0=1, Match1=1
0x0000000000000000	0x0000000100000001	--	TCAM Entry2/3 => Match2=1, Match3=1
0x0000000100000001	0x0000000100000001	--	TCAM Entry4/5 => Match4=1, Match5=1
0x0000000100000000	0x0000000000000001	--	TCAM Entry6/7 => Match6=1, Match7=0

You will see a Return data value of 0xbfff0000 for this operation, where

[7:0] are the offset of the first matched entry,

[15:8] are the uppermost byte of the data from the first matched entry and

[31:16] are the bit mask match bits (odd numbered bits are ignored for TCAM operations)

So,

offset 0x00

data 0x00

bit mask match 0xbfff

Bits [7:0] are 0x00 because the first match is at entry 0.

Bits [15:8] are 0x00 from bits [31:24] from the first match are 0x00.

The bit\_mask match portion of the Return data (return[31:16]) is where you look to find how many matches were found. For TCAM operations, only bits 30, 28, 26, 24, 22, 20, 18, & 16 of the bit mask match within the Return data are valid.

These bits are Match7 - Match0, from above.

Bits 31, 29, 27, 25, 23, 21, 19 & 17 of the Return data are ignored for TCAM operations.

From the example above we have 0111\_1111 for bits 30, 28, 26, 24, 22, 20, 18, & 16. Entries 0, 1, 2, & 6 are matches due to the Mask being 0x0. Entries 3, 4, & 5 are outright matches.

Example 2:

```
I changed the cls tcam data to:  
TCAM Entry 0-----TCAM Entry 1  
Data + Mask-----Data + Mask  
0x000100000010000 0x0000030000000300 -- TCAM Entry0/1 => Match0=0, Match1=0  
0x000000c0000000c0 0x0000000100000001 -- TCAM Entry2/3 => Match2=0, Match3=1  
0x0000000100000001 0x0000000100000001 -- TCAM Entry4/5 => Match4=1, Match5=1  
0x0000000100000000 0x0000000000000001 -- TCAM Entry6/7 => Match6=1, Match7=0  
The Pull Data is 0x00000001.
```

You will see Return data of 0xbfc00018 for this operation, where

[7:0] are the offset of the first matched entry,  
[15:8] are the uppermost byte of the data from the first matched entry and  
[31:16] are the bit mask match bits (odd numbered bits are ignored for TCAM operations)

So,

```
offset 0x18 (entry 3<<3 = 0x18)  
data 0x0 (bits [31:24] from first match)  
bit mask match 0xbfc0
```

The bit mask match portion of the Return data (return[31:16]) is where you look to find how many matches were found. For TCAM operations, only bits 30, 28, 26, 24, 22, 20, 18, & 16 of the bit mask match within the Return data are valid. These bits are Match7 - Match0, from above. Bits 31, 29, 27, 25, 23, 21, 19 & 17 of the Return data are ignored for TCAM operations.

From the Match7-0 bits above we have 0111\_1000 for bits 30, 28, 26, 24, 22, 20, 18, & 16, so entries 3, 4, 5 & 6 are matches (entry 6 is a match because the mask is 0x0).

## 2.2.38 Cluster Target (CT)

### 2.2.38.1 CT (Cluster Target)

The Cluster Target Memory Unit is tightly coupled with the local cluster masters for quick command execution and data transfers. One of the subcomponents of the Cluster Target MU is the Misc Engine. The functions of the MU Misc Engine are:

- XPB Arbiter\_Master
- Cluster Target Next Neighbor Writes
- Reflector Support
- Circular Rings
- Inter-Thread Signalling Support

The Misc Engine receives commands via the XPB master interface or the CPP command bus. On the CPP bus the Misc Engine responds only to the XPB/ClusterTarget target ID. The Misc Engine execution units will process XPB read and write commands, and the other will handle all the other Misc Engine commands. All commands are shown below.



### Note

The CTM has no understanding of the source/destinations specified in CT reflect\_read\* and reflect\_write\* commands. These commands are treated as standard pull/push ops from/ to xfer register fields. It is therefore the command issuer's responsibility to ensure that the command appropriately utilizes the xfer and ref\_cnt fields according to the source/ destination used.



### Note

Cluster Target Reflector operations are limited to 8 bits of CSR address (not including the LocalCsr/XferReg select). Accessing CSRs outside this addressable range should be done with Local Scratch (CLS) Reflector commands. The Local Scratch supports a wider addressable range via additional address bits for Reflector operations.



### Note

**Important:** . The ring registers must be programmed prior to any ring operations. RingBase should be programmed as desired and the RingHead/RingTail registers must be written to 0. While the Ring is in use the RingHead and RingTail registers are maintained by hardware. It is recommended that users follow any XPB writes of the Ring CSRs with an XPB Read operation to the same CSR before issuing any ring operations. This will ensure the CSR update is complete before accessing the rings.

**Important:** . The hardware does not prevent overfilling the ring. It is the responsibility of the 'Putting' thread to guarantee that the Ring has room before doing a RingPut.

**Note:** . The local MEs in the cluster can only accept status signals from fifteen of the rings. Therefore, only the low fifteen (0-14) rings provide ring status indicators.

## Instruction Format

```
ct[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

## Parameter Descriptions

Parameter	Command	Description
cmd	ctnn_write	Write data to the next neighbor register or FIFO of a Flow Processor. Specify command arguments island, ME, signal, AddressMode and register in the command address.  Where address[29:24] specifies the island number of the XPB target device, address[20:17] the ME (FPC)

Parameter	Command	Description
		<p>ID within the island to access, address[16:10] the signal number within the ME to set on completion, address[9] AddressMode: if set, relative addressing (absolute mode) and NN register number indicated by address[8:2]; if cleared, NN register FIFO mode (Indexed mode) is used. address[8:2] the NN register number. If ME is in NN FIFO mode address[15:2] is ignored.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	interthread_signal	<p>Instruct MU Misc Engine to send signal to another Flow Processor core over the CPP bus. Specify command arguments Island, ME, context and signal in the command address.</p> <p>Where address[29:24] is the Island number of the XPB target device, address[12:9] the ME (FPC) to signal, address[8:6] the thread (context) number, and address[5:2] the signal number.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	reflect_read_none	<p>Read data from a remote FPC and write it to the initiating FPC without providing a signal. Specify command arguments island, XferCsrRegSel select, ME and XferCsrRegAddress select.</p> <p>Where address[29:24] specifies the Island Id of remote FPC to read from, address[16] is the XferCsrRegSel select bit (0: Transfer Registers, 1: CSR Registers), address[13:10] is the master number (= FPC + 4) within the island to read data from, address[9:2] is the first register address (Register depends upon XferCsrRegSel).</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	reflect_read_sig_both	<p>Read data from a remote Flow Processor and write it to the initiating FPC, signaling the remote Flow Processor on the read and the initiating Flow Processor on the write. See reflect_read_none command for specifying command arguments in command address.</p>
	reflect_read_sig_init	<p>Read data from a remote Flow Processor and write it to the initiating Flow Processor, signaling only the initiating Flow Processor. See reflect_read_none command for specifying command arguments in command address.</p>

Parameter	Command	Description
	reflect_read_sig_remote	Read data from a remote FPC and write it to the initiating Flow Processor, signaling only the remote FPC on the read. See reflect_read_none command for specifying command arguments in command address.
	reflect_write_none	<p>Read data from the initiating FPC and write it to a remote Flow Processor without providing a signal. Specify command arguments island, XferCsrRegSel select, ME and XferCsrRegAddress select in command address field.</p> <p>Where address[29:24] specifies the Island Id of remote ME to write to, address[16] is the XferCsrRegSel select bit (0: Transfer Registers, 1: CSR Registers), address[13:10] is the master number (= FPC + 4) within the island to write data to, address[9:2] is the first register address (Register depends upon XferCsrRegSel) to write to.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	reflect_write_sig_both	Read data from the initiating FPC and write it to a remote Flow Processor, signaling the initiating FPC on the read and the remote FPC on the write. See reflect_write_none command for specifying command arguments in command address.
	reflect_write_sig_init	Read data from the initiating FPC and write it to a remote FPC, signaling only the initiating FPC on the read. See reflect_write_none command for specifying command arguments in command address.
	reflect_write_sig_remote	Read data from the initiating FPC and write it to a remote FPC, signaling only the remote FPC on the write. See reflect_write_none command for specifying command arguments in command address.
	ring_get	<p>Retrieve 32-bit data from a circular ring within the CTM memory. Specify command arguments island and Ring number in command address. The hardware computes the address using the ring number by concatenating BaseAddress[ring] and HeadOffset[ring]. See NFP-6xxx Databook MUMERingBase and MUMERingHead in section MU Registers and Breakouts.</p> <p>Where address[29:24] specifies the Island Number of the target CTM and address[5:2] the Ring number.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>

Parameter	Command	Description
	ring_put	<p>Put 32-bit data onto a circular ring within the CTM memory. Specify command arguments island and ring number in the command address. The hardware computes the address using the ring number by concatenating BaseAddress[ring] and HeadOffset[ring]. See NFP-6xxx Databook MUMERingBase and MUMERingHead in section MU Registers and Breakouts.</p> <p>Where address[29:24] specifies the Island Number of the target CTM and address[5:2] the Ring number.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	xpb_read	<p>Read data from a XPB target over the XPB bus. Transactions from the ARM may be global, but from other islands must be to within their island or its slaves. Specify command arguments global/local select, island, XPB slave ID, XPB device ID and XPB Target address in command address field.</p> <p>Where address[30] specifies Global/Local XPB select (0: Local XPB, 1: Global XPB), address[29:24] the Island number of XPB target (if 0: use own island ID), address[23:22] the slave XPB Slave ID (when accessing XPB Slave Island), address[21:16] the XPB Device ID number of XPB target, address[15:2] the XPB Target Register address of the first 32-bit register to be accessed in the burst.</p> <p>Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.</p>
	xpb_write	<p>Write data to a XPB target over the XPB bus. Transactions from the ARM may be global, but from other islands must be to within their island or its slaves. Specify command arguments global/local select, island, XPB slave ID, XPB device ID and XPB Target address in command address field.</p> <p>Where address[30] specifies Global/Local XPB select (0: Local XPB, 1: Global XPB), address[29:24] the Island number of XPB target (if 0: use own island ID), address[23:22] the slave XPB Slave ID (when accessing XPB Slave Island), address[21:16] the XPB Device ID number of XPB target, address[15:2] the XPB Target Register address of the first 32-bit register to be accessed in the burst.</p>

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
		Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.
xfer	ctnn_write, reflect_write_none, reflect_write_sig_both, reflect_write_sig_init, reflect_write_sig_remote ring_put, xpb_write	Write transfer registers that hold data that is pulled by target.
	reflect_read_none, reflect_read_sig_both, reflect_read_sig_init, reflect_read_sig_remote, ring_get, xpb_read	Read transfer registers that hold data that is pushed by target.
	interthread_signal	Not Required. See <a href="#">Section 2.1.6.3</a>
src_op1 , src_op2	ctnn_write, interthread_signal, reflect_read_none, reflect_read_sig_both, reflect_read_sig_init, reflect_read_sig_remote, reflect_write_none, reflect_write_sig_both, reflect_write_sig_init, reflect_write_sig_remote, ring_get, ring_put, xpb_read, xpb_write	Restricted source operands (src_op1 and src_op2) combine to form preferred 40-bit address that encodes to meaningful parameters for each command. Refer to each command above for address bits and description of each encoded parameter.
ref_cnt	ctnn_write, reflect_read_none, reflect_read_sig_both, reflect_read_sig_init, reflect_read_sig_remote, reflect_write_none, reflect_write_sig_both, reflect_write_sig_init, reflect_write_sig_remote, ring_get, ring_put, xpb_read, xpb_write	Reference count in 4-byte words. Valid values are 1-16 for commands ring_get and xpb_read. For all other commands, valid values are 1-14. Values above 8 must be specified using indirect reference. Refer to section <a href="#">Section 2.1.6.4.2</a>
	interthread_signal	Not Required. See <a href="#">Section 2.1.6.3</a>
opt Tok	interthread_signal, reflect_read_none, reflect_write_none	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a>
	ctnn_write, reflect_read_sig_both, reflect_read_sig_init,	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .

Parameter	Command	Description
	reflect_read_sig_remote, reflect_write_sig_both, reflect_write_sig_init, reflect_write_sig_remote, ring_get, ring_put, xpb_read xpb_write	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> . ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Condition Codes Affected

N	Z	V	C
Not Affected			

### Examples of ct[reflect]

#### ct[reflect\_read\_sig\_xxx] Description

Read 32-bit data words from the remote FPC xfer/csr register and write data to initiating FPC xfer/csr register who initiates ct[reflect\_read]. The initiating FPC and/or remote FPC can be signaled. The FPC signaled on data transfer is determined by the reflect\_read\_sig command.

#### ct[reflect\_write\_sig\_xxx] Description

Read 32-bit data words from the initiator FPC xfer/csr register and write data to remote FPC xfer/csr register. The initiating FPC and/or remote FPC can be signaled. The FPC signaled on data transfer is determined by the reflect\_write\_sig command.

#### ct[reflect] Address[31:0] fields definition

Address[31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2b0	Island#				7b0				XferCsrRegSel	2b0		Master ID#		Xfer Reg Adr						2b0											

The address field used with ct[reflect\_read] (ct[reflect\_write]) specifies the following:

- address[29:24] specifies the Island ID of remote FPC to read from (write to)
- address[16] specifies XferCsrRegSel select bit (0: Transfer Register, 1: CSR Register)
- address[13:10] specifies the FPC within the island to read data from (write data to)
- address[9:2] specifies the first register address to read from (write to)

Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.

### **Example 1. ct[reflect\_read\_sig\_init]**

In this example FPC 2 reads remote FPC 4 xfer1 register and the data is pushed into FPC 2 xfer1 register. The initiating FPC (FPC 2) will receive a signal when data push is completed.

The address field used in ct[reflect\_read\_sig\_init] is not a memory address but ct[] encoded data for transferring data between FPCs. The encoded fields for this example are explained below and apply to subsequent examples below.

**Island #** field specifies the Island number for the operation.

**Master ID#** field for FPC 4 is set to 8 (4+4) and is placed in addr1[13:10] in example below. The FPC ID is added to the value of 4. Harrier specifies FPC 0-3 as NOT USED thus start with 4.

**Xfer Reg Adr** Specifies the first xfer register to read from FPC 4. With Xfer Reg Adr specified the ref\_cnt in ct[] instruction argument indicates how many contiguous xfer registers to read from FPC 4. The value derived from the &remote() is loaded in addr1[9:2] Xfer Reg Adr field show below and in FPC 2 example code.

The “**remote**” keyword in example below is provided so the named transfer register “\$r\_xfer1” is declared as visible in FPC 4 and is seen by FPC 2. Also FPC 2 must declare “\$r\_xfer1” as remote. “**\_\_nfp\_meid(island, menum)**”, with arguments island=32 and menum=4 used to specify \$r\_xfer1 located in Island 32 FPC 4 and is accessed using normal src/dst references. **Refer to Netronome ug\_nfp6000\_nfas.pdf for remote and \_\_nfp\_meid() details.**

**XferCsrRegSel** is cleared indicating xfer register is the resource, otherwise one can specify CSR register by setting this bit to 1. In this example addr1[16] is low implicitly.

```

; -----
; | FPC 2
; -----

.reg addr1 ; Will hold ct[] encoded data
.reg remote $r_xfer1 ; must be declared in FPC2 as stated above.
.reg $my_xfer1 ; declare FPC xfer1 register where data is pushed from ct instruction.
.sig sig3

; Begin to compose addr1 for ct[] instruction below.
; See remote description above example block.
immed[addr1, ((4 + 4) <<10 | &remote($r_xfer1, (__nfp_meid(32, 4))) <<2)]

; Set the destination island# in address[29:24] field. Must be specified for remote or
; local destination island#. Use immed_w1[] to load island# in addr1[29:24].
immed_w1[addr1, (32 <<8)] ; specify remote FPC island# as 32 to read from.

; Next issue ct[reflect_read_sig_init] operation and data from FPC 4 $r_xfer1 register is
; pushed to FPC2 $my_xfer1 register.
; After data is pushed into FPC 2 xfer1 In register, FPC 2 receives sig3 signal indicating
; data is pushed. In ct[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
; ref_cnt is 1 indicating read 1 xfer register from remote FPC 4.
ct[reflect_read_sig_init, $my_xfer1, addr1, 0, 1], ctx_swap[sig3]

; -----
; | FPC 4
; -----

```

```
.reg visible $r_xfer1 ; specify as visible to FPC 2
```

### Example 2. ct[reflect\_read\_sig\_both]

There are two ways to signal the remote FPC and the first example below specifies to use sig2 to both initiator and remote FPCs.

Example 2A: The code is similar to example 1 ct[reflect\_read\_sig\_init], but in this example both initiator FPC and remote FPC is signaled by sig2. FPC 4 is signaled when data is pulled and FPC 2 is signaled when data is pushed. Here ctx\_arb[signal] must be used to check if signal 2 is returned.

```
; -----
; | FPC 2
; -----

.reg addr1
.reg remote $r_xfer1
.reg $my_xfer1
.sig sig2

immed[addr1, ((4+4) <<10 | &remote($r_xfer1, (__nfp_meid(32,4))) <<2)]
immed_w1[addr1, (32<<8)]

ct[reflect_read_sig_both, $my_xfer1, addr1, 0, 1] , sig_done[sig2]
ctx_arb[sig2]
```

Example 2B: The code is similar to example 1 ct[reflect\_read\_sig\_init], but here both initiator FPC and remote FPC is signaled. In this example FPC 2 is signaled by sig3 and FPC 4 is signaled sig2. Sig2 is specified in signal\_ref which is specified by using NFP-6xxx indirect\_ref format in the initiating FPC ct[] instruction. Indirect\_ref optional token specifies that the output from the previous ALU instruction be used to override the signal number and the signal number is loaded in the Indirect CSR register bits [12:9]. FPC 4 is signaled when data is pulled and FPC 2 is signaled when data is pushed.

```
; -----
; | FPC 2
; -----

.reg addr1
.reg ind_ref_data
.reg $my_xfer1
.sig sig2, sig3

immed[addr1, ((4+4) <<10 | &remote($r_xfer1, (__nfp_meid(32, 4))) <<2)]
immed_w1[addr1, (32<<8)]

; Use CSR cmd_indirect_ref_0 to override signal when setting bit 13
; Override signal #2 to remote signal. sig2 is also used as local signal.
alu[ind_ref_data, --, B, 2, <<9] ; Put sig_num 2 in bits [12:9]
local_csr_wr[cmd_indirect_ref_0, ind_ref_data] ; write sig_num 2 to Indirect CSR Register
alu[ind_ref_data, --, B, 1, <<13] // Prev ALU operation, Bit 13 : Override SIG_NUM

; Signal both current and remote FPC
ct[reflect_read_sig_both, $my_xfer1, addr1, 0, 1] , ctx_swap[sig3], indirect_ref
```

### Example 3. ct[reflect\_read\_sig\_remote]

Example code is similar Example 2A and 2B above except only the remote FPC is signaled. Again there are two methods to specify the signal to send and they are shown in example 2 above and will show just the ct[] instruction in these two examples below.

Example 3A: Code is similar to Example 2A. Show only ct[] command to signal only remote FPC 4 with sig2. sig\_done[sig2] specifies to send a signal 2 to remote FPC.

```
; -----
;| FPC 2
;-----

; Code similar to Example 2A except ct[] command is signal remote FPC ONLY with sig2.
ct[reflect_read_sig_remote, $my_xfer1, addr1, 0, 1], sig_done[sig2]
ctx_arb[sig2]
```

Example 3B: Code is similar to Example 2B. Show only ct[] command to signal only remote FPC 4 with NFP6xxx Indirect\_ref to override signal number. ctx\_arb[sig3] is not valid since local FPC does not receive a signal when data is pushed.

```
; -----
;| FPC 2
;-----

; Code similar to Example 2B except ct[] command is signal remote FPC ONLY with sig3.
ct[reflect_read_sig_remote, $my_xfer1, addr1, 0, 1], ctx_swap[sig3], indirect_ref
```

#### **Example 4. ct[reflect\_read\_none]**

Do not generate a signal to either initiator or remote FPC when data is pushed and pulled respectively. Note that opt\_tok field is left blank, thus no signal is generated.

```
; -----
;| FPC 2
;-----

; Code similar to Example 1 except ct[] command does not signal initiator or remote FPC.
ct[reflect_read_none, $my_xfer1, addr1, 0, 1]
```

#### **Example 5. ct[reflect\_write\_init]**

The ct[reflect\_write\_sig\_xxx] commands are similar to ct[reflect\_read\_sig\_xxx] commands except that the data is read from the initiator FPC and written to the remote FPC. The ct[reflect\_write\_sig\_xxx] command signal can be generated to the local FPC when data is pulled and to the remote FPC when data is pushed. When a signal is generated to the remote FPC, sig\_done[signal] or indirect\_ref methods can be used as shown above in ct[reflect\_read\_sig\_both] and ct[reflect\_read\_sig\_remote] examples above.

In this example one 32-bit data is transferred from FPC 2 \$my\_xfer1 and is pushed to FPC 4 \$r\_xfer1. Addr1 setup is the same as in example 1 above. Signal sig3 is sent to initiator FPC 2 when data is pulled.

```
; -----
;| FPC 2
;-----

.reg addr1 ; Will hold ct[] encoded data
.reg remote $r_xfer1 ; must be declared in FPC2.
```

```

.reg $my_xfer1 ; declare FPC 2 xfer1 register where data is pulled from ct[] instruction.
.sig sig3 ; declare signal for ct[] instruction.

; Begin to compose addr1 for ct[] instruction below.
; See remote description just above ct[reflect_read_sig_init] Example 1.
immed[addr1, ((4 + 4) <<10 | &remote($r_xfer1, (__nfp_meid(32, 4))) <<2)]

; Set the destination island# in address[29:24] field even if local island.
; Must be specified for remote or local destination island#.
; Use immed_w1[] to load island# in addr1[29:24].
immed_w1[addr1, (32 <<8)] ; specify destination FPC island# even if local island.

; Load FPC 2 local transfer register ($my_xfer1) that will be written to FPC 4
; $r_xfer1 register.
immed[$my_xfer1, 0x222]

; Next issue ct[reflect_write_sig_init] operation and data is pulled from FPC 2
; $my_xfer1 register and is pushed into FPC 4 $r_xfer1 register.
; After data is pulled from FPC 2 xfer1 Out register, FPC 2 receives signal sig3 indicating
; data is pulled. In ct[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
; ref_cnt is 1 indicating read 1 xfer register from local FPC 2.
ct[reflect_write_sig_init, $my_xfer1, addr1, 0, 1], ctx_swap[sig3]

;

; -----
; |FPC 4
; -----

.reg visible $r_xfer1 ; specify as visible to FPC 2. Data is pushed to this register

```

Note ct[reflect\_write\_none], ct[reflect\_write\_sig\_both] and ct[reflect\_write\_sig\_remote] are similar to ct[reflect\_read\_none], ct[reflect\_read\_sig\_both] and ct[reflect\_read\_sig\_remote] except for where the data is pulled and pushed. In ct[reflect\_read\_xxx] data is read from remote FPC and written to initiator FPC, where in ct[reflect\_write\_xxx] data is read from initiator FPC and written to remote FPC.

#### **Example 6. ct[xpb\_write/xpb\_read]**

XPB is a management bus for configuring the device's registers. The XPB bus is used within an island to provide a single master with access to the configuration interfaces of many of the internal components.

The XPB is used to access across much of the chip to provide single master with access to the configuration interface of many of the internal components.

xpb\_read and xpb\_write commands instructions support from 1 to 16 individual reads or writes given in the length argument in the instruction.

The address filed provides the following:

- Addr[30] – Global/Local XPB steering bit. 0 : local XPB, 1: Global XPB
- Addr[29:24] – XPB Target Island ID. Island ID of XPB target device. When ‘0’ use own island id when sending XPB command.
- Addr[23:22] – XPB Slave Index. When accessing XPB slave Island, set this to slave's XPB Slave ID.
- Addr[21:16] – XPB Target Device ID. XPB Device ID number of XPB target device.

- Addr[15:2] – XPB Target Register Address. 32-bit word address of the first register to be accessed in the burst. Burst is dependent upon the ref\_cnt or indirect\_ref if greater than 8 words.

This example will show how to Write and read MUSECsrXpb Address MAP CSR registers via the XPB bus.  
(Memory Unit Statistics Engine CSR)

First need to build the XPB write command address with following information:

XPB Target Island ID = 0x1c or 28 + Instance#=0,

XPB Slave Index = 2b00,

XPB Device ID within Island = 0x18 or 24

MUSECsrXpb Address Map offset is 0x00, 0x4, 0x8 and 0xc (32-bit word boundary)

Program bits[18:0] with Base Address for Statistics.

The Global/Local XPB steering bit must be set to allow the FPC 0 to access Internal MU outside the island it belongs to.

This example code will write to the STATS Engine base addresses in burst fashion.

```
; -----
; | FPC 0
; -----

#define XPB_TARGET_ISLAND_ID 0x1c
#define XPB_SLAVE_IDX 0x0
#define XPB_DEVICE_ID 0x18
#define XPB_GLOBAL 0x1

.sig s1 s2
.reg $xfer0 $xfer1 $xfer2 $xfer3
.set $xfer0 $xfer1 $xfer2 $xfer3
.reg_order $xfer0 $xfer1 $xfer2 $xfer3

.reg address
.reg data1
.reg data2
.reg data3
.reg data4

; Load MU SE Base Addresses in local registers
immed[data1, 0x1010]
immed[data2, 0x2020]
immed[data3, 0x3030]
immed[data4, 0x4040]

; Load MU SE Base Addresses in write transfer registers
alu[$xfer0, --, B, data1]
alu[$xfer2, --, B, data2]
alu[$xfer3, --, B, data3]
alu[$xfer4, --, B, data4]

; Setup the ct XPB write command address fields to address MU Statistics Engine MU
; SE Base Address CSRs.
```

```

; XPB Global bit set, MUSECsrXpb MUSEBaseAddr CSRs are located at
; XPB Target Island ID = 0x1c + 0, XPB Slave Index = 0, XPB Device ID within Island = 0x18
; address desire: 0x5c180000
address = (XPB_GLOBAL<<30) | (XPB_TARGET_ISLAND_ID<<24) |
           (XPB_SLAVE_IDX <<22) | (XPB_DEVICE_ID <<16)

; Write four words in Write transfer registers to MUSEBaseAddr CSRs via XPB bus.
ct[xpb_write, $xfer0, address, 0, 4], ctx_swap[s1]

; Read back MUSEBaseAddr CSRs in Read transfer registers
ct[xpb_read, $xfer0, address, 0 4], ctx_swap[s2]

; $xfer0-$xfer3 contain the MuStatsReg 0, 4, 8, C offset values
alu[--, $xfer0, -, 0x1010]
beq[ERROR#]
alu[--, $xfer1, -, 0x2020]
beq[ERROR#]
alu[--, $xfer2, -, 0x3030]
beq[ERROR#]
alu[--, $xfer3, -, 0x4040]
beq[ERROR#]
b[DONE#]

ERROR#:
; handle error condition

DONE#:

```

### **Example 7. ct[interthread\_signal]**

Communicating between FPCs (MEs) often may need a coordination mechanism, and a simple mechanism is to use signalling. At the minimum, one may want to be able to signal, from a given FPC thread, another given FPC thread or threads. The ct[interthread\_signal] can be used by an FPC thread to signal any other FPC thread. In order to signal multiple FPC threads, from a single FPC thread, the ct[interthread\_signal] must be performed once per FPC thread. Each thread can keep track of up to 15 different signals from any combination of sources at a time.

In this example the Cluster Target ct[interthread\_signal] instruction is used to allow FPCs to signal other FPCs using the CPP bus. Four FPCs (FPC4, FPC5, FPC6 and FPC7) thread 1 will participate in a sleep-wakeup-work\\_signal\\_next\\_FPC step process. The signals are the trigger mechanism to get work done in an orderly manner.

It is important to note that in the micro-code the signal assignment must be specified with .addr directive in order to instruct the compiler to manually assign the signal name to the hardware signal number. See example code below. (e.x. ".addr sig1 1" assigns sig1 to hardware signal 1)

The following shows the ct[interthread\_signal] address field definitions that determines which Island, FPC, thread# and signal# to send.

#### **ct[interthread\_signal] Address[31:0] fields definition**

Address[31:0]																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2b0	Island#										11b0										ME (FPC to signal)	Context#	Signal#	2b0							

The address field used with ct[interthread\_signal] specifies the following:

- Island# - Island number of the FPC to signal.
- FPC - FPC within the island to signal.
- Context - Context (thread) within the FPC to signal.
- Signal# - Signal# within the FPC to set.

Refer to NFP-6xxx Databook Table NFP-6xxx Pull IDs for list of Island IDs and Master IDs within Island.

```

/*
 * Setup address field for ct[interthread_signal, --, address, 0, 1]
 *
 *
 * _thread           - Thread number (ctx)
 * _fpcToSignal     - FPC (ME) to signal. (4 - 15)
 * _island          - Island ID to send the signal to. (32-38)
 * _signalNumber    - Signal Number to generate. (1-15)
 * _address         - Output address result value containing the above inputs for
                     ct[interthread_sigal]
 *
 */
#endif

#macro signalFPCSetup(_thread, _fpcToSignal, _island, _signalNumber, _address)
.begin
    alu_shf[_address, --, B, _thread, <<6]           ; thread #
    alu_shf[_address, _address, OR, _fpcToSignal, <<9] ; FPC to signal (FPC 1)
    alu_shf[_address, _address, OR, _island, <<24]    ; Island number of XPB target device
    alu_shf[_address, _address, OR, _signalNumber, <<2] ; signal number 1
.end
#endifm

; Declare signals, Transfer Registers, and GPRs
.sig sig1 sig2 sig3 sig4 sig5 sig6 sig7
.reg $xfer0 $xfer1 $xfer2 $xfer3 $xfer4
.xfer_order $xfer0 $xfer1 $xfer2 $xfer3

; Must use .addr to manually allocate signals. The compiler will assign signal numbers
; to correct FPC signal name. Without .addr usage the signalling name and hardware signal
; may not match.
.addr sig1 1
.addr sig2 2
.addr sig3 3
.addr sig4 4
.addr sig5 5
.addr sig6 6
.addr sig7 7
.addr sig8 8

; Define GPRs
.reg address
.reg thread
.reg fpcToSignal
.reg signalNumber
.reg fpc_num
.reg cl_num
.reg mAddr

```

```
; Run code on thread 1 on selected FPCs, others will
; be put to sleep and never woken.
;if(ctx() == 1)

;-----
; Set Up the BASE scratch address
; Each FPC will use a different region
;-----
; The FPC_NUM is in bits [7:3]
local_csr_rd[ACTIVE_CTX_STS]
immed[fpc_num,0]

alu_shf[cl_num,0x3F, AND,fpc_num,>>25]
alu_shf[fpc_num,0xF, AND,fpc_num,>>3]

; FPC's are numbered starting with 4 for Islands 32-38
; Check each FPC number and branch to it's designated
; code space.
alu[--,fpc_num,-,7]
BEQ[FPC7_code#] ; branch to FPC code space if FPC7, (7-7=0; FPC7)

alu[--,fpc_num,-,6]
BEQ[FPC6_code#] ; branch to FPC code space if FPC6, (6-6=0; FPC6)

alu[--,fpc_num,-,5]
BEQ[FPC5_code#] ; branch to FPC code space if FPC5, (5-5=0; FPC5)

alu[--,fpc_num,-,4]
BEQ[FPC4_code#] ; branch to FPC code space if FPC4, (4-4=0; FPC4)

BR[FPCx_code_NOTUSED#] ; All other FPCs put to sleep and now wakeup

; ****
; FPC4 Code that initializes GPRs and Sends a signal,
; writes to CLS memory value of 1, and waits for receive signal
; from FPC7 (last FPC in chain).
;****

FPC4_code#:
; Pre-initialize GPR registers that do not change throughout
; the work loop.
immed[thread, 1]
alu[fpcToSignal, fpc_num, +, 1]
alu[signalNumber, --, B, fpc_num]
immed[mAddr, 0]
; Call macro to configure GPR address fields
signalFPCSetup(thread, fpcToSignal, cl_num, signalNumber, address)

; Main loop for FPC4
FPC4_MAIN#:
; Assume we have data to do work on from another FPC thread.
; Will perform work on the data
; write result to memory, let's write a value of 1 to cls memory
; for this example.
immed[$xfer0, 1] ; load 1 into $xfer write register
; Write value of 1 to cls memory
cls[write, $xfer0, mAddr, 0, 1], ctx_swap[sig1]
; Send signal (sig4) to FPC5
ct[interthread_signal, --, address, 0, 1]
; Wait for sig7 from last FPC.
```

```
ctx_arb[sig7]
br[FPC4_MAIN#]

; ****
; FPC5 Code initializes GPRs, goes to sleep and is woken when
; FPC4 sends a signal sig4. FPC5 will wake up and performs work:
; Updates CLS memory atomic incr, sends signal sig5 to FPC6
; and then goes to sleep waiting for next signal from FPC4.
;****

FPC5_code#:
; Pre-initialize GPR registers that do not change throughout
; the work loop.
immed[thread, 1]
alu[fpcToSignal, fpc_num, +, 1]
alu[signalNumber, --, B, fpc_num]
immed[mAddr, 0]
; Call macro to configure GPR address fields
signalFPCSetup(thread, fpcToSignal, cl_num, signalNumber, address)

; Main loop for FPC5
FPC5_MAIN#:
; Put to sleep and wakeup when signal sig4 from FPC4 arrives.
ctx_arb[sig4]
; sig4 arrived and FPC5 thread 1 wokeup and performs work on
; data. Perform atomic incr on cls memory location.
cls[incr, --, mAddr, 0]
; Send signal to FPC6 with sig5 (encoded in address initialized
; above).
ct[interthread_signal, --, address, 0, 1]

br[FPC5_MAIN#]

; ****
; FPC6 Code initializes GPRs, goes to sleep and is woken when
; FPC5 sends a signal sig5. FPC6 will wake up and performs work:
; Updates CLS memory atomic incr, sends signal sig6 to FPC7
; and then goes to sleep waiting for next signal from FPC5.
;****

FPC6_code#:
; Pre-initialize GPR registers that do not change throughout
; the work loop.
immed[thread, 1]
alu[fpcToSignal, fpc_num, +, 1]
alu[signalNumber, --, B, fpc_num]
immed[mAddr, 0]
; Call macro to configure GPR address fields
signalFPCSetup(thread, fpcToSignal, cl_num, signalNumber, address)

; Main loop for FPC6
FPC6_MAIN#:
; Go to sleep and wakeup when signal sig5 from FPC5 arrives.
ctx_arb[sig5]
; sig5 arrived and FPC6 thread 1 wokeup and performs work on
; data. Perform atomic incr on cls memory location.
cls[incr, --, mAddr, 0]
; Send signal to FPC7 with signal 6 (encoded in address initialized
; above).
```

```
ct[interthread_signal, --, address, 0, 1]

br[FPC6_MAIN#]

; ****
; FPC7 Code initializes GPRs, goes to sleep and is woken when
; FPC6 sends a signal sig6. FPC7 will wake up and performs work:
; Updates CLS memory atomic incr, sends signal sig7 to FPC4
; and then goes to sleep waiting for next signal from FPC6.
;****

FPC7_code#:
; Pre-initialize GPR registers that do not change throughout
; the work loop.
immed[thread, 1]
alu[fpcToSignal, fpc_num, -, 3]
alu[signalNumber, --, B, fpc_num]
immed[mAddr, 0]
; Call macro to configure GPR address fields
signalFPCSetup(thread, fpcToSignal, cl_num, signalNumber, address)
; Main loop for FPC7
FPC7_MAIN#:
; Put to sleep and wakeup when signal sig6 from FPC6 arrives.
ctx_arb[sig6]
; sig6 arrived and FPC7 thread 1 wokeup and performs work on
; data. Perform atomic incr on cls memory location.
cls[incr, --, mAddr, 0]
; Send signal to FPC4 with sig7 (encoded in address initialized
; above).
ct[interthread_signal, --, address, 0, 1]

br[FPC7_MAIN#]

; Here is where non selected FPC-threads are put to sleep
; because they are not used in this example.
FPCx_code_NOTUSED#:
    ctx_arb[kill]

.endif
ctx_arb[kill]
```

### 2.2.38.1.1 Address field definitions for other Cluster Target commands

#### ct[ctnn\_write] Address[31:0] fields definition

Address[31:0]																															
2b0	Island#				3b0	Data Master ID				Signal				Address Mode	NN Reg Adr				2b0												

- address[29:24] specifies the Island ID of remote FPC to write to
- address[20:17] specifies the FPC within the island to write data to
- address[16:10] specifies the Signal to send
- address[9] specifies the Address Mode of the write. '1' = direct address mode. '0' = FIFO mode
- address[8:2] specifies the first register address to write to. Ignored if Address Mode is 'FIFO mode'.

#### ct[ring\_put/ring\_get] Address[31:0] fields definition

Address[31:0]																																
2b0	Island#								18b0								Ring Number				2b0											

- address[29:24] specifies the Island ID to access
- address[5:2] specifies the CTM Ring to access.

#### ct[xpb\_write/xpb\_read] Address[31:0] fields definition

Address[31:0]																																		
1b0	Global	Island#				XPB Slave ID	XPB Device ID				Address				2b0																			

- address[30] specifies type of XPB access (1: Global XPB; 0: normal XPB).
- address[29:24] specifies the Island ID to access
- address[23:22] specifies the Slave ID of the XPB Target to access
- address[21:16] specifies the XPB Device ID of the XPB Target to access
- address[15:2] specifies the Address to access within the XPB Target device.

## 2.2.39 Memory Unit (MU)

There are three different types of Memory Unit in the NFP-6xxx architecture:

- CTM (Cluster Target Memory) has: bulk, atomic, Misc, and packet engines.
- IMU (Internal Memory Unit) has: bulk, atomic, stats, load balance, and lookup engines.
- EMU (External Memory Unit) has: bulk, atomic, queue, and lookup engines.



### Important

All Memory Units in the system respond to the same target ID in the CPP command. Addressing a specific Memory Unit requires addressing the correct island in addition to the target ID. (There is only one Memory Unit per island.)



### Note

When reading the last 64 bit word in DRAM memory space, you must use an 64-bit-aligned read. DRAM reads that go beyond the end of physical memory, can cause cache corruption in later, unrelated, DRAM transactions.



### Note

32-bit addressing mode is supported only for Bulk, Atomic and Queue operations. Attempting to address any other type of operation, for instance Packet Engine commands, in 32-bit addressing mode is not supported and results are unpredictable.

### 2.2.39.1 MEM (Atomic Operations)

Logic and arithmetic commands.

#### Instruction Formats

```
mem[cmd, --, src_op1, src_op2]    // 32-bit addressing
mem[cmd, --, src_op1, <<8, src_op2] // 40-bit addressing
mem[cmd, --, src_op1, src_op2, <<8] // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt_tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt_tok // 40-bit addressing
```

The following table lists the instructions that have been aliased to other instructions.

#### Aliased Instructions

Command	Description
add_imm_sat	See: addsat_imm

Command	Description
add_sat	See: addsat
add64_imm_sat	See: addsat64_imm
add64_sat	See: addsat64
compare_write (CmpAndWr)	See: compare_write_or_incr
compare_write (CmpAndWr32)	See: mask_compare_write
read_atomic	See: atomic_read
sub_imm_sat	See: subsat_imm
sub_sat	See: subsat
sub64_imm_sat	See: subsat64_imm
sub64_sat	See: subsat64
test_and_add	See: test_add
test_and_add_imm	See: test_add_imm
test_and_add_imm_sat	See: test_addsat_imm
test_and_add_sat	See: test_addsat
test_and_add64	See: test_add64
test_and_add64_imm	See: test_add64_imm
test_and_add64_imm_sat	See: test_addsat64_imm
test_and_add64_sat	See: test_addsat64
test_and_clr	See: test_clr
test_and_clr_imm	See: test_clr_imm
test_and_compare_write (CmpAndWr)	See: test_compare_write_or_incr.
test_and_compare_write (CmpAndWr32)	See: test_mask_compare_write.
test_and_set	See: test_set
test_and_set_imm	See: test_set_imm
test_and_sub	See: test_sub
test_and_sub_imm	See: test_sub_imm
test_and_sub_imm_sat	See: test_subsat_imm
test_and_sub_sat	See: test_subsat
test_and_sub64	See: test_sub64
test_and_sub64_imm	See: test_sub64_imm
test_and_sub64_imm_sat	See: test_subsat64_imm
test_and_sub64_sat	See: test_subsat64
test_and_xor	See: test_xor
test_and_xor_imm	See: test_xor_imm
write_atomic	See: atomic_write

Command	Description
write_atomic_imm	See: atomic_write_imm



## Atomic Cache Line Restriction

For all Atomic commands, when specifying address and ref\_cnt of the command, address + (ref\_cnt - 1) must not wrap a 64-byte cache line boundary. In other words, you must insure that all memory words accessed reside within the same cache line. Writes that exceed a cache line boundary are likely to cause corruption to neighboring cache lines in the memory unit cache, which in turn might map to unrelated and unpredictable physical (external) memory addresses. The read-modify-write nature of the atomic operations combined with the fact that the neighboring cache line may be evicted and replaced means that such corruptions are not limited to the data written by the atomic engine and may manifest as 4 word chunks of data being copied from one memory location to another.

## Parameter Descriptions

Param	Command	Description
cmd	add	Add the 2s complement value in the transfer registers to the data at the address, which is treated as an unsigned number. Negative number addition resulting in a value of < 0, will roll over as a positive number. Positive number addition with results of > 0xFFFF FFFF will roll over. See <a href="#">Note 8</a> .
	add_imm	Same as the add instruction but the data to be added to memory is specified as the immediate data and the operation is applied only to a single 32-bit memory word. See <a href="#">Note 4</a> .
	add64	Add the 2s complement of the 64-bit value in the transfer registers to the data at the address, which is treated as an unsigned number. See <a href="#">Note 8</a> .
	add64_imm	Same as the add64 instruction but the data to be added to memory is specified as the immediate data and the operation is applied only to a 64-bit memory word. See <a href="#">Note 4</a> .
	addsat	Add the signed value in the transfer registers to the data at the address. Addition with results of < 0 will saturate the result in the value in memory at 0x0000 0000. Positive number addition resulting in a value of > 0xFFFF FFFF, will saturate the result at 0xFFFF FFFF. See <a href="#">Note 8</a> .
	addsat_imm	Add the signed value in the immediate data to the single 32-bit memory word at the address. Addition with results of < 0 will saturate the result in the value in memory at 0x0000 0000. Positive number addition resulting in a value of > 0xFFFF FFFF, will saturate the result at 0xFFFF FFFF. See <a href="#">Note 4</a> .
	addsat64	Add the signed 64-bit value in the transfer registers to the data at the address. Addition with results of < 0 will saturate the result in the value in memory at 0x0000 0000 0000.

Param	Command	Description
		Positive number addition resulting in a value of > 0xFFFF FFFF FFFF FFFF, will saturate the result at 0xFFFF FFFF FFFF FFFF. See <a href="#">Note 8</a> .
	addsat64_imm	Add the signed 64-bit value in the immediate data to the data at the address. Addition with results of < 0 will saturate the result in the value in memory at 0x0000 0000 0000 0000. Positive number addition resulting in a value of > 0xFFFF FFFF FFFF FFFF, will saturate the result at 0xFFFF FFFF FFFF FFFF. See <a href="#">Note 4</a> . Same as the addsat64 instruction but the data to be added to memory is specified as the immediate data and the operation is applied only to a 64-bit memory word. See <a href="#">Note 4</a> .
	atomic_read	Read atomically 32-bit words to the specified address.
	atomic_write	Write atomically 32-bit words in transfer registers to the specified address.
	atomic_write_imm	Write atomically 32-bit words in immediate data to the specified address. See <a href="#">Note 3</a> .
	clr	Clear bits specified in write transfers registers to a number of 32-bit memory words. A byte mask specifies which memory bytes are affected for all memory words.
	clr_imm	Same as the clr instruction but the bit mask is specified as the immediate data. See <a href="#">Note 3</a> .
	compare_write_or_incr	Compare bottom 16 bits of the first write transfer register with the bottom 16 bits of memory location. If they are equal, set the bottom 16 bits of memory to top 16 bits of the transfer register and write the remaining write registers to the subsequent memory locations. If the compare fails, increment the top 16 bits of memory location saturating at 0xffff.
	decr, decr64	Single instruction decrement for a memory location. The decr commands are implemented as add_imm with an immediate of 0xff, ie. -1. If ref_cnt is provided in the instruction, it directly specifies how many consecutive memory locations (starting from the provided address) to decrement by 1. decr for 32-bit value, and decr64 for 64-bit value in memory.
	incr, incr64	Single instruction increment for a memory location. The incr commands are implemented as sub_imm with an immediate of 0xff, ie. -1. If ref_cnt is provided in the instruction, it directly specifies how many consecutive memory locations (starting from the provided address) to increment by 1. incr for 32-bit value, and incr64 for 64-bit value in memory.
	meter	Perform the metering operation. See to <a href="#">Note 7</a> .
	mask_compare_write	Read from one up to eight 32-bit words in memory starting at supplied address. Compare each word to a corresponding

Param	Command	Description
		write transfer register that has one or more byte-mask bits set (4-bits). If a match is found for one or more words the corresponding memory location(s) is overwritten with the corresponding write transfer register word(s); otherwise if there is no match return unmodified data back to memory. Note that for data to be written, at least one byte in the 32-bit word needs to be compared. Byte mask values of 0 and all 1's are reserved. This command requires indirect_ref usage to override length and byte-mask. See <a href="#">Note 1</a> .
	set	Set bits specified in write transfers registers to a number of 32-bit memory words. A byte mask specifies which memory bytes are affected for all memory words.
	set_imm	Same as the set instruction but the bit mask is specified as the immediate data. See <a href="#">Note 3</a> .
	sub	Subtract the 2s complement value in the transfer registers from the data at the address, which is treated as an unsigned number. See <a href="#">Note 8</a> .
	sub_imm	Same as the sub instruction but the data to be added to memory is specified as the immediate data and the operation is applied only to a single 32-bit memory word. See <a href="#">Note 4</a> .
	sub64	Subtract the 2s complement 64-bit value in the transfer registers from the data at the address, which is treated as an unsigned number. See <a href="#">Note 8</a> .
	sub64_imm	Same as the sub64 instruction but the data to be added to memory is specified as the immediate data. See <a href="#">Note 4</a> .
	subsat	Subtract the unsigned 32-bit value in the transfer registers from the data at the address. Subtraction with results of < 0 will saturate the result in the value in memory at 0x0000 0000. See <a href="#">Note 8</a> .
	subsat_imm	Subtract the unsigned 32-bit value in the immediate data from the single 32-bit memory word at the address. Subtraction with results of < 0 will saturate the result in the value in memory at 0x0000 0000. See <a href="#">Note 4</a> .
	subsat64	Subtract the unsigned 64-bit value in the transfer registers from the data at the address. Subtraction with results of < 0 will saturate the result in the value in memory at 0x0000 0000. See <a href="#">Note 8</a> .
	subsat64_imm	Subtract the unsigned 64-bit value in the immediate data from the data at the address. Subtraction with results of < 0 will saturate the result in the value in memory at 0x0000 0000. See <a href="#">Note 4</a> .
	swap	Same as the atomic_write instruction but it also returns premodified memory values in the read transfer registers.

Param	Command	Description
	swap_imm	Same as the swap instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 3</a> .
	test_add	Same as the add instruction but it also returns the premodified memory value in the read transfer register. See <a href="#">Note 8</a> .
	test_add_imm	Same as the add_imm instruction but it also returns the premodified memory value in the read transfer register. See <a href="#">Note 6</a> .
	test_addsat	Same as the addsat instruction but it also returns the premodified memory value in the read transfer register. See <a href="#">Note 8</a> .
	test_add64	Same as the add64 instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_add64_imm	Same as the add64_imm instruction but it also returns the premodified memory value in the read transfer register. See <a href="#">Note 6</a> .
	test_addsat_imm	Same as the addsat_imm instruction but it also returns the premodified memory values in the read transfer register. See <a href="#">Note 6</a> .
	test_addsat64	Same as the addsat_64 instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_addsat64_imm	Same as the addsat64_imm instruction but it also returns the premodified memory value in the read transfer register. See <a href="#">Note 6</a> .
	test_clr	Same as the clr instruction but it also returns the premodified memory values in the read transfer registers.
	test_clr_imm	Same as the clr_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 5</a> .
	test_compare_write_or_incr	Same as the compare_write_or_incr instruction but it also returns a pre-modified 32-bit word of the first memory location.
	test_mask_compare_write	Same as the mask_compare_write instruction but it also returns pre-modified memory values in the read transfer registers.
	test_set	Same as the set instruction but it also returns the premodified memory values in the read transfer registers.
	test_set_imm	Same as the set_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 5</a> .

Param	Command	Description
	test_sub	Same as the sub instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_sub_imm	Same as the sub_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 6</a> .
	test_sub64	Same as the sub64 instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_sub64_imm	Same as the sub64_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 6</a> .
	test_subsat	Same as the subsat instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_subsat_imm	Same as the subsat_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 6</a> .
	test_subsat64	Same as the subsat64 instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 8</a> .
	test_subsat64_imm	Same as the subsat64_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 6</a> .
	test_xor	Same as the xor instruction but it also returns the premodified memory values in the read transfer registers.
	test_xor_imm	Same as the xor_imm instruction but it also returns the premodified memory values in the read transfer registers. See <a href="#">Note 5</a> .
	xor	Do an exclusive OR operation between write transfer registers and a number of 32-bit memory words. A byte mask specifies which memory bytes are affected for all memory words.
	xor_imm	Same as the xor instruction but the bit mask is specified as the immediate data. See <a href="#">Note 3</a> .
xfer	Logical and arithmetic non-immediate instructions without readback: add, add64, addsat, addsat64, atomic_read, atomic_write, clr, compare_write, compare_write_or_incr, mask_compare_write, set, sub, sub64, subsat, subsat64, xor	Transfer write registers hold data modifiers.
	Logical and arithmetic non-immediate instructions with readback: swap,	Transfer write registers hold data modifiers and transfer read registers to hold the premodified data.

Param	Command	Description
	test_add, test_addsat, test_add64, test_addsat64, test_clr, test_clr, test_compare_write_or_incr, test_mask_compare_write, test_set, test_set, test_sub, test_sub64, test_sub64, test_subsat, test_subsat, test_subsat64, test_subsat64, test_xor	
	All immediate instructions without readback: add_imm, add64_imm, addsat_imm, addsat64_imm, atomic_write_imm, clr_imm, set_imm, sub_imm, sub64_imm, subsat_imm, subsat64_imm, xor_imm	Not Required. See <a href="#">Section 2.1.6.3</a>
	meter	A write transfer register contains a 32-bit data. A read transfer register contains a 32-bit data.
	Relational and swap immediate instructions with readback: swap_imm, test_clr_imm, test_set_imm, test_sub_imm, test_sub64_imm, test_subsat_imm, test_subsat64_imm, test_xor_imm	Transfer read registers to hold the premodified data.
	Arithmetic immediate instructions with readback: test_add_imm, test_add64_imm, test_add64_imm, test_addsat_imm, test_addsat64_imm, test_addsat64_imm, test_sub_imm, test_sub64_imm, test_subsat_imm, test_subsat64_imm	A read transfer register that holds the premodified data.
	decr, decr64, incr, incr64	Must be omitted; should be “--”.
src_op1 and src_op2	All commands	Restricted source operands that define 32 or 40-bit byte memory address.
ref_cnt	atomic_read, atomic_write, atomic_write_imm, clr, clr_imm, compare_write, compare_write_or_incr, mask_compare_write, set, set_imm, swap, swap_imm, test_clr, test_clr_imm, test_compare_write_or_incr, test_mask_compare_write, test_set, test_set_imm, test_xor, test_xor_imm, xor, xor_imm	Reference count in increments of 4 byte words. Valid values are 1 to 8. See <a href="#">Atomic Cache Line Restriction</a> .

Param	Command	Description
	32-bit add and sub non-immediate operations: add, addsat, sub, subsat, test_add, test_addsat, test_sub, test_subsat	Reference count in increments of 4 byte words. Valid values are 1 to 4. See <a href="#">Atomic Cache Line Restriction</a> .
	64-bit add and sub non-immediate operations: add64, addsat64, sub64, subsat64, test_add64, test_addsat64, test_sub64, test_subsat64	Reference count in increments of 8 byte words. Valid values are 1 to 4. See <a href="#">Atomic Cache Line Restriction</a> .
	32 immediate add and sub operations: add_imm, addsat_imm, sub_imm, subsat_imm, test_add_imm, test_addsat_imm, test_sub_imm, test_subsat_imm	Reference count in increments of 4 byte words. Valid values are 1 to 4. See <a href="#">Atomic Cache Line Restriction</a> .
	64-bit immediate add and sub operations: add64_imm, addsat64_imm, sub64_imm, subsat64_imm, test_add64_imm, test_addsat64_imm, test_sub64_imm, test_subsat64_imm	Reference count in increments of 8 byte words. Valid values are 1 to 4. See <a href="#">Atomic Cache Line Restriction</a> .
	decr, decr64, incr, incr64	Reference count value, when provided, is directly loaded into the CPP LENGTH field for specifying how many memory locations to apply the command operation. Valid values are 1 to 4. When ref_cnt is not specified ref_cnt value is 1. See <a href="#">Atomic Cache Line Restriction</a> .
opt_tok	All non-immediate instructions without readback: add, add64, addsat, addsat64, atomic_read, atomic_write, clr, compare_write, compare_write_or_incr, mask_compare_write, set, sub, subsat, sub64, subsat64, xor	sig_done[sig_name] ctx_swap[sig_name] indirect_ref (See <a href="#">Note 2</a> ); refer to <a href="#">Section 2.1.6.4.2</a> . defer[n] where n = 1 or 2 ind_targets[me1, me2, ...]
	All immediate instructions without readback: add_imm, add64_imm, addsat_imm, addsat64_imm, atomic_write_imm, clr_imm, set_imm, sub_imm, sub64_imm, subsat_imm, subsat64_imm, xor_imm	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2, ...]
	decr, decr64, incr, incr64	Must be omitted; should be “--”.
	All instructions with readback: swap, swap_imm, test_add, test_add_imm, test_addsat, test_add64, test_add64_imm, test_addsat_imm, test_addsat64, test_addsat64_imm, test_clr, test_clr_imm,	sig_done[sig_name] indirect_ref (See <a href="#">Note 2</a> ); refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2, ...]

Param	Command	Description
	test_compare_write_or_incr, test_mask_compare_write, test_set, test_set_imm, test_sub, test_sub_imm, test_sub64, test_sub64_imm, test_subsat, test_subsat_imm, test_subsat64, test_subsat64_imm, test_xor, test_xor_imm  meter	

**Notes:**

1. The mask\_compare\_write command compares 4 bytes for equality under the 4 bit byte mask control.

Bits of byte mask				Bytes to compare
0	0	0	0	No bytes are compared. Comparison always fails.
0	0	0	1	7:0
0	0	1	0	15:8
0	0	1	1	7:0, 15:8
0	1	0	0	23:16
0	1	0	1	7:0, 23:16
0	1	1	0	15:8, 23:16
0	1	1	1	7:0, 15:8, 23:16
1	0	0	0	31:24
1	0	0	1	7:0, 31:24
1	0	1	0	15:8, 31:24
1	0	1	1	7:0, 15:8, 31:24
1	1	0	0	23:16, 31:24
1	1	1	0	15:8, 23:16, 31:24
1	1	1	1	31:0

2. The indirect\_ref token is required to invoke the mask\_compare\_write and test\_compare\_write\_or\_incr. Ensure that the length on the bus is in the range 8 to 15. Recommend using V2 Indirect Reference format, section [Section 2.1.6.4.6](#).
3. For immediate atomic operations, the immediate data is specified in the indirect field definitions. There are two options: immediate data in data\_master and data\_ref or immediate data in byte\_mask. The V2 Indirect Reference format is recommended over the V1 Indirect Reference mode.

### V2 Indirect Reference Mode:

Depending on the setting of the length field, 16-bit immediate data can be specified in the byte mask field or in the combined data master and data reference fields.

- LENGTH[3] = 0 : When bit [3:3] of the length field is clear, the immediate data is represented by bits [7:0] of the byte mask field. Byte mask field would be sign extended to form the immediate data.
- LENGTH[3] = 1 : With 16-bit immediate data in data\_master, data\_ref (LENGTH[3] = 1), the LENGTH[2:0] specifies the number of 32 bit words. For V2 Indirect Reference mode using PREV\_ALU is explained below.

In the PREV\_ALU, OVE\_DATA = 2 will select Overrides 16-bit immediate data. PREV\_ALU\_LENGTH[3] = 1 and PREV\_ALU LENGTH[2:0] field determines the number of 32/64 bit words from LENGTH[2:0].

Note: V2 Indirect Reference mode shows the limitation of the V1 Indirect Reference mode when it comes to 16-bit immediate; here one can override the length, data\_master and data\_reference. Example code is shown below.

```
immed[tmp,((1<<(3+8))|((REF_CNT-1)<<8)|(1<<7)|(2<<3))]
alu[--,tmp, OR, data, <<16] ; use 16-bits from register data
mem[set_imm, --, src1, <<8, src2], indirect_ref
Note: ref_cnt defaults to 1 and overridden by indirect_ref
or
mem[set_imm, --, src1, <<8, src2, REF_CNT], indirect_ref
NOTE: REF_CNT will be overridden to set bit 3 of length.
```

8-bit immediate data in byte\_mask:

- PREV\_ALU OVE\_DATA = 6 will select Overrides byte mask from DATA16. DATA16 is loaded with data byte. It is not necessary to set LENGTH[3] for immediate data in byte\_mask. It is not necessary to override the length field LENGTH[2:0] as long as ref\_cnt is constant (1..8). Example code is shown below.

```
alu[--, (6<<3), OR, data, <<16]
mem[set_imm, --, src1, <<8, src2, REF_CNT], indirect_ref
```

- (PREV\_ALU OV\_BM\_CSR = 1), CmdIndirectRef0.BYTE\_MASK[7:0] is loaded with data byte. It is not necessary to set LENGTH[3] for immediate data in byte\_mask. It is not necessary to override the length field LENGTH[2:0] as long as ref\_cnt is constant (1..8). Example code is shown below.

```
local_csr_wr[CMD_INDIRECT_REF_0, data]
alu[--, --, B, (1<<6)] ; Set OVE_BM_CSR
mem[set_imm, --, src1, <<8, src2, REF_CNT], indirect_ref
```

In V2 Indirect Reference mode other fields may also be overridden, like the island, which may also make one option more efficient than the another.

### V1 Indirect Reference Mode:

Depending on the setting of the length field, it can be in the byte mask or combined data master and data reference fields. When bit [3:3] of the length field is clear, the immediate data is represented by bits [7:0] of the byte mask field. In the example below the immediate value of 0x5B is used. Reference count is specified in bits [1:0] of the length field. The reference count of (2+1) is used.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoding	Reserved										Length	Immediate data																			
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	1	0	1	1

When the bit [3:3] of the length field is set, the immediate data is represented by data reference or combination of data master and data reference field. In the example below the immediate value of 0xCD is used and the reference count is set to (6+1).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoding	Reserved										Immediate data										Length										
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	0	1	1	1	0	

4. For immediate arithmetic operations without read back, bit [2:2] of the length field has to be cleared (0) for 32 bit operations and set (1) for 64 bit operations. Also refer to Note 3 above.
5. For immediate non-arithmetic operations with read back of premodified memory value (e.g. test\_set\_imm), the immediate value can be only in the byte mask field of the indirect reference. Bit [3] of the length field always must be set to 0. Also refer to Note 3 above.
6. For immediate arithmetic operations with read back of premodified memory value (e.g. test\_add\_imm), the immediate value can be only in the byte mask field of the indirect reference. Bit [3] of the length field must always be set to 0. Bit [2:2] has to be cleared for 32 bit operations and set for 64 bit operations. Also refer to Note 3 above.
7. Atomic metering command meters the IP Packet stream and marks its packets either Red(2b10), Yellow(2b01) or Green(2b00). It operates within a single 128-bit portion of a cache-line. There are two different operating modes for the command, Color-blind mode and Color-aware mode. Color-blind mode is effectively Color-aware mode with Green color.

Following is required for Metering Instruction:

- Address is 16-byte aligned address.
- Address[0] selects RFC type: 0 == RFC2698, 1 == RFC2697/4115.
- Address[2:1] indicates incoming packet color: 2b00 == Green, 2b01 == Yellow, 2b10 == Red
- Address bit[3] selects either Meter-pair0 (64-bits) or Meter-pair1 (64-bits) contents of the 128-bit cache-line. Which gives a Long-term credit value (32-bits) and Short-term credit value (32-bits) unsigned.
- Write transfer register containing a 32-bit unsigned value (packet size in bytes) will be subtracted from the selected Meter-pair Long-term credit value and Short-term Credit value. The subtracted values, Decremented Long-term Credit value and Decrement Short-term Credit value, are further used by the Meter algorithm to provide the final Metering result in the read transfer register.

The Metering algorithm is explained in the NFP-6xxx Databook. The Meter instruction result is 32-bits data in read Transfer register indicating either 2b00 (Green), 2b01 (Yellow) or 2b10 (Red) in bits[1:0]. The remaining bits are don't care.

8. For non-immediate arithmetic operations with or without readback of premodified memory value (e.g. test\_add64), bit [3] of the length field must always be set to 0. Bit [2:2] has to be cleared (0) for 32 bit operations and set (1) for 64 bit operations.

## 2.2.39.2 MEM (CAM and TCAM lookup)

These command instructions allow the users to do two types of operations on CAM and TCAM – add new entries and lookup existing entries. These operations can be executed on full 32-bit word, bottom 24 bit, 16 bit, or 8 bit of each 32-bit word of CAM and TCAM. There can be four sizes of CAM and TCAM – 128, 256, 384, and 512 bits.

The CAM and TCAM lookup commands utilize the pull data (write xfer registers) and compares it with every specified size word in the specified memory range. If a match is found the push data will hold match information (See NFP-6xxx Databook), and if no match is found the push data will be equal to 0x0000\_00ff.

### Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	cam128_lookup8, cam256_lookup8, cam384_lookup8, cam512_lookup8	8-bit CAM lookup. The instruction matches the lower 8 bits of a write transfer register with each CAM entry in big endian order. A 64-bit map is returned where each N bit corresponds to a matched N byte in CAM. See <a href="#">Note 1</a> .
	cam128_lookup16, cam256_lookup16, cam384_lookup16, cam512_lookup16	16-bit CAM lookup. The instruction matches the lower 16 bits of a write transfer register with each CAM entry in big endian order. A 64-bit map is returned where each 32+N bit corresponds to a matched N 16-bit word in CAM. Bits 0-7 in the lower part of bit map contains the lowest match entry number. On lookup failure 0x0000 00ff 0000 0000 is returned. See <a href="#">Note 1</a> .
	cam128_lookup24, cam256_lookup24, cam384_lookup24, cam512_lookup24	24-bit CAM lookup. The instruction matches the lower 24 bits of a write transfer register with each lower 24 bit part of 32-bit CAM entry in big endian order. A 32-bit map is returned where each 16+N bit corresponds to a matched N 32-bit word in CAM. Bits 0-7 of the bit map contains the lowest match entry number. Bits 8-16 contain bits 24-32 of the matching memory data. On match failure 0x0000 00ff is returned.
	cam128_lookup32, cam256_lookup32, cam384_lookup32, cam512_lookup32	32-bit CAM lookup. The instruction matches 32 bits of a write transfer register with 32-bit CAM entries in big endian order. A 32-bit map is returned where each 16+N bit corresponds to a matched N 32-bit word in CAM. Bits 0-7 of the bit map contains the lowest match entry number. Bits

Parameter	Command	Description
		8-16 contain bits 24-32 of the matching memory data. On match failure 0x0000 00ff is returned.
	cam128_lookup8_add, cam256_lookup8_add, cam384_lookup8_add, cam512_lookup8_add	8-bit CAM lookup with new entry addition. The instruction matches the lower 8 bits of a write transfer register with each CAM entry in big endian order. A 64-bit map is returned where each N bit corresponds to a matched N byte in CAM. If CAM match is not found, the entry is added to CAM. If empty slot is not found 0x0000 00ff 0000 0000 is returned.
	cam128_lookup16_add, cam256_lookup16_add, cam384_lookup16_add, cam512_lookup16_add	16-bit CAM lookup with new entry addition. The instruction matches the lower 16 bits of a write transfer register with each CAM entry in big endian order. A 64-bit map is returned where each 32+N bit corresponds to a matched N 16-bit word in CAM. Bits 0-7 in the lower part of bit map contains the lowest match entry number. If a CAM match is not found, the entry is added to CAM. If empty slot is not found 0x0000 00ff 0000 0000 is returned.
	cam128_lookup24_add, cam256_lookup24_add, cam384_lookup24_add, cam512_lookup24_add	24-bit CAM lookup with new entry addition. The instruction matches the lower 24 bits of a write transfer register with each lower 24 bit part of 32-bit CAM entry in big endian order. A 32-bit map is returned where each 16+N bit corresponds to a matched N 32-bit word in CAM. Bits 0-7 of the bit map contains the lowest match entry number. Bits 8-16 contain bits 24-32 of the matching memory data. If a CAM match is not found, the entry is added to CAM. If empty slot is not found 0x0000 00ff is returned.
	cam128_lookup32_add, cam256_lookup32_add, cam384_lookup32_add, cam512_lookup32_add	32-bit CAM lookup with new entry addition. The instruction matches 32 bits of a write transfer register with 32-bit CAM entries in big endian order. A 32-bit map is returned where each 16+N bit corresponds to a matched N 32-bit word in CAM. Bits 0-7 of the bit map contains the lowest match entry number. Bits 8-16 contain bits 24-32 of the matching memory data. If a CAM match is not found, the entry is added to CAM. If empty slot is not found 0x0000 00ff is returned.
	cam128_lookup24_add_inc	Similar to cam128_lookup24_add but if a match is found, the unused top 8 bits in memory are used as a match count.
	cam256_lookup24_add_inc	Similar to cam256_lookup24_add but if a match is found, the unused top 8 bits in memory are used as a match count.
	cam384_lookup24_add_inc	Similar to cam384_lookup24_add but if a match is found, the unused top 8 bits in memory are used as a match count.
	cam512_lookup24_add_inc	Similar to cam512_lookup24_add but if a match is found, the unused top 8 bits in memory are used as a match count.

Parameter	Command	Description
	cam_lookup24_add_inc	Perform a 24 bit CAM lookup on a 384 or 512 bit CAM and if a entry is not found, add it. If an entry was found, increment the count field for the entry. The indirect ref method must be employed (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	cam_lookup24_add_extend	See cam_lookup_add_extend.
	cam_lookup24_add_lock	See cam_lookup_add_lock.
	cam_lookup	Perform a 8, 16, 24 or 32 bit CAM lookup on a 384 or 512 bit CAM. The indirect ref method must be employed (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	cam_lookup_add	Perform a 8, 16, 24 or 32 bit CAM lookup on a 384 or 512 bit CAM and if a entry is not found, pull data is automatically add to the CAM data if there is an empty slot. Empty slot is when CAM entry is zero. The indirect ref method must be employed (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	cam_lookup_add_extend	This instruction is small extension to cam_lookup_add_lock, meaning that if no match is found and there is no room for an entry in the CAM then set bit[31] of the first memory location used for the CAM command. It will also push back the PREVIOUS value of bit[31] of the first memory location in bit[15] of the push data (which is normally zero). The indirect ref method must be employed (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	cam_lookup_add_inc	Perform a 24 bit CAM lookup on a 384 or 512 bit CAM and if a entry is not found, pull data is automatically add to the CAM entry if there is an empty slot. Empty slot is when CAM entry is zero. If an match is found increment the (non-saturating) usage count field for that entry (top 8 bits). The indirect ref method must be employed (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	cam_lookup_add_lock	Perform a 32 bit CAM lookup on a 384 or 512 bit CAM. This command is effectively an extension of CAM32 lookup and add. It uses a 31-bit CAM value (top bit is ignored). bit[31] of Pull-data should always be one. If a CAM match is found for an entry (i.e. the pulled 32-bits of data have the bottom 31 bits matching a memory location) then the top bit of the matching memory location is set indicating that the entry is now locked. The pre-modified memory data top byte is returned in the push data as with other CAM32 operations (in bits [8;8]) so that the software can tell if the location was already locked. If a CAM match is not found for an entry then an empty slot is searched for (contents of all zero), and the new entry is added at

Parameter	Command	Description
		that empty slot with the top bit of the memory location set indicating that the entry is now locked. If a CAM match is not found and there is no room for the entry then as with normal CAM32-and-add operations, no memory update is performed and 0x0000_00ff in the push data. (See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> ).
	tcam_lookup	Search an area of memory using 32-bit write transfer register. Using indirect_ref specify in Length field the Lookup length (8, 16, 24 or 32 bit lookup) and lookup bit length (128-bit to 512-bit lookup)
	tcam128_lookup8, tcam256_lookup8, tcam384_lookup8, tcam512_lookup8	An 8-bit TCAM match pulls 32-bits of data and compares bits [7:0] of that pull data with successive bytes from even 32-bit word memory locations in the specified memory range, using masks from odd word memory locations. For each match found (pull data & mask == memory & mask) at byte 'n' (big endian byte offset from search address), then bit 32+(((n<<3)>>2)&(n&3)) of the push data will be set. The lowest 'n' for which this is true, is returned in bits [7:0]. If no match is found, then the result 0x0000_0000_0000_00ff will be returned. See <a href="#">Note 2</a> .
	tcam128_lookup16, tcam256_lookup16, tcam384_lookup16, tcam512_lookup16	A 16-bit TCAM match pulls 32-bits of data and compares bits [15:0] of that pull data with successive half-words from even 32-bit word memory locations in the specified memory range, using masks from odd word memory locations. For each match found (pull data & mask == memory & mask) at half-word 'n'(big endian half-word offset from search address), then bit 16+(((n<<2)>>1) (n&1)) of the push data will be set. The lowest 'n' for which this is true, is returned in bits [7:0]. If no match is found, then the result 0x0000_00ff will be returned. See <a href="#">Note 2</a> .
	tcam128_lookup24, tcam256_lookup24, tcam384_lookup24, tcam512_lookup24	A 24-bit TCAM match pulls 32-bits of data and compares bits [23:0] of that pull data with bits [23:0] of successive even 32-bit word memory locations in the specified memory range, using masks from odd word memory locations. For each match found (pull data & mask == memory & mask) at word 'n' (big endian word offset from search address), then bit 16+(n/2) of the push data will be set. The lowest 'n' for which this is true, is returned in bits [7:0]. Bits [15:8] will be taken from bits [31:24] of the matching data. If no match is found, then the result 0x0000_00ff will be returned. See <a href="#">Note 2</a> .
	tcam128_lookup32, tcam256_lookup32, tcam384_lookup32, tcam512_lookup32	A 32-bit TCAM match pulls 32-bits of data and compares bits [31:0] of that pull data with successive 32-bit words from even 32-bit word memory locations in the specified memory range, using masks from odd word memory

Parameter	Command	Description
		locations. For each match found, (pull data & mask == memory & mask) at word 'n' (big endian word offset from search address), then bit 16+(n/2) of the push data will be set. The lowest 'n' for which this is true, is returned in bits [7:0]. Bits [15:8] will be taken from bits [31:24] of the matching data. If no match is found, then the result 0x0000_00ff will be returned. See <a href="#">Note 2</a> .
xfer	All CAM 8 and 16 bit lookup instructions.  All TCAM 8 bit lookup instructions.  cam_lookup, cam_lookup_add_extend, cam_lookup_add_inc, cam_lookup_add_lock, cam_lookup24_add_extend, cam_lookup24_add_lock, tcam_lookup	A write transfer register holding matching data. Two read transfer registers containing the result of operation which is either bit map or 0x0000 0000 0000 00ff on failure.
	All CAM 24 and 32 bit lookup instructions.  All TCAM 16, 24, and 32 bit lookup instructions.	A write transfer register holding matching data. A read transfer register containing result of operation as described for each CAM or TCAM operation or 0x0000 00ff on failure.
src_op1, src_op2	All commands	Restricted source operands that define 40-bit byte CAM or TCAM address.
opt Tok	All commands, except 384 and 512 CAM and TCAM operations	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]
	All 384 and 512 CAM and TCAM commands	sig_done[sig_name]
		indirect_ref(required). See <a href="#">Note 3</a> and refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]

#### Notes:

- Example of 8-bit lookup, 128-bit CAM. CAM Entries # 4 and 9 are matched.

First row below is bits[31:0] and second row is bits[63:32]. Since match was found for entries #4 and #9, then bit[4] and bit[9] are set in the returned data (read transfer registers).

64 bit map returned in the read transfer registers																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

128-bit CAM, 8 bit entries, total number of entries - 16:

Entry # 0
Entry # 1
...
Entry # 4 (matched)
Entry # 9 (matched)
Entry # 15

Example of 16-bit lookup, 512 bit CAM. Entries 10, 14,31 are matched.

First row below is bits[31:0] and second row is bits[63:32]. In the returned 64 bit map, bits 0-7 in the first word contain the number of the lowest match entry (entry #10). The second 32-bit word (bit-map) has bits set corresponding to matched CAM entries. Matched entries #10, #14, #31 are indicated by set bits in second row bit[31] bit[14] and bit[10] respectively.

64 bit map returned in the read transfer registers																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

512-bit CAM, 16 bit entries, total number of entries – 32:

Entry # 0
Entry # 1
...
Entry # 10 (matched)
...
Entry # 14 (matched)
...
Entry # 31 (matched)

2. 512-bit TCAM, 32-bit entries. Total number of TCAM entries is 16.

32-bit map returned in the read transfer register		
Bit mask	Content of bits 24-31 of matching memory data	Lowest matched entry number
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1	

Entry # 0
Entry # 1
...
Entry # 10 (matched)
...
Entry # 14 (matched)
...
Entry # 31 (matched)

If a write transfer register contains 0xf000abcd and mask #1 contains 0x00008888, and entry #1 (word 2) contains 0xfac0dcba, then bitwise AND (0xf000abcd & 0x00008888) is equal to (0xfac0dcba & 0x00008888) and TCAM entry #1 memory location will match. As the result, bit 16+2/2=17 will be set.

3. All 384-bits and 512-bits CAM and TCAM commands are required to have an indirect\_ref token. This is necessary since each 384 and 512-bit command must set bit 3 in the length field. The eight Length values and corresponding CAM/TCAM command is shown below.

Given the Length field value the CAM/TCAM lookup command is selected.

- Length == 4b1000; 8 bit lookup for 384 bits CAM/TCAM
- Length == 4b1001; 16 bit lookup for 384 bits CAM/TCAM
- Length == 4b1010; 24 bit lookup for 384 bits CAM/TCAM
- Length == 4b1011; 32 bit lookup for 384 bits CAM/TCAM
- Length == 4b1100; 8 bit lookup for 512 bits CAM/TCAM
- Length == 4b1101; 16 bit lookup for 512 bits CAM/TCAM
- Length == 4b1110; 24 bit lookup for 512 bits CAM/TCAM
- Length == 4b1111; 32 bit lookup for 512 bits CAM/TCAM

Two indirect\_ref formats are available to load the Length field; the new and recommended V2 Indirect\_ref format and the V1 Indirect\_ref format. Each are shown below.

**V2 Indirect\_ref method to override the Length field:** In the Indirect\_ref Previous ALU Result the OV\_LEN bit is set and LENGTH field is set according to the CAM/TACM command required. Refer to section [Section 2.1.6.4.6.2](#).

### Example using V2 Indirect\_ref format for cam384\_lookup8:

```
alu_shf[--,--,B, ((8 << 1) | 1 ), <<7]
mem[cam_lookup, $x[0], addr, 0, 1], indirect_ref, sig_done[s1]
```

**V1 Indirect\_ref method to override the Length field:** The Indirect\_ref encoding 4b0010 “Override length only” format is chosen. Refer to section [Section 2.1.6.4.5](#) .

### Example of mem[cam\_lookup]

The MU CAM supports various sizes of input match data and CAM table sizes. CAM table may reside in Cluster Target, External or Internal memory. NFP-6xxx memory architecture provides hardware acceleration engines for CAM operations on various CAM table entry sizes.

CAM table requires pre-initializing all table entries to zero, and search for CAM entry of zero is invalid.

The CAM operations are inherently atomic, so multiple FPC threads may perform CAM operations - and they will happen atomically. No external atomic mechanism is needed.

On a CAM match the entry index is returned and can be used as an offset into another table that contains the actual application data structure. One use case is to use a CAM table to front-end a hash table. For every CAM index there is a corresponding hash bucket into a hash table. One can, for instance, place 32-bits of hash value of a certain tuple-key as CAM value and look it up at runtime. If matched, the index number of matched CAM entry can be used to access corresponding hash table entry.

Managing the CAM table is up to the application.

- One use case is when CAM entries are static and a management thread adds and deletes CAM entries with mem[cam\_lookup\_add] and mem[clr], while other threads perform the CAM lookups with mem[cam\_lookup].
- Another use case is where entries are added and removed dynamically and mem[lookup\_add] and mem[clr] commands are executed by each of the worker threads. Thus all worker threads participate in managing the CAM table.

In order to remove old/stale entries from the CAM, one will need to write zero atomically to CAM entry of interest. So any mem-atomic operation can be used (clr for example). Depending on the application, there can be different reasons to purge entries in CAM; for example to make room for a new entry in an already full CAM, one can adopt Least Recently Used algorithm where timestamp is maintained in a separate data structure and may be used to find the LRU and purge it. In other applications, an entry may be explicitly deleted due to certain delete event.

The mem[cam\_lookup\_add] operation has nice feature where if lookup of a CAM tag fails, it is added, provided there is space. If there is no space, CAM result returns 0xFF which then can be used by the ME thread in operation to purge an LRU and then add its tag to CAM table entry.

### Example 1. Atomic CAM lookup on 128-bit CAM with sixteen 8-bit entries.

With mem[cam128\_lookup8\_add, \$xfer0, ...] command the CAM is 128-bit table size with 1-byte entries for a total of 16 entries. This command accepts a 1-byte search input value loaded in \$xfer0[7:0] register. The lookup engine pulls data from transfer register and performs atomic lookup on the CAM. If no match exists and one or

more CAM entries are unused (byte value is 0x00) then the command search byte value is written to the first available entry in the CAM table.

The mem[cam128\_lookup8\_add] example below demonstrates the use case where a worker thread performs CAM Table management using LRU algorithm on CAM table in Cluster Target Memory 0x80000\_1000. The code will handle three cases:

- Match
- No match and add to CAM Table
- No match, apply LRU and add to CAM Table

If the CAM table is full one can adopt Least Recently Used (LRU) algorithm where timestamp is maintained in separate data structure and is used to find the LRU and purge it with mem[clr] command. Or an entry may be explicitly deleted due to certain delete event. Nonetheless the CAM table is managed in an atomic fashion because all CAM operations are inherently atomic.

```

; -----
; Worker thread has received a packet and its job
; is to perform CAM lookups and manage CAM Table
; CAM Table is 128 bits in CTM memory at starting address 0x8000001000

.sig s1
.reg r_mem_addr_ulw
.reg r_mem_addr_llw
.reg r_camAdr_llw
.reg r_cam_entry
.reg tag
.reg r_byteMask
.reg $xdataClr0
.set $xdataClr0
.xfer_order $xdataClr0
.reg $xdata0 $xdata1 $xdata2 $xdata3
.set $xdata0 $xdata1 $xdata2 $xdata3
.xfer_order $xdata0 $xdata1 $xdata2 $xdata3

; Register initialization
r_mem_addr_ulw = 0x80000000
r_mem_addr_llw = 0x1000
tag = 0x1
r_byteMask = 0x00000000
r_camAdr_llw = 0x1000

; CAM search on tag byte from packet received
CAM_LOOKUP#:

; Load tag byte into transfer register that will be used in lookup instruction.
alu[r_cam_entry, --, B, tag]
alu[$xdata0, --, B, r_cam_entry]
; perform lookup on CAM table in CTM address 0x8000001000
mem[cam128_lookup8_add, $xdata0, r_mem_addr_ulw, <<8, r_mem_addr_llw], sig_done[s1]
ctx_arb[s1]

; Check transfer register for result of CAM lookup.
; If return value is 0xff then no match is found and no free entries in CAM.
; Delete CAM entry determined by LRU algorithm - write zero to CAM entry (CTM memory)
; If return value is 0x8y, where y is between 0 and f (for up to 16 CAM entries) then

```

```

; lookup did not find match but was added to CAM Table and y is CAM entry index.
; Bit[7] indicates the entry was added to CAM table.
; If return value is 0x01 then match is found in CAM table.
; Return value is index where match was found.
alu[--, $xdata0, -, 0xff]

bne[MATCH_OR_CAM_ADD#]
; No match found and no entry in CAM Table, make room using LRU algorithm
; LRU algorithm determines which CAM entry to remove.
; LRU macro details not shown for this example
; LRU might return a mask byte and CTM memory offset from CAM base address that will
; be used to make an entry free.
lru(r_byteMask, r_camAddr_llw) ;
alu[$xdataClr0, --, B, r_byteMask]

; Clear byte in CTM memory, byte mask is loaded in $xdataClr0 transfer register.
mem[clr, $xdataClr0, r_mem_addr_ulw, <<8, r_camAddr_llw], sig_done[s1]
ctx_arb[s1]

; Now add new entry to CAM Table now that CAM entry is free (set to zero in CTM CAM memory)
br[CAM_LOOKUP#]

MATCH_OR_CAM_ADD#:
; check if tag was added to CAM entry and update LRU data structure.
; $xdata[7] set == added to CAM
br_bset[$xdata0, 7, CAM_ADD#]

; $xdata0 contains the entry where MATCH data is added
; Update LRU table (details not shown here)
; Use $xdata0 in byte lower 7 bits that contain CAM table entry index for hash table lookup.
br[DONE#]

CAM_ADD#:
; No match and CAM entry available and was added.
; Update LRU table (details not shown here)
; Use $xdata0 in byte lower 7 bits to update hash table with new entry added to CAM table.

DONE#:

```

### **Example of mem[tcam\_lookup]**

The MU TCAM supports various sizes of input match data and TCAM table sizes. TCAM table may reside in Cluster Target, External or Internal memory. NFP-6xxx memory architecture provides hardware acceleration engines for TCAM operations.

### **Example 2. TCAM lookup on 512-bit TCAM with 32-bit TCAM Data entries.**

TCAM example below is configured for 512-bit TCAM, eight long word TCAM entries, where each entry is 32-bits of Data and 32-bits of Mask value.

The mem[tcam512\_lookup32] will return in read transfer register the following field values:

- \$xfer0[7:0] = Lowest matched TCAM Entry Index (even word Entry#)
- \$xfer0[15:8] = Contents of lowest matched TCAM memory data[31:24]
- \$xfer0[31:16] = Bit mask value indicating all matched TCAM entries, defined as bit[16] is entry#(0-1), bit[17] is entry#(2-3) and so on to bit[23] entry#(15-16). (See [Table 2.26](#))

If there is no TCAM match the read transfer register will contain the value of 0x0000\_00ff.

For a particular TCAM lookup where the result is a match, the actual lookup might result in multiple matches given the TCAM configuration of data and mask words. The read transfer register bit[7:0] will contain the lowest TCAM entry index where a match was found. This value is a word index from the start of TCAM memory entry# 0. Valid values will be 0, 2, 4, 6, 8, 10, 12, 14. Read transfer register [31:16] provides a bit map of all TCAM matches and this value is bit mapped to long word (even-odd) entry#s shown in [Table 2.26](#)

In the TCAM Table below ([Table 2.26](#)) the mask values determine which bits to mask for matching each of the 32-bits of the TCAM word and match data word from write transfer register. If mask bit is cleared then that bit is don't care and if set then this bit must match the TCAM data bit and pulled transfer register bit value.

If the write transfer register contains 0xfa000cba (match data) the TCAM data is evaluated as follows:

For each even and odd word entry pairs (data and mask) in TCAM table, a compare is performed with the match data (pull data), TCAM data and TCAM mask. If both bit-wide AND of the match data and TCAM data against the TCAM mask word evaluate to be true then there is a match. The equation is shown below followed by three TCAM long word match operations.

**(match\_data & TCAM\_Mask == TCAM\_Data & TCAM\_Mask)**

```
; Three evaluations - no match and two match examples:  
(Match data & TCAM Mask == TCAM data & TCAM Mask) ? Match : NO Match  
  
0xfa000cba & 0xffff00f00 == 0xfa030201 & 0xffff00f00  
-> 0xfa000c00 != 0xfa000200 ; NO match  
  
0xfa000cba & 0xff0000ff == 0xfac0dcba & 0xff0000ff  
-> 0xfa0000ba == 0xfa0000ba ; Match  
  
0xfa000cba & 0x08000f0f == 0xc8e25caa & 0x08000f0f  
-> 0x08000c0a == 0x08000c0a ; Match
```

Read Transfer register will contain the following:

\$xdata[7:0] = 0x04 ; word offset of lowest entry# (entry#(4-5))

\$xdata[15:8] = 0xfa ; TCAM data[31:24]

\$xdata[31:16] = 0x0044 ; Entry# (4-5) and (12-13), two hits found

**Table 2.26. TCAM Table layout for mem[tcam512\_lookup32] example**

(even-odd) word Entry#	bitmask map to Rd xfer[31:16]	CTM Address	TCAM DATA value (even word)	TCAM MATCH value (odd word)
0-1	xfer[16]	0x1000:1007	0xaa030201	0xffff00ff0
2-3	xfer[17]	0x1008:100f	0xbb13f1e1	0xffff00f00
4-5	xfer[18]	0x1010:1017	0xfac0dcba	0xff0000ff
6-7	xfer[19]	0x1018:101f	0xcc24f5e5	0xffffffff00
8-9	xfer[20]	0x1020:1027	0xfb1ccaa	0xffff00f0f
10-11	xfer[21]	0x1028:102f	0xfc29981	0xffffffff0
12-13	xfer[22]	0x1030:1037	0xc8e25caa	0x08000f0f
14-15	xfer[23]	0x1038:103f	0xa2b25211	0xffffffff

The code below runs on a thread that receives a 32 bit match data value constructed from a packet. The objective of this code is to search in the TCAM for a match and return the pushed data for determining what to do with the packet. The TCAM has been pre-configured with the values shown above in TCAM Table layout. (See [Table 2.26](#))

```
; Atomic 512bit wide TCAM test routine with 32bit entry size
.sig s1 s2
.reg r_match_data
.reg ind_ref_reg
.reg r_mem_addr_ulw
.reg r_mem_addr_llw

.reg read write $xdata0 $xdata1, $xdata2, $xdata3
.xfer_order $xdata0 $xdata1, $xdata2, $xdata3

r_mem_addr_ulw = 0x80000000
r_mem_addr_llw = 0x1000

; Move r_match_data containing the 0xfa000cba into $xfer0 write transfer register
alu[$xdata0, --, B, r_match_data]

; For tcam512_lookup32 must set the Length to 0xf with V2 indirect_ref
; Set OV_LEN = 1 and set length field
alu[ind_ref_reg, 0x80, OR, 0xf, <<8]
alu[--, --, B, ind_ref_reg]
mem[tcam512_lookup32,$xdata0,r_mem_addr_ulw,<<8,r_mem_addr_llw],indirect_ref,sig_done[s1]
ctx_arb[s1]

alu[--, $xdata0, -, 0xff]
beq[NO_MATCH#]
; Match found, send 32-bit result in $xdata0 to statistics handler
; Forward the packet based upon the result data.
; $xdata will contain the value of 0x0044fa04
; Apply to hash table

NO_MATCH#:
; No match found in TCAM drop the packet or handle this as exception case.

DONE#:
```

## 2.2.39.3 MEM (Lists)

There are four types of link lists:

1. Segment-based linked list (type 0)
2. Buffer-based linked list counting packets (packet mode) (type 1)
3. 32-bit buffer-based linked list counting buffers (type 2)
4. 24-bit buffer-based linked list counting buffers (type 3).



### Note

MUConfigCPP[DirAccWays] has no effect on the Queue Engine Commands Dequeue and Enqueue. These commands use the QueueLoc field of the Queue Descriptor to determine whether memory access is to the External Memory or direct access (to DCache).

### 2.2.39.3.1 Type 0 Lists – Dequeue Segments and Count Packets

Segment-based linked list: each link includes SOP, EOP, a segment count, and a 24-bit long word address. Type 0 allows multiple packets to be stored per buffer in a chain of buffers.

If this is the first entry on the list, the enqueue operation sets the tail and head pointers to the element being enqueued. The segment count, EOP and SOP is set from the enqueue command. Queue count is set to one.

When there are already entries on the list, the enqueue operation writes the new entry at the tail pointer, updates the tail pointer in the queue array to point to the new entry and increments the count in the queue array.

The dequeue operation decrements the segment count value and only removes the entry from the list when the segment count is zero. The queue count is only decremented when the EOP bit is set.

**Table 2.27. Type 0 - List descriptor**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
seg_cnt				head_ptr																EOP	SOP										
tail_ptr																												0	0		
q_loc	Reserved		q_page	q_count																											
Reserved																															

**Table 2.28. Type 0 - Link format for NFP enqueue and dequeue command**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
seg_cnt				q_link																EOP	SOP										

**Table 2.29. Type 0 - Link format in memory when using ME to construct chain of packets**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EOP	SOP	seg_cnt				q_link																									

The byte address of the head item is {q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:24], head\_ptr[23:0], 2b0}.

3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0											
q_loc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0					

The byte address of the tail item is {q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:0], 2b0}.

3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0														
q_loc	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0							

### 2.2.39.3.2 Type 1 Lists - Dequeue Buffers and Count Packets

Buffer-based linked list counting packets: each link includes SOP, EOP, a 24-bit byte address plus 6 bits of user data.

If this is the first entry on the list, the enqueue operation sets the tail and head pointers to the buffer being enqueue. The segment count, EOP and SOP is set from the enqueue command. Queue count is set to one.

When there are already entries on the list, the enqueue operation writes the new entry at the tail pointer, updates the tail pointer in the queue array to point to the new entry and increments the count in the queue array.

The dequeue command always removes the entry from the head of the list, but only decrements the queue count when the EOP bit is set.

Type 1 lists are typically used when packets are larger than a single buffer, allowing a packet to be stored across a few buffers.

This type of list also provides a 6 bit field which can be used for user data which is unaltered between enqueue and dequeue operations.

**Table 2.30. Type 1 - List descriptor**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
User Data				head_ptr																				SOP	EOP						
tail_ptr																														0	1
q_loc	Reserved	q_page	q_count																												
Reserved																													Reserved		

**Table 2.31. Type 1 - Link format for NFP enqueue and dequeue command**

**Table 2.32.** Type 1 - Link format in memory when using ME to construct chain of packets

The byte address of the head entry is  $\{q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:24], head\_ptr[23:0], 2b0\}$ .

The byte address of the tail entry is  $\{q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:0], 2b0\}$ .

### 2.2.39.3.3 Type 2 Lists - Dequeue Buffers and Count Buffers

32-bit buffer-based linked list counting buffers: each link includes a 30-bit long word address.

If this is the first entry on the list, the enqueue operation sets the tail and head pointers to the buffer being enqueued. The segment count, EOP and SOP is set from the enqueue command. The queue count is set to one.

When there are already entries on the list, the enqueue operation writes the new entry at the tail pointer in memory, updates the tail pointer in the queue array to point to the new entry and increments the queue count in the queue array by one.

The dequeue command always removes the entry from the head of the list, but only decrements the queue count when the EOP bit for the entry was set on the enqueue command. The values for SOP and EOP in the link returned from the dequeue command, are inverted from the values used for the enqueue command. The values in the queue array is the same as the enqueue command.

Type two lists are typically used for single buffer per packet operations where all the buffers are in the same 4GB window.

**Table 2.33. Type 2 - List descriptor**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																												SOP			
																												EOP			
head_ptr																															
tail_ptr																													1	0	
q_loc	Reserved		q_page	q_count																											
Reserved																												Reserved			

The byte address of the head entry is  $\{q\_loc[1:0], 4b0, q\_page[1:0], head\_ptr[29:0], 2b0\}$ .

The byte address of the tail entry is  $\{q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:0], 2b0\}$ .

#### Type 2 - Link format in memory when using ME to construct chain of packets

It is not advisable to manually construct a chain of packets for type 2 lists as the enqueue command does not support the seg\_cnt parameter and one will have to manipulate the descriptor to update the queue array count manually to ensure all the packets are accounted for.

**Table 2.34.** Type 2 - Link format for enqueue command

q\_link

EOP      SOP

**Table 2.35.** Type 2 - Link format from dequeue command

q\_link

#### 2.2.39.3.4 Type 3 Lists - Dequeue Buffers and Count Buffers

24-bit buffer-based linked list counting buffers: each link includes a 24-bit word address.

If this is the first entry on the list, the enqueue operation sets the tail and head pointers to the buffer being enqueued. The SOP and EOP is set from the enqueue command and the queue count is set from the segment count.

When there are already entries on the list, the enqueue operation writes the new entry at the tail pointer in memory, updates the tail pointer in the queue array to point to the new entry and adds the segment count to the count in the queue array. It is illegal to use a segment count of 0 for the enqueue command.

The dequeue command always removes the entry from the head of the list and decrements the queue count by one. The values for SOP and EOP are the same as used for the enqueue command.

Type three lists are typically used for single buffer per packet operations with all the buffers in the same 64MB window.

**Table 2.36. Type 3 - List descriptor**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
seg_cnt																													SOP	EOP	
																													1	1	
tail_ptr																															
q_loc	Reserved	q_page																											q_count		
																													Reserved	Reserved	

**Table 2.37. Type 3 - Link format for NFP enqueue and dequeue command**

**Table 2.38. Type 3 - Link format in memory when using a ME to construct chain of packets**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EOP	SOP	seg_cnt				q_link																									

The byte address of the head entry is {q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:24], and head\_ptr[23:0], 2b0}.

3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0									
q_loc	0	0	0	0	0	q_page	tail_ptr[29:24]				head_ptr																		0	0								

The byte address of the tail entry is {q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:0], 2b0}.

3	3	3	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	9	8	7	6	5	4	3	2	1	0
9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0										
q_loc	0	0	0	0	0	q_page	tail_ptr																								0	0							

### Memory alignment

The list link element must be aligned to a 16 byte address boundary.

### Example of Link-list Queue

When one needs to store or free single buffer packets in strict order, queues are a good way to implement FIFOs. Queues allow for multiple packets to be received and transmitted simultaneously. Queues allow for providing quality of service utilizing multiple queues. If there are many queues and performance is critical then avoiding accessing external memory is desired. The NFP architecture provides a QArray in the MU Queue Engine (one per Memory Unit) that can store up to 1024 queue descriptors, thus avoiding accesses to external memory. If more than 1024 queue descriptors are required then software must manage swapping queue descriptors between external memory and MU Queue Engine QArray. Link lists type-2 are typically used for single buffer packet operations where all the buffers are in the same 4GB window.

Queue Descriptor contains the following fields for type-2 Lists:

- Head – 30 bit word address pointer to the head of the linked list.
- Tail – 30 bit word address pointer to the last item in the linked list.
- Count – 24 bit count of the number of buffers in the linked list. Will decrement during dequeue operation only when the EOP bit for the entry was set during the enqueue.
- EOP – 1 bit field indicating that the first item of the linked list is an end-of-packet. Used by dequeue commands to indicate that Count field should be decremented when the buffer is dequeued.
- SOP – 1 bit field indicating that the first item of the linked list is the start-of-packet. Used by dequeue commands to indicate that this is the first time a link has been returned.

- QueuePage – A 2-bit value, from second word bits[25:24] during ReadQDesc command. It is used for the top bits of the queue entry addresses (see below); all queue items must be within this window of memory.
- QueueLoc – A 2-bit value, which is from second word bit[31:30] during ReadQDesc command. It is used to determine whether the access is to External Memory or to Internal Memory for commands except ReadQDesc and WriteQDesc as for them it is part of command address bits[39:38].
- QueueType – A 2-bit value indicating what type of dequeue should be supported by the queue. The mode for Link List type-2 is 2, which means 30-bit word address buffer mode.

The Queue pointer addresses are composed of the following (Elements can be anywhere in the 4GB address space):

- Head item of buffer linked list {QueueLoc[1:0], 4b0000, QueuePage[1:0], Tail[29:0], 2b00}
- Tail item of buffer linked list {QueueLoc[1:0], 4b0000, QueuePage[1:0], Tail[29:0], 2b00}

The example code below shows how to create a Queue Descriptor and perform enqueue and dequeue operations on link list Type-2. Assume that one buffer contains the entire packet, thus the SOP and EOP are asserted on enqueue operations. The first step is to create a queue descriptor in DDR memory for a link list type-2 queue, followed by reading the queue descriptor into the memory unit Queue Engine QArray. Finally the enqueue and dequeue operations will be shown to add and delete entries in the link list queue.

```

/*
 * init linked list queue descriptor in QArray from fields, nfp format. used to init
 * linked list to initial ( empty ) condition.
 *
 * _ddr_addr           - dram location to use for descriptor. must
 *                       be 16 byte aligned
 * _ddr_loc            - locality of mem w/queue desc
 * _ddr_ilnd           - island id of mem w/queue desc
 * _que_no             - que number, 0-1023
 * _que_addr           - base memory location of queue data.
 *                       must be lword aligned ( bits 1:0 = 0 )
 *                       and max of 24 bits ( types 0,1,3 ) or
 *                       max of 30 bits ( type 2 )
 * _page, _type, _loc   - page, type, loc fields. 2 bits each
 *
 */
#define _queLinkListInit( _ddr_addr, _ddr_loc, _ddr_ilnd, _que_no,\
                      _que_addr, _page, _type, _loc)
.begin
    .sig s1
    .reg r_addr, r_que_no, r_head_ptr_mask, r_tail_ptr_mask, r_tmp, r_loc_ilnd
    .reg write $xfer0[4]
    .reg read $xfer1[4]
    .xfer_order $xfer0
    .xfer_order $xfer1

    r_head_ptr_mask = 0x03fffffc
    r_tail_ptr_mask = 0xfffffff0

    // prepare descriptor
    // word 0
    .if (_type != 2)
        $xfer0[0]= _que_addr & r_head_ptr_mask

```

```

.else
    $xfer0[0] = _que_addr
.endif

// word 1
r_tmp = _que_addr & r_tail_ptr_mask // tail.
r_tmp = r_tmp | _type           // type
$xfer0[1] = r_tmp

// word 2
r_tmp = 0
r_tmp = r_tmp | (_loc << 30)    // loc
r_tmp = r_tmp | (_page << 24)   // page
$xfer0[2] = r_tmp

// word 3
$xfer0[3] = 0                  // reserved

// Write descriptor to ddr, 40-bit address,
// EXT MEM 0 Island ID 0x18
// MU Access mode (_ddr_loc= 2b10)
r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
r_loc_ilnd = r_loc_ilnd << 24
r_addr = _ddr_addr
mem[write, $xfer0[0], r_loc_ilnd, <<8, r_addr, 2], ctx_swap[s1]

// Read back to ensure data reaches DRAM
mem[read, $xfer1[0], r_loc_ilnd, <<8, r_addr, 2], ctx_swap[s1]
.end
#endifm

/*
 * Read queue descriptor from memory into QArray
 *
 * _queue_no      - QArray number 0-1023
 * _queue_addr    - 16 byte aligned mem location of initialized QDesc
 * _buf_loc       - locality of mem w/queue desc
 * _buf_ilnd      - island ID of mem w/queue desc
 */
#macro _queReadDesc_nfp(_queue_no, _queue_addr, _buf_loc, _buf_ilnd)
.begin
    .sig s1
    .reg r_tmp, r_loc_ilnd
    .reg read $xdebug[4]
    .xfer_order $xdebug

    r_loc_ilnd = _buf_ilnd | (_buf_loc << 6)
    r_loc_ilnd = r_loc_ilnd << 24

    ; Use NFP-6xxx Indirect reference format where Queue no is written
    ; to the prev_alu Data16 field and OVE_DATA value is 1
    ; indicating load prev_alu DATA[13:0] into cpp_command.data_ref[13:0]
    r_tmp = _queue_no << 16 ; Load queue no in register for next instruction
    alu[--, r_tmp, +, 8]      ; Set prev_alu[7] and prev_alu DATA16
    mem[rd_qdesc, --, r_loc_ilnd, <<8, _queue_addr], indirect_ref

    ; Can use push_qdesc to debug the QArray contents.
    mem[push_qdesc, $xdebug[0], r_loc_ilnd, <<8, _queue_no], ctx_swap[s1]

```

```

nop
nop
.end
#endif

/*
 * Add entry to link list
*
* _ddr_qlink_addr - dram location to use for descriptor. must
*                     be 16 byte aligned
* _ddr_ilnd       - island id of mem w/queue desc
* _ddr_loc        - locality of mem w/queue desc
* _que_no         - que number, 0-1023
*
*/
#endif

#define _enqueueBuf(_ddr_qlink_addr, _ddr_ilnd, _ddr_loc, _que_no)
.begin
    .sig s1
    .reg r_tmp
    .reg r_loc_ilnd
    .reg r_buf_addr, r_qadr
    .reg read $xdebug[4]
    .xfer_order $xdebug

    r_buf_addr = _ddr_qlink_addr
    r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
    r_loc_ilnd = r_loc_ilnd << 24

    ; Enqueue buffer on the Link List of the Queue Descriptor in QArray.
    ; Set SOP and EOP meaning that one packet is contained in 1 buffer.
    ; This way when dequeue is executed the QCount will decrement.
    ; Use NFP-6xxx indirect reference format to set queue number in prev_alu
    ; DATA16 field and set OVE_DATA = 1.
    r_qadr = r_buf_addr + 3 ; set SOP and EOP
    r_tmp = _que_no << 16
    alu[--, r_tmp, +, 8]           // Sets the prev_alu
    mem[enqueue, --, r_loc_ilnd, <<8, r_qadr], indirect_ref

    ; Can use push_qdesc to debug the QArray contents.
    ; $xdebug[0] will contain head ptr
    ; $xdebug[1] will contain the tail ptr
    ; $xdebug[2] will contain the Queue Counter value.
    mem[push_qdesc, $xdebug[0], r_loc_ilnd, <<8, _que_no], ctx_swap[s1]
    nop
.end
#endif

/*
 * Remove first entry from the head of the link list
* Returns the Qdesc head pointer, !EOP and !SOP
* _ddr_ilnd - island id of mem w/queue desc
* _ddr_loc  - locality of mem w/queue desc
* _que_no   - que number, 0-1023
*
*/
#endif

#define _dequeueBuf(_ddr_ilnd, _ddr_loc, _que_no)
.begin

```

```

.sig s1
.reg read $xref $xdebug[4]
.reg r_loc_ilnd
.xfer_order $xdebug

r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
r_loc_ilnd = r_loc_ilnd << 24

; Remove buffer form Link List.
; $xref will contain the pointer to the buffer removed from the Link List.
mem[dequeue, $xref, r_loc_ilnd, <<8, _que_no], ctx_swap[s1]

; Can use push_qdesc to debug the QArray contents.
; $xdebug[0] will contain head ptr
; $xdebug[1] will contain the tail ptr
; $xdebug[2] will contain the Queue Counter value.
mem[push_qdesc, $xdebug[0], r_loc_ilnd, <<8, _que_no], ctx_swap[s1]

.end
#endifm

; Main
.begin
.if(ctx() == 0)
.reg _ddr_addr, _ddr_loc, _ddr_ilnd
.reg _que_addr, _page, _type, _loc _que_no

_ddr_ilnd = 0x18
_ddr_loc = 0x2
_ddr_addr = 0x0
_que_addr = 0x0
_que_no = 0x1
_page = 0x0
_loc = 0x0
_type = 0x02

; Initliaze the type 2 link list Queue Descriptor in External memory
._queLinkListInit( _ddr_addr, _ddr_loc, _ddr_ilnd, _que_no,\ 
                    _que_addr, _page, _type, _loc)

; Read Queue descriptor into MU Queue Engine QArray
._queReadDesc_nfp(_que_no, _que_addr, _ddr_loc, _ddr_ilnd)

; Enqueue a buffer to the link list queue
._enqueueBuf(_ddr_addr, _ddr_ilnd, _ddr_loc, _que_no)

; Dequeue a buffer from the link list queue
._dequeueBuf(_ddr_ilnd, _ddr_loc, _que_no)

.endif
.ctx_arb[kill]
.end

```

## 2.2.39.4 MEM (Lists Dequeue)

This command removes a segment or a buffer from the queue and is only applicable to lists.

## Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	dequeue	Dequeue an entry from a linked list and return it in the read transfer register. If a queue is empty, return 0.
xfer	dequeue	A read transfer register containing a queue entry.
src_op1, src_op2	dequeue	Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"><li>• [9:0] - Queue array entry number.</li></ul>
opt Tok	dequeue	sig_done[sig_name] ctx_swap[sig_name] defer[n] where n = 1 or 2 indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2, ...]

## 2.2.39.5 MEM (Lists Enqueue)

Two types of enqueue commands are supported – one for adding a buffer to the queue and another one for updating the queue tail pointer. This command is only applicable to lists.

## Instruction Format

mem[cmd, --, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, --, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, --, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	enqueue	This command adds a buffer to the queue described by the queue descriptor and sets the tail to point to the buffer. If necessary, a link is established from the old tail buffer to the new buffer. This command is used to add an entire frame to the queue or to add the Start-of-Packet buffer of a multi-buffer frame to the queue. See <a href="#">Note 1</a> .
	enqueue_tail	This command updates the tail pointer only. This command must be preceded by an enqueue command to the same entry. See <a href="#">Note 1</a> .
--	Both commands	Must always be “--”.

Parameter	Command	Description
src_op1, src_op2	Both commands	<p>Restricted operands are added (<math>\text{src\_op1} + \text{src\_op2}</math>) to define the following:</p> <ul style="list-style-type: none"> <li>• [0:0] - SOP</li> <li>• [1:1] - EOP.</li> </ul> <p>For buffer-based linked lists:</p> <ul style="list-style-type: none"> <li>• [31:2] - Entry to enqueue.</li> </ul> <p>For segment-based linked lists:</p> <ul style="list-style-type: none"> <li>• [31:26] - Segment count</li> <li>• [25:2] - Entry to enqueue.</li> </ul>
opt Tok	Both commands	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

#### Notes:

1. A queue array entry number has to be specified in the data reference field of indirect format.

#### V2 Indirect Reference Mode:

V2 Indirect reference mode example using queue array number 0x11 is:

PREV\_ALU DATA16 = 0x11 and PREV\_ALU OVE\_DATA = 0x1. Note that cpp\_command.data\_ref[13:0] is 14-bits wide and PREF\_ALU DATA16 is 16-bits wide. Do not extend the queue array entry value beyond the cpp\_command.data\_ref[13:0] size. Refer to [Section 2.1.6.4.6](#). Example code is shown below.

```
immed[tmp,((0x11<<16)|(0x8))] ; PREV_ALU=0x0011_0008
mem[enqueue,--,src1,src2], indirect_ref
```

#### V1 Indirect Reference Mode:

V1 Indirect Reference mode example of using a queue array number 0x11 is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Encoding	Reserved						Queue array entry number						Reserved						Reserved															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0

## 2.2.39.6 MEM (Load Balance Engine Operations)

The Load Balance Engine in the MU provides software with acceleration for dynamic load balancing; this can be used in a heterogeneous processing system to load balance individual CPU cores, or it might be used to load balance across individual links in an aggregated network connection (such as a pair of 10GbE links between two switch nodes).

The key to the operation of a load balanced system of 'pipes' is to maintain the same flows travelling over the same pipe. For example in the heterogeneous case, the CPUs have caches, and migrating data between the

caches is to be avoided; the cache data has affinity to the flow, so having an affinity between the flow and the CPU helps to remove inter-cache transfer requirements.

The Load Balance Engine takes a hash of the flow and performs a lookup that returns an id, which the software can utilize to steer the flow. To aid dynamic balancing, the engine also maintains statistics (based on data supplied with the flow id by the software) for every 'steer' that the engine performs. This data can be used to dynamically tweak the engine's tables.

## Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8, ref_cnt], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	lb_bucket_read_dcache	Read and return HashBucket data from the HashBucket residing in the designated location in DCache.
	lb_bucket_read_local	Read and return HashBucket data from the local HashBucket Array.
	lb_bucket_write_dcache	Write HashBucket data into the HashBucket residing in the designated location in DCache.
	lb_bucket_write_local	Write HashBucket data into the Local HashBucket Array.
	lb_lookup_bundleid	Perform a comparison lookup on 24-bit initial Hash input and return BundleId. Add 16-bit Stat input data to the statistics counter. Pull one 64-bit word of Hash and Stat, push two 64-bit words as result.
	lb_lookup_dcache	Perform a comparison lookup on 24-bit initial Hash input, use the BundleId to address IDTable, and use IDTable result to address the designated location in DCache that becomes the DCache lookup result. Add 16-bit Stat input data to the statistics counter. Pull one 64-bit word of Hash and Stat, push two 64-bit words as result.
	lb_lookup_idtable	Perform a comparison lookup on 24-bit initial Hash input, use BundleId to address IDTable and return IDTable result. Add 16-bit Stat input data to the statistics counter. Pull one 64-bit word of Hash and Stat, push two 64-bit words as result.
	lb_push_stats_dcache	Read and return Stats data from the designated location in DCache.
	lb_push_stats_dcache_clr	Read and return Stats data from the designated location in DCache. Write zeros into same location to clear Stats data.

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
	lb_push_stats_local	Read and return Stats data from the Local Stats Table Array.
	lb_push_stats_local_clr	Read and return Stats data from the Local Stats Table Array. Write zeros into same location to clear Stats data.
	lb_read_desc	Read and return Descriptor data from Local Descriptor Array.
	lb_read_idtable	Read and return IDTable data from the Local IDTable Array.
	lb_write_desc	Write Descriptor data into the Local Descriptor Array. The pull data associated with the command is written into the entries specified in the command address and length fields. The array supports up to 64 Descriptors. (See <a href="#">Note 1</a> )
	lb_write_idtable	Write IDTable data to the Local IDTable Array. The pull data associated with the command is written to the entries specified in the command address and length fields. The array supports up to 256 IDTable entries. (See <a href="#">Note 1</a> )
xfer	lb_bucket_read_dcache, lb_bucket_read_local, lb_push_stats_dcache, lb_push_stats_dcache_clr, lb_push_stats_local, lb_push_stats_local_clr, lb_read_desc, lb_read_idtable	Read transfer registers.
	lb_bucket_write_dcache, lb_bucket_write_local, lb_write_desc, lb_write_idtable	Write transfer registers.
	lb_lookup_bundleid, lb_lookup_dcache, lb_lookup_idtable	Write transfer registers and response in Read transfer registers.
src_op1, src_op2	All Commands	Restricted source operands that define byte address. The address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a>
ref_cnt	lb_bucket_read_dcache, lb_bucket_read_local, lb_bucket_write_dcache, lb_bucket_write_local,	Reference count (n) in increments of (n*2) 8-byte words. Valid values are 1 to 8.
	lb_read_desc, lb_write_desc	Reference count (n) in increments of (n*3) 8-byte words. Valid values are 1 to 5.

Parameter	Command	Description
	lb_read_idtable, lb_write_idtable	Reference count in increments of 8-byte words. Valid values are 1 to 16. Values above 8 must be specified using indirect reference.
	lb_push_stats_dcache, lb_push_stats_dcache_clr, lb_push_stats_local, lb_push_stats_local_clr	Reference count (n) in increments of (n*2) 4-byte words. Valid values are 1 to 16. Value of 1 returns one 64-bit value (two 32-bit words of statistics). Values above 8 must be specified using indirect reference.
	lb_lookup_bundleid, lb_lookup_dcache, lb_lookup_idtable	Not Required. See <a href="#">Section 2.1.6.3</a>
opt Tok	All Commands	sig_done[sig_name] ctx_swap[sig_name] defer[n] where n = 1 or 2 indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2,...]

#### Notes:

- Prior to use, it is required to setup the HashBucket, IDTable, DCache and clear the statistics counters. Descriptors must be loaded to the Descriptor Array prior to use for lookups. This is achieved via the lb\_write\_desc command. This command will load a descriptor from the pull data and store it to a location in the Descriptor Array. Subsequent commands, such as lb\_lookup\_dcache, specify the appropriate descriptor to use by referencing the entry of the Descriptor Array where the descriptor is stored.

### 2.2.39.7 MEM (Lookup Engine Operations)

The Lookup Engine in the Memory Unit performs various lookup table operations and data matching functions, as needed by the packet processing and classification requirements. The lookup results are used for packet route determination, queue destination and memory buffer pool strategy. The intent of the Lookup Engine design is to be highly flexible, allowing software to program any number of possible table configurations.

There are three basic types of lookup operations: direct, algorithmic and hash. Each of the seven lookup operation types defined can be classified as one of these types. Refer to the Netronome Databook for further details.



#### Note

In case of Algorithmic Lookup commands which need to do a match of data with the contents of the memory, if more than one match is found, the result from the lowest address is returned.



## Note

In the case of Lookups which require configuration of various tables, the Maximum Table size depends on the memory being used by the Lookup Engine, as different memories have different maximum configurations. See list below for details:

- **Internal Memory Unit:** Maximum Table size = the size of the Dcache, which is 4MB
- **External Memory Unit, using Direct Access, but not using DirAccWays:**  
Maximum Table size = size of Direct Access Memory, which is 1MB
- **External Memory Unit, using Direct Access and using DirAccWays:**  
Maximum Table size = size of Direct Access Memory + the size of the DCache enabled for Direct Access via DirAccWays. For example, if four ways of cache are enabled for Direct access via DirAccWays, the Maximum Table size would be  $(1\text{MB} + 4 * 256\text{KB}) = 2\text{MB}$ . (Note S/W must take care to correctly calculate the addresses passed to the Lookup Engine, accounting for which ways of DCache are declared Direct Access via DirAccWays)
- **External Memory Unit, not using Direct Access:** all table sizes supported

## Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	lookup	Issue a request, encoded in the address field, to the lookup engine. The match data is specified in Write Transfer registers and the response provided in Read Transfer register.
xfer	lookup	2-4 Write transfer registers can be used to provide match data, as specified in the look-up instruction. Read transfer register will contain 32-bit response.
src_op1, src_op2	lookup	Restricted source operands that define byte address. The 40-bit address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a>  Address bits [30:29] determine the Lookup Operation: <ul style="list-style-type: none"> <li>• 2b00 : Algorithmic Lookup (See <a href="#">Note 1</a> )</li> <li>• 2b01 : Hash Lookup (See <a href="#">Note 2</a> )</li> <li>• 2b1x : Direct Lookup (where x is don't care). (See <a href="#">Note 3</a> )</li> </ul>
ref_cnt	lookup	Reference count in increments of 8 byte words. Valid values are 1 to 2. Length field indicates amount_of_pull_data that would be used with the command. In the case where ref_cnt

Parameter	Command	Description
		== 1 (length==0), then Lookup engine does a request for 64-bit of Pull_data. Different commands need pull_data for different purposes. See <a href="#">Note 1</a> , <a href="#">Note 2</a> and <a href="#">Note 3</a> .
opt Tok	lookup	sig_done[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]



## Note

If the Register AlgorithmicTableLocation[Location] == External (a value of '1'), (indicating the table is located in External Memory) then if Register LookupEngineConfig[MemoryLocalityConfig] is 2b00 (indicating to get the locality from CPP Command Address[39:38]), then CPP Command address bits[39:38] of the incoming CPP Command must not be 2b10 (which would indicate DirectAccess, i.e., Internal DCache), as that value should not happen since AlgorithmicTableLocation[Location] has already indicated the External memory is to be used, not the internal DCache. A value of CPP Address[39:38] = 2b10 would be illegal in this case.

### Notes:

1. Algorithmic Lookup commands are used to find the index, based on the starting bit position obtained from Dache data. Algorithmic Lookup Table operations are:

- ALUT24 : 24 bit Algorithmic Lookup
- ALUT32 : 32 bit Algorithmic Lookup
- PMM : Prefix Match with Mask Table
- TCAM : Ternary Content Addressable Memory Lookup
- SPLIT : Split Table Lookup
- PMM Prefix Match with Mask Table
- MULTI : BIT Multi-Bit Table Lookup

The 32 bit address to be constructed in the MEM (lookup) Instruction issued by an FPC takes the following form:

- Bits [23:0] : Address bits [27:4] of a specific location in memory, identified using the 3 bit Table Number, as well as pre-programmed algorithmic table locations in memory which are configured using the registers outlined in the Lookup Engine Registers Section of the NFP-6xxx Databook.
- Bits[26:24] : Table Number 0 – 7, where each Table Number is defined in Lookup Engine Registers. They contain Algorithmic Lookup Table Memory address bits [32:28], and internal (0) or external (1) memory select bit[0]. Algorithmic Lookup Table Memory Address[32:0] is :
 

{Lookup Engine Register Table # : bits[32:28], 24 lower bits of address from algorithmic lookup, 4b0000}
- Bits[28:27] : 2 bit Table size in Number of cache Lines 1 – 4 (2b00 thru 2b11)

- Bit[30:29] : Set to 2b00 indicating Algorithmic Lookup.
- Bit[31] : Set to 0 if Lookup complete; Set to 1 if further Lookup Required.

If the starting bit position is greater than 63 and the ref\_cnt equals to 1 (length == 0), then all the data would be 0.

2. Hash commands use pull data to find an address. Hash Lookup Table operations are:

- CAM : Content Addressable Memory Lookup returning memory location only.
- LHASH : Linear Hash Table Lookup returning memory location only.
- CAMR : Content Addressable Memory returning result.
- LHASH : Linear Hash Table Lookup returning result.

The 32 bit address to be constructed in the MEM (lookup) Instruction issued by an FPC takes the following form:

- Bit[31] : Set to 0 if Lookup complete; Set to 1 if further Lookup Required.
- Bits[30:29] : Set to 2b01 for HASH Lookup.
- Bits[28:12] : 17 bit Base Address.
- Bit[11] : Set to 0 indicating Internal Memory; Set to 1 indicating External Memory.
- Bits[10:8] : 3 bit Table Size: 3b000 (1K), 3b001 (2K), ... , 3b111 (128K)
- Bits[7:2] : 6 bit command opcode. Refer to NFP-6xxx Databook for list of command opcodes.
- Bits[1:0] : 2 bit starting position: 0->bit 0; 1->bit32; 2->bit64;3->bit96.

3. Direct Lookup commands use pull data to find an address. Direct Lookup Table operations are:

- DLUT24 Direct Lookup Table 24 bit result size
- DLUT32 Direct Lookup Table 32 bit result size

The 40-bit address to be constructed in the MEM (lookup) Instruction issued by an FPC takes the following form:

- Bit[39:38] : Indicate whether Internal (clear to zero) or External Memory (set to one).
- Bit[37:31] : Don't care.
- Bit[30] : Set to 1 indicating Direct Lookup.
- Bit[29:15] : upper 15 bit base address of Lookup Table for Small Table Size; or 15 bit base address of Lookup Table for Large Table Size.
- Bit[14] : 16 bit base address LSb of Lookup Table for Small Table Size; or Spare for Large Table size.
- Bit[13] : Spare
- Bit[12] : Result size. 24 bits when set to 0; 32 bits when set to 1
- Bit[11] : Don't care.
- Bits [10:8] : Table size

- Small Table sizes: 3b000 (1K), 3b001 (2K), ... , 3b111 (128K)
- Large Table sizes: 3b000 (64K), 3b001 (128K), ... , 3b111 (8M)
- Bits [7] : Table type – Small (1K – 128K) when set to 0; or Large (64K – 8M) when set to 1.
- Bits [6:0] : Starting bit position into the 64-bit index located in memory to be accessed.

The code below explains a Lookup LHASH command operation using MicroC. There are 3 things needed when we need to use Lookup engine.

- 1. We need values in memory.
- 2. We need to set up the address bus with the correct parameters for the entire table.
- 3. We need xfer regs for a specific lookup.

Some handy defines:

```
#define TEST_NAME           LOOKUP_HASH_COMMAND_LHASHR_48_60_16_7_TEST
#define NUM_MAX_BUCKETS      7
#define BYTES_PER_BUCKET     16
#define MULE_HASH_COMMAND    LOOKUP_HASH_COMMAND_LHASHR_48_60_16_7
#define WRITE_LHASH_BUCKET   write_lhashr_entry60_16_dataline

#define HASH_TEST_START_ADDRESS 0
#define ISLAND_ENCODING        0x9c00000000 // internal memory island 28
#define TEST_PASS              0
#define TEST_FAIL              1

* 1 * Lets set up memory
```

Here is a function `write_group_of_buckets`, when given `lookup_entry=0x3f9` and `start_search_bucket = 0x3f9`, will produce the following memory table:

```
unsigned long long write_lhashr_entry60_16_dataline(unsigned long long base_addr, unsigned long long entryval)
{
    __declspec(write_reg) unsigned int          general_write_reg[4];
    SIGNAL                                     sig;

    __declspec(mem_ptr, addr40) void *addr_ptr = (__declspec(mem_ptr, addr40) void *) (base_addr);

    general_write_reg[0] = entryval & 0xffffffff;           // entry 0
    general_write_reg[1] = ((entryval >> 32) & 0xffffffff) | ( bucket_valid << 29 );//entry 0

    general_write_reg[2] = 0x11111111 * bucket_valid; // result 0
    general_write_reg[3] = 0xd0cad0ca;                  // dontcare

    mem_write64_ptr40((void *)&general_write_reg, addr_ptr, 2, ctx_swap, &sig);
    return base_addr + 16;
}

void write_group_of_buckets ( unsigned long long lookup_entry, unsigned search_start_bucket )
{
    unsigned long long bucket_address, search_start_offset, expected, retval;
    unsigned          bucket_to_write_in_mem;
```

```

search_start_offset = BYTES_PER_BUCKET * search_start_bucket;
bucket_address = HASH_TEST_START_ADDRESS + search_start_offset;
// valid first entry
bucket_address = WRITE_LHASH_BUCKET (bucket_address, lookup_entry++, 1 );
// nothing done here if NUM_MAX_BUCKETS is 2
for ( bucket_to_write_in_mem = 2; bucket_to_write_in_mem <= NUM_MAX_BUCKETS - 1;
      bucket_to_write_in_mem++ )
    bucket_address = WRITE_LHASH_BUCKET (bucket_address, lookup_entry++, bucket_to_write_in_mem - 1 );

bucket_address = WRITE_LHASH_BUCKET (bucket_address, lookup_entry, NUM_MAX_BUCKETS + 1 ); // valid
}

*memory table*

0x3f90 0x000003f9 0x20000000 0x11111111 0xd0cad0ca
0x3fa0 0x000003fa 0x20000000 0x11111111 0xd0cad0ca
0x3fb0 0x000003fb 0x40000000 0x22222222 0xd0cad0ca
0x3fc0 0x000003fc 0x60000000 0x33333333 0xd0cad0ca
0x3fd0 0x000003fd 0x80000000 0x44444444 0xd0cad0ca
0x3fe0 0x000003fe 0xa0000000 0x55555555 0xd0cad0ca
0x3ff0 0x000003ff 0xe0000000 0x77777777 0xd0cad0ca

* 2 * we need to get the table id together on the address bus,
we will place the table in internal memory island 28:

int TEST_NAME ( void )
{
    volatile mem_lookup_hash_table_t           table_desc;    // microC mule hash structure
    SIGNAL_PAIR                                sig_pair;
    unsigned int                                total_table_buckets, expected;
    mem_lookup_result_in_read_reg_t            *read_result;
    volatile __declspec(write_reg) unsigned int lookup_index[8];
    unsigned int                                base_address = 0xFFFFFFFF & QA70_HASH_TEST_START_ADDRESS;
    unsigned long long                         lookup_addr, bucket_address, search_start_bucket, lookup_entry;

    table_desc.value = 0;                      // clear the struct
    table_desc.hash_lookup = 1;
// Table 9.99 EAS - Tables for HASH lookup must start at a 64KB Boundary,
// i.e. 16-bit aligned address
    table_desc.base_address = HASH_TEST_START_ADDRESS >> 16;
    table_desc.direct_lookup = 0;
    table_desc.hash_command = MULE_HASH_COMMAND;
    table_desc.internal_memory = 1;
    table_desc.table_size = LOOKUP_HASH_TABLE_SIZE_1;
// this will put a 32bit offset on the input data and effectively
    table_desc.start_bit_position = 1;

    // test the last valid index of this size table
    lookup_addr = ( table_desc.value | ISLAND_ENCODING );
    search_start_bucket = total_table_buckets - NUM_MAX_BUCKETS;    // this is bucket 0x3f9
// our expected return value for the 7th bucket from 0x3f9 (which is bucket 0x3ff)
    expected = 0x11111111 * NUM_MAX_BUCKETS;
// this effectively does * 1 *
    write_group_of_buckets ( search_start_bucket, search_start_bucket );
    ...

* 3 * then we need so set up the xfer regs

```

```

...
lookup_entry = (lookup_entry + NUM_MAX_BUCKETS -1 + (BYTES_PER_BUCKET - 16));
lookup_index[0] = 0; // this xfer reg is don't care because start_bit_position is 1,
                    // a 32bit offset on the input data
lookup_index[1] = ( lookup_entry << (test_table_size + 10) | search_start_bucket );
lookup_index[2] = ( lookup_entry >> ( 22 - test_table_size ) );
lookup_index[3] = ( lookup_entry >> ( 54 - test_table_size ) );

read_result = mem_lookup(
    (void *)&lookup_index[0],                                // this effectively does * 3 *
    // this effectively does * 2 * and puts our parameters on the address bus
    __declspec(mem_ptr, addr40) void *)(lookup_addr),
    2,
    sig_done,
    &sig_pair
);
wait_for_all(&sig_pair);

if (read_result->result != expected )
    return TEST_FAIL;
else
    return TEST_PASS;
}

```

and when we run LOOKUP\_HASH\_COMMAND\_LHASHR\_48\_60\_16\_7\_TEST(), TEST\_PASS is returned, read\_result was 0x

-----  
furthermore...

If we change a few of our defines to:

#define TEST_NAME	LOOKUP_HASH_COMMAND_LHASHR_48_60_64_2_TEST
#define NUM_MAX_BUCKETS	2
#define BYTES_PER_BUCKET	64
#define MULE_HASH_COMMAND	LOOKUP_HASH_COMMAND_LHASHR_48_60_64_2
#define WRITE_LHASH_BUCKET	write_lhashr_entry60_64_dataline

and add this function:

```

unsigned long long write_lhashr_entry60_64_dataline(unsigned long long base_addr,
                                                       unsigned long long entryval, unsigned int bucket_val)
{
    write_lhashr_entry60_16_dataline ( base_addr, entryval, bucket_valid );
    write_lhashr_entry60_16_dataline ( base_addr + 16, entryval + 0x10, bucket_valid );
    write_lhashr_entry60_16_dataline ( base_addr + 32, entryval + 0x20, bucket_valid );
    write_lhashr_entry60_16_dataline ( base_addr + 48, entryval + 0x30, bucket_valid );
    return base_addr + 64;
}

```

and then run LOOKUP\_HASH\_COMMAND\_LHASHR\_48\_60\_64\_2\_TEST(), we get

\*memory table\*

```

0xff80 0x000003fe 0x20000000 0x11111111 0xd0cad0ca
0xff90 0x0000040e 0x20000000 0x11111111 0xd0cad0ca
0xffa0 0x0000041e 0x20000000 0x11111111 0xd0cad0ca
0xffb0 0x0000042e 0x20000000 0x11111111 0xd0cad0ca
0xffc0 0x000003ff 0x40000000 0x22222222 0xd0cad0ca
0xffffd0 0x0000040f 0x40000000 0x22222222 0xd0cad0ca
0xfffe0 0x0000041f 0x40000000 0x22222222 0xd0cad0ca
0xffff0 0x0000042f 0x40000000 0x22222222 0xd0cad0ca

```

TEST\_PASS is returned, read\_result was 0x22222222 and lookup\_entry was 0x42f.

## 2.2.39.8 MEM (Memory Lock)

Memory lock commands support locking of specific ranges of memory as opposed to semantic locking supported by queue locking commands. It allows the user to request a lock on a memory region with a specific size and base address. The size of the locked region is specified in the byte mask and its address in the write transfer register. Memory locks are implemented using 16 bit TCAM using the lock128 - lock512 commands. The command memory\_lock allows the programmer to use one command by adjusting the Length field and indirect\_ref.

### Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	lock128	Request a memory lock within a region with specified base address and size. 128 bit TCAM allows a maximum of 4 memory locks. Only need to override the bytemask.
	lock256	Request a memory lock within a region with specified base address and size. 256 bit TCAM allows a maximum of 8 memory locks. Only need to override the bytemask.
	lock384	Request a memory lock within a region with specified base address and size. 384 bit TCAM allows a maximum of 12 memory locks. Need to override the bytemask and length fields.
	lock512	Request a memory lock within a region with specified base address and size. 512 bit TCAM allows a maximum of 16 memory locks. Need to override the bytemask and length fields.
	memory_lock	Request a memory lock within a region with specified base address and size (specified in the Length field). This command replaces all the above commands using indirect_ref to modify the Length field. Need to override the bytemask and length fields.
xfer	All commands	A write transfer register contains a base address of requested lock region. A read transfer register contains the result of operation: <ol style="list-style-type: none"> <li>If a requested region overlaps one of the locked areas, the top 16 bits contain a bit map of matching entries and the bottom 8 bits contain the lower matched entry.</li> <li>The lock TCAM is full – return value is 0x00ff.</li> <li>Successful lock request – return is 0x0080 plus lock entry number in TCAM.</li> </ol>

Parameter	Command	Description
src_op1, src_op2	All commands	Restricted source operands that define the 32 or 40-bit byte address of TCAM which contains memory locks.
opt Tok		sig_done[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2, ...]

## 2.2.39.9 MEM (MicroQ Operations)

These instructions manage a 128 or 256 bits cache line as a FIFO or LIFO lists with 32-bit or 16-bit entries. It is possible to place and remove a single entry at a time in atomic fashion.

Each micro queue has a structure where the first 32 bit word represents its status information and the rest 16 or 32 bit data entries aligned in accordance with the entry size.

The maximum number of entries for 128 bits micro queue is 6 for 16 bit entries and 3 for 32 bit entries. The maximum number of entries for 256 bit micro queue is 14 for 16 bit entries and 7 for 32 bit entries. The status word has the following format:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	Event Source										Reserved	E	S	O	U	RSV	Reserv	N	id												

where:

- Event Source = Event source to present on underflow or overflow of a queue, if event bit 9 is set.
- E = Event bit – set to generate event on micro queue underflow or overflow.
- S = Size of entry bit. When set, it is a micro queue with 32 bit entries otherwise it contains 16-bit entries.
- O = This bit is set on overflow. Cleared only on overwriting the memory.
- U = This bit is set on underflow. Cleared only on overwriting the memory.
- N = Current number of elements in micro queue.
- RSV = Reserved and should be cleared to zero.

### Instruction Format

```
mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <<8], opt Tok // 40-bit addressing
```

### Parameter Descriptions

Parameter	Command	Description
cmd	microq128_get	Read one element of queue in FIFO order for 128-bit cache line into read transfer register.

Parameter	Command	Description
	microq256_get	Read one element of queue in FIFO order for 256-bit cache line into read transfer register.
	microq128_pop	Read one element of queue in LIFO order for 128-bit cache line into read transfer register.
	microq256_pop	Read one element of queue in LIFO order for 256-bit cache line into read transfer register.
	microq128_put	Add one 128-bit queue element by writing contents of write transfer register into memory.
	microq256_put	Add one 256-bit queue element by writing contents of write transfer register into memory.
xfer	microq128_get, microq128_get, microq128_pop, microq128_pop	Read transfer register that contains a micro queue element.
	microq128_put, microq256_put	Write transfer register that contains a new micro queue element.
src_op1, src_op2	All commands	Restricted source operands that define 32 or 40-bit byte micro queue address.
opt_tok	microq128_put, microq256_put  See <a href="#">Note 1</a> .	sig_done[sig_name]
		ctx_swap[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2)
		ind_targets[me1, me2, ...]
	microq128_get, microq256_get, microq128_pop, microq256_pop  See <a href="#">Note 2</a> .	sig_done[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2, ...]

#### Notes:

- When put command is successful, only a single signal is pushed, otherwise on overflow extra data is discarded.
- When an entry is removed from the queue successfully, only a single signal is pushed. When a micro queue is empty (underflow), then two signals will be pushed with sig\_name[1] signaling error.

#### mem[microQ] Description

MicroQ provides a queue structure with a small number of entries, where each entry can be 16 or 32 bits. MicroQ offers a capability to queue up a small number of entries and retrieve them in LIFO or FIFO fashion atomically.

Multiple threads on the same FPC as well as multiple FPCs may contend to push and pop/get microQ entries in atomic fashion, where there is no need for separate locking mechanism. One example use of microQ could be as a way of inter-FPC communication where producer FPC threads put a 32-bit work entry to a microQ and consumer FPC threads do a FIFO get to retrieve work specified by these 32-bit entries. Because the microQ commands are atomic, multiple FPC threads or multiple FPCs can safely queue and retrieve work without concern for data corruption.

The microQ status word provides information if microQ is full on put and empty on a get/pop. One can use the current count of enqueued elements by reading the microQ status word bits[3:0] (# elements). If # elements is zero the microQ is empty and if # elements is equal to the microQ max size then microQ is full.

When an Overflow or Underflow condition occurs from a microQ put or get/pop respectively an event can be generated when status word bit 9 is set. Also just by reading the microQ status word the Overflow and Underflow condition can be learned. Writing zero to microQ memory status word bits [7:6] will clear the Overflow and Underflow bits.

microQ may be created in External, Internal or CT memory.

**Note:** On a microq command put instruction the signal will assert once the pull has been completed to indicate it is safe to access the xfer out register; NOT when the command has been completed and the data in memory is valid. The microq memory location will be written after 10 clock cycles after the signal returns.

### Example 1. mem[microql]

The microQ is setup as 128 bit wide with 16-bit entries and event generation. Demonstrate microq128\_put and microq128\_get commands on microQ in External memory.

```
.sig s1
.xfer $xdata0
.reg $rx[1]
.reg adrlo, adrhi
.reg r_work_data, r_q_init
.reg r_event_status r_embase_addr

; Setup microQ in External memory address 0x3000000000 for microq128
; 16 bit entries. adrhi and adrlo will be used below for all microq commands to
; address microq128 structure in External memory.
immed[adrlo, 0x0]
immed[adrhi, 0x0]
immed_w1[adrhi, 0x3000] ; External memory, 40-bit address

; Enable Event Manager to receive event if overflow or underflow occur on microQ.
; Use cls[read] and cls[write] to do filter act, filter mask, filter match and activate.
; Refer to NFP6xxx Databook "Event Manager Peripheral"
immed32(r_q_init, 0x00ff0200) ; microQ Status word: 16-bit, Enable event gen.
alu[$xdata0 --, B, r_q_init]

; Use 40 bit address thus shift adrhi 8 bits left
mem[write, $xdata0, adrhi, <<8, adrlo, 1], ctx_swap[s1]

; Put work_data in register
immed[r_work_data, 0x1234]

; Put six 16-bit work entries in microQ, status word should show bits[3:0]
; with value of 4b0110 and bits[7:6] == 2b00 (no overflow or underflow)
```

```

; For this example r_work_data contains the same data for each microQ put.
#repeat (6)
alu[$xdata0, --, B, r_work_data]
mem[microq128_put, $xdata0, adrhi, <<8, adrlo], ctx_swap[s1]
#endif

; Note that if another mem[microq128_put] occurs before mem[microq128_get]
; then an overflow will occur and microQ status word bit[7] is set.
; Can read microQ status word and check for errors (checking is not shown).
mem[read, $xdata0 adrh, <<8, adrl, 1], ctx_swap[s1]

; Drain microQ of work data entries as a FIFO. If use microq128_pop
; work data is pulled from workQ as LIFO. The status word bits [3:0]
; will decrement as each work data is retrieved and there is no microQ puts
; occurring.
#repeat(6)
mem[microq128_get, $xdata0, adrhi, <<8, adrlo], ctx_swap[s1]
#endif

; At this point the microQ status should show bits[3:0] == 4b0000
; (no work data entries in workQ) and
; bits[7:6] == 2b00 (no overflow or underflow)
mem[read, $xdata0, adrh, <<8, adrl, 1], ctx_swap[s1]

; Reading the Event Manager status for overflow or underflow event is also
; possible.
immed[r_embase_addr, 0x0000, <<16] ; island id 0 is for current island
immed_w0[r_embase_addr, 0x0200] ; Event Manager offset 0x2000
cls[read, $rx[0], r_embase_addr, <<8, 0x00, 1], ctx_swap[s1] ; filter status

; Check if event occurred. If r_event_status is zero no event occurred.
; alu[] operation will indicate (flag Z=0)
; But if event occurred flag Z=1. Jump to process_event# block to process the
; event.
alu[r_event_status, --, B, $rx[0]]
BEQ[process_event#]

```

## 2.2.39.10 MEM (Packet Engine Operations)

The Packet Engine is located in the Cluster Target Memory (CTM) in proximity to the Microengines. It has access to a 256KB CTM Dcache, which it allocates to incoming buffers. It also allocates a packet number to incoming buffers, thus facilitating packet addressing mode.



### Note

A packet buffer must be allocated prior to addressing.

The Packet Engine provides these buffers to the Microengines for processing. When the Microengines have completed processing the packet, it transmits the packets to the Traffic Manager. The Packet Engine is also a DMA Engine which can move packets between the CTM and the Main Memory. In summary, the packet engine provides the following functionality:

- Storing buffer data as packets
- Support for accessing buffer data stored as packets

- Index packets stored in CTM memory as Packet number plus offset (packet addressing mode)
- DMA packets between CTM DCache and Main Memory.
- Provide packet number to address translation for the Bulk Engine.
- Interface between Microengines and Traffic Manager for Packet Ready Commands.

These commands are used to process packets through the PacketEngine.



## Warning

The number of words an ME wishes to receive is specified in the ref\_cnt field of the AddToWorkQueue Command. The ref\_cnt can be between 1-32. The ME will always receive the first 6 32-bit words starting at offset 0, when it adds itself to the work queue. Next, it will receive (ref\_cnt - 6) 32-bit words starting at the offset it specified. If the ME specifies a ref\_cnt < 6 it will receive 6 32-bit words, regardless. Note that the ref\_cnt the ME provides, includes the first 6 32-bit words it receives, as described above.

Example: If the ME adds itself to the Work Queue with a ref\_cnt '8' and offset '3', it receives the first 6 32-bit words. Next it receives 2 32-bit words starting at offset 3 (12 bytes), at the xfer register it specified.



## Warning

When an ME adds itself to a CTM configured to always send the packet when the first segment is received, it is very important that the sum of the ref\_cnt and (four-byte-aligned) offset does not exceed 128 bytes. For such a 'SendOnFirst' configuration, it may be easier for the ME to set the ref\_cnt to 6 and offset to 0, and it will receive words 0-5, which contain the header information. If the ME needs more information than the header, it can access the Packet using regular MU Commands. For a SendOnLast configuration, the ME should be careful that the sum of the ref\_cnt and offset does not exceed the split length stored in the CTM e.g. 256B, 512B, etc.



## Note

For the packet\_complete\_\* commands, you must add the NBI offset (e.g. 32 bytes) to the length obtained in the NBI Header to get the "total packet length".



## Note

Tip for Programmer: You may have no more than 16 outstanding DMA commands per CTM.



## Warning

If the PacketMode bit is set and the packet is invalid, the address is returned as is. No translation is done.

## Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8, ref_cnt], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	packet_add_thread	Add an FPC thread to the Packet Engine work queue to process incoming packets.
	packet_alloc	Allocate packet buffer. ref_cnt can be one of four values ranging from 0 to 3. 2b00:256B, 2b01: 512B, 2b10: 1KB, 2b11: 2KB. The command Address[1:0] specify master number where valid values are 2b00: ME, 2b01: NBI-0 and 2b10: NBI-1.  The Packet Engines responds with push data to the master. Read Transfer register will contain Packet Number in data[29:20], packet credit# in data[19:9] and buffer credit# in data[9:0]. If the packet cannot be allocated because either or both the PacketAllocationTable or the MemoryAllocation Table are full, the allocation request waits for a FreePacketRequest to arrive.
	packet_alloc_poll	The packet_alloc instruction should be used to allocate packets in the CTM when the CTM is shared between the MEs and the NBI for packet allocations. The MEs are expected to keep track of both packet and buffer credits while using packet_alloc. The packet_alloc command should be sent only when the MEs have non-zero packet and buffer credits available.
		Allocate packet buffer. See packet_alloc for specifying Packet length and master number.  The Packet Engines responds with push data to the master. The push data is similar to packet_alloc. But with packet_alloc_poll command if a Packet cannot be allocated, a response is sent immediately to the master indicating that the packet cannot be allocated at this point. This response contains all 1s in the data field.

Parameter	Command	Description
		The packet_alloc_poll instruction should be used to allocate packets in the CTM when the MEs are the only packet allocation master in that CTM. The MEs are not required to keep track of packet or buffer credits while using packet_alloc_poll.
	packet_complete_drop	<p>When the drop packet token is used in the packet_complete command, it is for keeping track of sequence numbers only. The packet is not dropped, since the command may be sent to multiple NBI blocks. The drop token is used for updating the packet sequencers, and descriptor is not stored in memory. Therefore the packet is not dropped, and no memory buffers are freed as a result.</p> <p>Since a packet can egress from either NBI, each NBI needs to keep track of each sequence number. One NBI gets the packet itself, and will subsequently free that packet, and the other NBI gets the packet command with the drop token in order to maintain packet sequencing. If the intention of system software is to drop the packet, then the packet must also be freed in CTM memory and the MU buffer returned to the free list.</p>
	packet_complete_multicast	Packet Processing completed. Multicast Packet.
	packet_complete_multicast_free	Packet Processing completed. Multicast Packet and Free Packet.
	packet_complete_unicast	Packet Processing completed. Unicast Packet.
	packet_credit_get	<p>Return packet and buffer credits.</p> <p>The Packet Engine responds with push data. The response is sent irrespective of the availability of packets or buffer credits. Returned data is buffer credit in data[9:0] and packet credit in data[19:9].</p>
	packet_free	Free Packet.
	packet_free_and_return_pointer	Free Packet and push MU Pointer used for the freed packet.
	packet_free_and_signal	Free Packet and push a signal when the free completes.

Parameter	Command	Description
	packet_read_packet_status	Read status of packet. Respond with an error if all threads are busy.
	packet_return_pointer	Return MU Pointer available in packet header.
	packet_wait_packet_status	Read the status of a packet. Respond with status only after the last segment of the packet becomes available. Respond with an error if all threads are busy.
	pe_dma_from_memory_buffer	DMA memory range from External Memory to CTM Memory.
	pe_dma_from_memory_buffer_le	DMA memory range from External Memory to CTM Memory. Use little-endian addressing.
	pe_dma_from_memory_buffer_le_swap	DMA memory range from External Memory to CTM Memory. Use little-endian addressing. Byte swap transferred data.
	pe_dma_from_memory_buffer_swap	DMA memory range from External Memory to CTM Memory. Byte swap transferred data.
	pe_dma_to_memory_buffer	DMA memory range from CTM memory to External Memory.
	pe_dma_to_memory_buffer_le	DMA memory range from External Memory to CTM Memory. Use little-endian addressing.
	pe_dma_to_memory_buffer_le_swap	DMA memory range from CTM Memory to External Memory. Use little-endian addressing. Byte swap transferred data.
	pe_dma_to_memory_buffer_swap	DMA memory range from CTM Memory to External Memory. Byte swap the transferred data.
	pe_dma_to_memory_indirect	DMA packet buffer contents from CTM Memory to External memory buffer, where packet number is encoded in address[9:0], and memory address is encoded in packet.
	pe_dma_to_memory_indirect_free	DMA packet buffer contents from CTM Memory to External memory buffer, where packet number is encoded in address[9:0] and memory address is encoded in the packet. Free packet buffer after DMA completes.
	pe_dma_to_memory_indirect_free_swap	DMA packet buffer contents from CTM Memory to External memory buffer, where packet number is encoded in address[9:0] and memory address is encoded in the packet. Free packet buffer after DMA completes. Byte swap packet contents.

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
	pe_dma_to_memory_indirect_swap	DMA packet buffer contents from CTM Memory to External memory buffer, where packet number is encoded in address[9:0] and memory ddress is encoded in the packet. Byte swap packet contents.
	pe_dma_to_memory_packet	DMA packet buffer contents from CTM Memory to External Memory buffer specified with the command.
	pe_dma_to_memory_packet_free	DMA packet buffer contents from CTM Memory to External Memory buffer specified with the command. Free packet buffer after DMA completes.
	pe_dma_to_memory_packet_free_swap	DMA packet buffer contents from CTM Memory to External Memory buffer specified with the command. Byte swap the transferred data.
	pe_dma_to_memory_packet_swap	DMA packet buffer contents from CTM Memory to External memory buffer specified with the command. Byte swap transferred data.
xfer	packet_alloc, packet_alloc_poll, packet_credit_get, packet_read_packet_status, packet_wait_packet_status,	Read transfer register containing 32-bit result or status.
	packet_free_and_return_pointer, packet_return_pointer, pe_dma_to_memory_indirect, pe_dma_to_memory_indirect_free, pe_dma_to_memory_indirect_free_swap, pe_dma_to_memory_indirect_swap	Read transfer registers containing 64-bit result or status.
	packet_add_thread	Read transfer registers containing packet data of size specified in CPP Length field.
	packet_complete_drop, packet_complete_multicast, packet_complete_multicast_free, packet_complete_unicast, packet_free, packet_free_and_signal, pe_dma_from_memory_buffer, pe_dma_from_memory_buffer_le, pe_dma_from_memory_buffer_le_swap, pe_dma_from_memory_buffer_swap, pe_dma_to_memory_buffer, pe_dma_to_memory_buffer_le_swap, pe_dma_to_memory_buffer_swap, pe_dma_to_memory_packet, pe_dma_to_memory_packet_free,	Not Required/Ignored, should be “--”.

Parameter	Command	Description
	pe_dma_to_memory_packet_free_swap, pe_dma_to_memory_packet_swap	
src_op1, src_op2	All Commands	Restricted source operands that define address. The address is specified by src_op1 + src_op2. The address encoding specifies CPP fields for a command and is defined in the NFP-6xxx Databook Table “Cluster-Target MU Detailed CPP command map ” Address column.
ref_cnt	packet_add_thread	Reference count in increments of 4 byte words. Valid values are 6 to 32. Values above must be specified using indirect reference.
	packet_alloc, packet_alloc_poll	Reference count value is immediate push to CPP LENGTH field. Valid values are 0, 1, 2 and 3.
	packet_complete_drop, packet_complete_multicast, packet_complete_multicast_free, packet_complete_unicast, pe_dma_from_memory_buffer, pe_dma_from_memory_buffer_swap, pe_dma_to_memory_buffer, pe_dam_to_memory_buffer_le, pe_dma_to_memory_buffer_le_swap, pe_dma_to_memory_buffer_swap, pe_dma_to_memory_packet, pe_dma_to_memory_packet_free, pe_dma_to_memory_packet_free_swap, pe_dma_to_memory_packet_swap	Not Required. See <a href="#">Section 2.1.6.3</a>
opt_tok	packet_add_thread, packet_alloc, packet_alloc_poll, packet_credit_get, packet_free_and_return_pointer, packet_free_and_signal, packet_read_packet_status, packet_return_pointer, packet_wait_packet_status, pe_dma_from_memory_buffer, pd_dma_from_memory_buffer_le, pe_dma_from_memory_buffer_le_swap, pe_dma_from_memory_buffer_swap, pe_dma_to_memory_buffer, pe_dma_to_memory_buffer_swap, pe_dma_to_memory_indirect, pe_dma_to_memory_indirect_free, pe_dma_to_memory_indirect_swap, pe_dma_to_memory_indirect_free_swap,	sig_done[sig_name] ctx_swap[sig_name] defer[n] where n = 1 or 2 indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2,..]

Parameter	Command	Description
	pe_dma_to_memory_packet, pe_dma_to_memory_packet_free, pe_dma_memory_packet_free_swap, pe_dma_to_memory_packet_swap	
	packet_complete_drop, packet_complete_multicast, packet_complete_multicast_free, packet_complete_unicast, packet_free	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

## 2.2.39.11 MEM (Push Queue Descriptor)

This command reads an entire queue descriptor into the read transfer registers. For details of the queue descriptor, see the NFP-6xxx Databook, MU Queue Engine Operational Details section "Push Queue Descriptor".

### Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	push_qdesc	Read a queue descriptor at the specified queue array entry into read transfer registers.
xfer	push_qdesc	4 read transfer registers to contain the entire queue descriptor. See <a href="#">Note 1</a> .
src_op1, src_op2	push_qdesc	Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"><li>• [9:0] – Queue array entry number.</li></ul>
opt Tok	push_qdesc	sig_done[sig_name]
		ctx_swap[sig_name]
		defer[n] where n = 1 or 2
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2, ...]

### Notes:

- Refer to [Section 2.2.39.12.1](#) for the format of the queue descriptor.

## 2.2.39.12 MEM (Queue)

The queue engine supports two classes of queues – circular buffers (rings) and linked lists.



### Note

The RingSize for circular buffer is never explicitly loaded; it is part of just the CellCount (for QueueType==2), or the CellCount, SOP and EOP (for other QueueTypes). For details of CellCount and the queue descriptor, see the NFP-6xxx Databook, MU Queue Engine Operational Details section "Queue Descriptor Overview".

### 2.2.39.12.1 NFP-6xxx Queue Descriptor Format

The use of queue descriptors depends on the configuration of the queue engine. Please refer to link list queue descriptor formats in [Section 2.2.39.3](#) and circular queue descriptor formats in [Section 2.2.39.18](#).

### 2.2.39.13 MEM (Queue Lock Operations)

These commands manage a 128 or 256 bit cache line as a FIFO of semantic lock claimants.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																						E	S	O	U	RSV	L		N		

where:

- Event Source = Event source to present on underflow or overflow of a queue, if event bit 9 is set.
- E = Event bit – set to generate event on micro queue underflow or overflow.
- S = Size of entry bit. When set, it is a micro queue with 32 bit entries otherwise it contains 16-bit entries.
- O = This bit is set on overflow. Cleared only on overwriting the memory.
- U = This bit is set on underflow. Cleared only on overwriting the memory.
- L = Used only for lock queue commands. When set indicates the lock has been given out and any LockClaim will be queued. For more, see [Note](#) below.
- N = Current number of elements in micro queue.
- RSV = Reserved and should be cleared to zero.

### Instruction Format



### Note

If the Queue is Unlocked (bit[4]==0) then the rest of the memory fields are undefined, except that bits[4:0] should be zero. If Locked with an empty queue (bit[5:0]==5b10000) then the remaining contents are undefined. If Locked with a non-empty queue (bit[4]==1b1 and bit[4:0] are non-zero) then the remaining contents, up to the queue length are valid; remaining memory is undefined.



## Note

Underflowing on a Queue Lock operation has undefined operation.

```
mem[cmd, --, src_op1, src_op2], opt_tok // 32-bit addressing
mem[cmd, --, src_op1, <<8, src_op2], opt_tok // 40-bit addressing
mem[cmd, --, src_op1, src_op2, <<8], opt_tok // 40-bit addressing
```

## Parameter Descriptions

Parameter	Command	Description
cmd	queue128_lock	Request a lock within 128 bits cache line at the specified address. If command is not successful, an error signal is pushed.
	queue256_lock	Request a lock within 256 bits cache line at the specified address. If command is not successful, an error signal is pushed.
	queue128_unlock	Release a lock within 128 bits cache line at the specified address.
	queue256_unlock	Release a lock within 256 bits cache line at the specified address.
--	All commands	Must always be “--”.
src_op1, src_op2	All commands	Restricted source operands that define 32 or 40-bit byte locks queue address.
opt_tok	queue128_lock, queue256_lock	sig_done[sig_name] indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
	See <a href="#">Note 1</a> .	ind_targets[me1, me2, ...]
	queue128_unlock, queue256_unlock	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2, ...]

## Notes:

- When a lock is successful, only a single signal is pushed otherwise two signals are pushed with sig\_name[1] signaling error.

## 2.2.39.14 MEM (Read and Write)

Move data between Memory Unit and data masters.



## Note

When reading the last 64 bit word in DRAM memory space, you must use an 64-bit-aligned read. DRAM reads that go beyond the end of physical memory, can cause cache corruption in later, unrelated, DRAM transactions.

## Instruction Formats

```
mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt_tok // 40-bit addressing
```

```
mem[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt Tok      //40-bit addressing
```

## Parameter Descriptions

Parameter	Command	Description
cmd	read, read_be	64-bit read from memory unit in big endian (LWBE) format.
	read_le	64-bit read from memory unit in little endian format.
	read_swap, read_swap_be	64-bit read from memory unit in big endian (LWBE) format with byte swapping.
	read_swap_le	64-bit read from memory unit in little endian format with byte swapping.
	read32, read32_be	32-bit read from memory unit in big endian (LWBE) format.
	read32_le	32-bit read from memory unit in little endian format.
	read32_swap, read32_swap_be	32-bit read from memory unit in big endian (LWBE) format with byte swapping.
	read32_swap_le	32-bit read from memory unit in little endian format with byte swapping.
	read8, read8_be	8-bit read from memory unit in big endian (LWBE) format.
	write, write_be	64-bit write to memory unit in big endian (LWBE) format.
	write_le	64-bit write to memory unit in little endian format.
	write_swap, write_swap_be	64-bit write to memory unit in big endian (LWBE) format with byte swapping.
	write_swap_le	64-bit write to memory unit in little endian format with byte swapping.
	write32, write32_be	32-bit write to memory unit in big endian (LWBE) format.
	write32_le	32-bit write to memory unit in little endian format.
	write32_swap, write32_swap_be	32-bit write to memory unit in big endian (LWBE) format with byte swapping.
	write32_swap_le	32-bit write to memory unit in little endian format with byte swapping.
	write8, write8_be	8-bit write to memory unit in big endian (LWBE) format.
	write8_le	8-bit write to memory unit in little endian format.
	write8_swap, write8_swap_be	8-bit write to memory unit in big endian (LWBE) format with byte swapping.
	write8_swap_le	8-bit write to memory unit in little endian format with byte swapping.
xfer	read, read_be, read_le read_swap, read_swap_le, read_swap_be, read32, read32_be,	Read transfer registers.

Parameter	Command	Description
	read32_le, read32_swap, read32_swap_be, read32_swap_le, read8, read8_be	
	write, write_be, write_le, write_swap, write_swap_be, write_swap_le, write32, write32_be, write_32_le, write32_swap, write32_swap_be, write32_swap_le, write8, write8_be, write8_le, write8_swap, write8_swap_be, write8_swap_le	Write transfer registers.
src_op1, src_op2	All commands	Restricted source operands that define byte address. The address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a>
ref_cnt	read, read_be, read_le, read_swap, read_swap_le, read_swap_be, write, write_be, write_le, write_swap, write_swap_be, write_swap_le	Reference count in 8-byte words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
	read32, read32_be, read32_le, read32_swap, read32_swap_be, read32_swap_le, write32, write32_be, write32_le, write32_swap, write32_swap_be, write32_swap_le	Reference count in 4-byte words. Valid values are 1-32. Values above 8 must be specified using indirect reference.
	read8, write8, write8_be, write8_le, write8_swap, write8_swap_be, write8_swap_le	Reference count in bytes. Valid values are 1-32. Values above 8 must be specified using indirect reference.
opt Tok	All commands	sig_done[sig_name]

Parameter	Command	Description
		ctx_swap[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a>
		defer[n] where n = 1 or 2
		ind_targets[me1, me2, ...]

## 2.2.39.15 MEM (Read/Write Queue Descriptor)

These commands are applicable both to lists and rings.

### Instruction Format

mem[cmd, --, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, --, src_op1, <<8, src_op2], opt Tok // 40-bit addressing
mem[cmd, --, src_op1, src_op2, <<8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	rd_qdesc	Read NFP-6xxx queue array descriptor from memory into queue array entry. A queue array entry number is specified in the data reference. See <a href="#">Note 1</a> .
	wr_qdesc	Write a queue array entry into a queue descriptor in memory. A queue array entry number is specified in the data reference. See <a href="#">Note 1</a> .
--	Both commands	Must always be “--”.
src_op1, src_op2	Both commands	Restricted operands are added (src_op1 + src_op2) to define the byte address of the NFP-6xxx queue descriptor structure in memory. This address must be aligned to a 16-byte boundary.
opt Tok	Both commands	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

### Notes:

1. A queue array entry number has to be specified in the data reference field of indirect format. An example using a queue array entry number 0x11 is:

In the recommended V2 Indirect Reference Format set OVE\_DATA field to 0x1 (Over-rides the full data reference). Set the PREV\_ALU.DATA16[13:0] = 0x11. Refer to section [Section 2.1.6.4.6.2](#)

Note that the above covers only those fields that had to be overridden, and not those that could optionally be overridden. If one wanted to override say the data master as well, the V2 Indirect Reference mode can support this additional override field and more. The V1 Indirect Reference mode has 13 Override formats and not all combinations are supported. And this is why V2 Indirect Reference mode is preferred over V1 Indirect Reference mode.

V1 Indirect Reference Format the Encoding field is 0x0 and data reference filed is 0x11. Refer to section [Section 2.1.6.4.5](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoding	Reserved							Queue array entry number							Reserved																
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0

## 2.2.39.16 MEM (Ring Add to Tail)

This command adds a specified amount to the ring tail pointer, wrapping it appropriately. The same amount is added to the number of buffer entries.



### Note

The command does not ensure that the resulting number of entries is less than or equal to the ring size. The invoking software is responsible for ensuring that the number of entries will not exceed the ring size.



### Note

MUConfigCPP[DirAccWays] has no effect on the Queue Engine Command add\_tail. This command uses the QueueLoc field of the Queue Descriptor to determine whether memory access is to the External Memory or direct access (to DCache).

### Instruction Format

mem[cmd, --, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, --, src_op1, <<8, src_op2], opt Tok // 40-bit addressing
mem[cmd, --, src_op1, src_op2, <<8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	add_tail	Increase a number of ring entries by a specified amount. The tail is wrapped to the size of the ring. The resulting number of entries is not constrained to be within the ring size and it is up to the caller to ensure this.
--	add_tail	Must always be “--”.
src_op1, src_op2	add_tail	Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>[23:0] - Number of ring entries to add to the tail.</li> </ul>
opt Tok	add_tail	indirect_ref. (See <a href="#">Note 1</a> )

### Notes:

1. A queue array entry number has to be specified in the data reference field of an indirect format. An example of using a queue array number 0x11 is:

In the recommended V2 Indirect Reference Format set OVE\_DATA field to 0x1 (Overrides the full data reference). Set the PREV\_ALU.DATA16[13:0] = 0x11. Refer to section [Section 2.1.6.4.6.2](#)

V1 Indirect Reference Format the Encoding field is 0x0 and data reference filed is 0x11. Refer to section [Section 2.1.6.4.5](#)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Encoding	Reserved								Queue array entry number								Reserved														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0

## 2.2.39.17 MEM (Ring Put, Journal, Get and Pop Operations)

These commands are only applicable to rings (circular buffers) and add (put / journal) or remove from head (get) or tail (pop) a requested number of entries from a circular buffer. After this command has been completed, head or tail pointers are updated appropriately.

### Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	get	Remove a ref_cnt number of entries from the head of a circular buffer. A double signal is returned if a buffer does not have sufficient words to satisfy the request.
	get_eop	Remove a ref_cnt number of entries from the head of a circular buffer. This command is used for type 2 rings. A double signal is returned in two cases:  1. If a buffer does not have sufficient words to satisfy the request.  2. The EOP bit is set in the queue descriptor.  The EOP bit is cleared by this command.
	get_freely	Remove a specified number of 32-bit words (entries) from the head of a circular buffer. If there are not enough entries in the circular buffer to support the request for “length” words, then zeros are returned for the remaining length after valid entries are removed. No error signal is generated.
	get_safe	See: get_freely
	pop	Remove a ref_cnt number of entries from the tail of a circular buffer. A double signal is returned if a buffer does not have sufficient words to satisfy the request.

Parameter	Command	Description
	pop_eop	<p>Remove a ref_cnt number of entries from the tail of a circular buffer. This command is used for type 2 rings. A double signal is returned in two cases:</p> <ol style="list-style-type: none"> <li>1. If a buffer does not have sufficient words to satisfy the request.</li> <li>2. The EOP bit is set in the queue descriptor.</li> </ol> <p>The EOP bit is cleared by this command.</p>
	pop_freely	Remove a specified number of 32-bit words (entries) from the tail of a circular buffer. If there are not enough entries in the circular buffer to support the request for “length” words, then zeros are returned for the remaining length after valid entries are removed. No error signal is generated.
	pop_safe	See: pop_freely
	put	Add a ref_cnt number of entries to a circular buffer. If this command fails, the read transfer register will contain 0. See <a href="#">Note 1</a> .
	journal	Add a ref_cnt number of entries to a circular buffer. On overflow, the oldest data in the ring will be overwritten and the EOP bit will be set.
xfer	get, get_eop, get_freely, get_safe, pop, pop_eop, pop_freely, pop_safe	Read transfer registers containing data read from ring entries.
	put, journal	Write transfer registers containing data to be written to ring entries. For put, a single long word is also pushed to the first read transfer register. See <a href="#">Note 1</a> .
src_op1, src_op2	get, get_eop, get_freely, get_safe, pop, pop_eop, pop_freely pop_safe, put, journal	<p>Restricted operands (src_op1 + src_op2) specify qa_index in address[9:0].</p> <ul style="list-style-type: none"> <li>• qa_index[9:0] == Queue array entry number in Queue Engine SRAM.</li> </ul> <p>See <a href="#">Note 3</a> for how qa_index is used.</p>
ref_cnt	All commands	Reference count in increments of 4 byte words. Valid values are 1 to 16. Values above 8 must be specified using indirect reference.
opt Tok	get, get_eop, pop, pop_eop, put	<p>sig_done[sig_name]. See <a href="#">Note 2</a>.</p> <p>indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a>.</p> <p>ind_targets[me1, me2, ...]</p>
	journal, get_freely, get_safe, pop_freely, pop_safe	<p>sig_done[sig_name]</p> <p>ctx_swap[sig_name]</p> <p>defer[n] where n = 1 or 2</p>
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

Parameter	Command	Description
		ind_targets[me1, me2, ...]

**Note:**

1. Long word pushed to first read transfer register by the put command:

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

Result (bit 31) will be set when the data was placed on the ring.

2. If there are insufficient words in the ring for get and pop commands, two signals will be pushed, where sig\_name[1] signals error. Put command instructions always return two signals.

If the EOP bit is set for get\_eop and pop\_eop commands, two signals will be pushed, where sig\_name[1] signals that the ring overflowed on a journal command.

3. There are 1024 queue descriptor's which the Queue engine can work on. The qa\_index specifies which Queue Descriptor it needs to work on. These Queue Descriptors are stored in internal SRAM of Queue engine and is referred to as Queue Array. Queue Descriptors that are out in external memory or Dcache can be brought inside the Queue engine SRAM with mem(rd\_qdesc) command. The mem(wr\_qdesc) is used to write a Queue Descriptor from Queue Array to external memory or Dcache. After these operations are completed and Queue Array is not empty can RingPut, RingPop and other commands be used in specifying which qa\_index it needs to work on. The Queue engine will know which Queue Descriptor corresponds to which Queue Array entry specified by qa\_index.

## 2.2.39.18 MEM (Rings/Circular Buffers)

The queue descriptor can be configured as type 0-3 for rings. Type 2 however offers the most flexibility as the ring can hold up to 16M long words. There is also no 64 long word fullness threshold for types 0, 1 or 3 which results in better ring utilization, especially for smaller ring sizes. When used as journals, type 2 also offers an indication that an overflow occurred via the EOP bit in the descriptor.

Below shows the Ring descriptors fields for Types 0, 1, 2 and 3.

### 2.2.39.18.1 Ring Descriptor – Type 0, 1 and 3 queue

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ring Size	Reserved																													0	0
																															Ring Type

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
q_loc	Reserved	q_page	q_count																												
Reserved																													Reserved		

### Ring Size Encoding

The ring size is encoded in bits 29-31 of the first long word of the descriptor.

**Table 2.39. Ring Size Encoding for type 0, 1 and 3 queue**

Bit value	Ring Size (Long Words)
000	512
001	1K
010	2K
011	4K
100	8K
101	16K
110	32K
111	64K

### Ring Type Encoding

The ring type is encoded in bits 0 and 1 of the second long word of the descriptor.

**Table 2.40. Ring Type Encoding**

Bit value	Ring Type
00	Type 0
01	Type 1
10	Type 2
11	Type 3

### Queue Locality (q\_loc)

The memory unit handles the full CPP command address, but it does not support 1TB of memory. Instead, the address contains some further information about the type of memory being accessed in a command, plus some high order bits are also ignored. The top two bits of a 40-bit memory unit address in fact encode the memory addressing type as in the following table:

**Table 2.41. Memory Unit Addressing Modes**

Bit value	Addressing mode
00	High locality of reference data
01	Low locality of reference data
10	Direct access
11	Discard after read

### 1. High Locality of Reference Data

When memory is accessed as high locality of reference it is accessed through the cache system, and the cache lines are used as write-back on eviction and marked as 'preferred to keep'. This latter means that the cache line will only be evicted if no low locality of reference cache lines are in the same set.

### 2. Low Locality of Reference Data

When memory is accessed as low locality of reference it is accessed through the cache system, and the cache lines are used as write-back soon and marked as 'preferred to evict'. This latter means that the cache line will be evicted in preference to high locality of reference cache lines in the same set.

### 3. Direct Access Mapping

A direct access uses the bottom 21 bits of the supplied address as the actual data cache address to access. This mechanism avoids using the cache tag system entirely, and is therefore faster than cached accesses, even locked cache accesses. It must be used with care, as it would be unwise to use a location in cache memory for both direct access and cached accesses; therefore the cache memory location should be mapped in the cache tag array to a locked location that is never accessed; this will prevent eviction of the cache line and dual use of the cache memory.

To aid this, the cache addresses for every way 0 of a cache set is mapped to the bottom 1/8th of the data cache; way 1 is mapped to the next 1/8th of the data cache, and so on. Therefore the data cache can be split at a granularity of 1/8ths for direct access, from the bottom upwards, by locking way 0 up to way 'n' for (n+1) eighths of the cache.

A further assist is supplied with a configuration register which controls the regions of the data cache that are permitted to be accessed with 'direct access'. This is an eight bit register, one bit per way. The register should be 0 if no direct accesses are permitted - in this case, the request for 'direct access' is ignored entirely, and high locality accesses are requested for every 'direct access' request. If only direct accesses are permitted then the register should be set to all 1's, in which case all accesses are mapped on to 'direct access'. Finally, if some of the bits are set then direct accesses to the cache regions indicated (bit 0 indicating the bottom 1/8th of the data cache, etc) is permitted, and a direct access request to a region where the corresponding bit is clear will lead to a high locality access.



#### Note

In the event that the DDR channels are not populated for any given External Memory Unit, that Memory Unit is still available in the system as a psuedo 'Internal' Memory Unit. It is software's responsibility to configure the unit for and address it only in direct access mode for this scenario.

#### 4. Discard after Read

When memory is accessed as discard after read it is accessed through the cache system, and the cache lines are *marked as clean and not written back* to DDR memory; the cache line preference is unchanged (or, if a cache line fill is required to accommodate the access, the cache line is marked as 'preferred to evict').

This is a potentially dangerous mode of operation, as any writes that have been performed to the data will be lost. This operating mode should only be used for transient data, such as packets on their final transmission from a system.

#### Queue Page (q\_page)

A 2-bit value used for the top bits of the queue entry addresses; all queue items must be within this window of memory, as these bits are just prepended.

#### 2.2.39.18.2 Ring Descriptor – Type 2 queue

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ring Size	Reserved																											EOP	0		
																													1	0	
q_loc	Reserved	q_page																											q_count		
																												Reserved	Reserved		

#### Note:

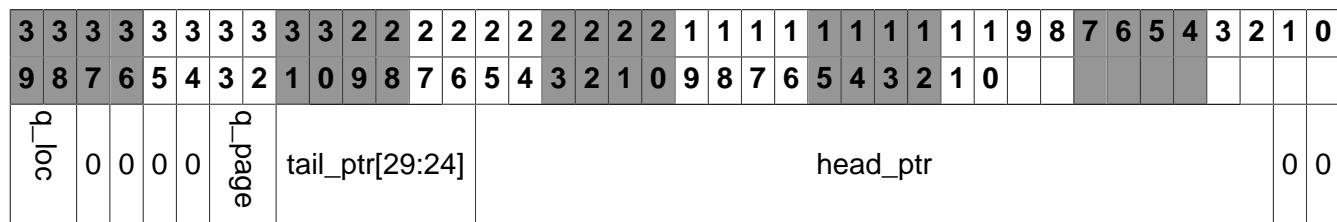
1. The EOP bit is used for overflow indication when used with the journal, get\_eop and pop\_eop commands.

**Table 2.42.** Ring Size encoding for Type 2 Rings

<b>Bit value</b>	<b>Ring Size (32 bit Words)</b>
0000	512
0001	1K
0010	2K
0011	4K
0100	8K
0101	16K
0110	32K
0111	64K
1000	128K
1001	256K
1010	512K
1011	1M
1100	2M
1101	4M
1110	8M
1111	16M

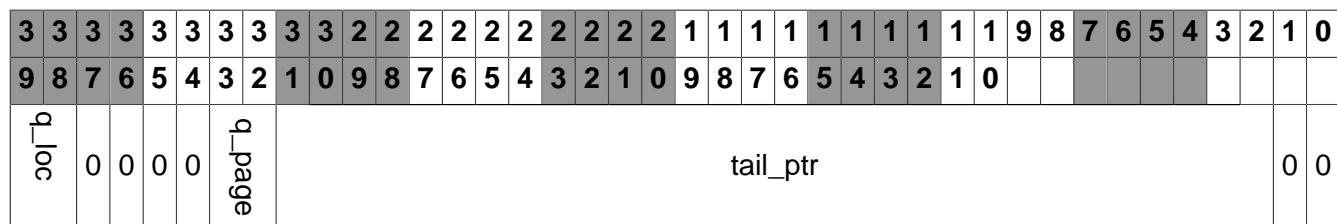
### **Memory address of head of circular buffer**

The byte address of the head of a circular buffer is given by {q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:24], and head\_ptr[23:0], 2b0}.



### **Memory address of tail of circular buffer**

The byte address of the tail of a circular buffer is given by  $\{q\_loc[1:0], 4b0, q\_page[1:0], tail\_ptr[29:0], 2b0\}$ .



## Put Status

<b>31</b>	<b>30</b>	<b>29</b>	<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>	<b>20</b>	<b>19</b>	<b>18</b>	<b>17</b>	<b>16</b>	<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>		
<b>Result</b>	0	0	0	0	0	0	0																										

Result (bit 31) is 1 when there is enough space on ring for put data.

For type 2 rings the result will be 0 if, after the put operation, the ring would be full or its size exceeded. This is unlike other ring sizes where there is a 64 long word margin.

## Memory alignment

A circular buffer must be aligned to a multiple of its size, for example a 512 word ring must be byte aligned to 2048.

## Example of Ring (Circular) Queue

Ring queues are configured to a specified size and are specified in the Ring Descriptor. There are four type of ring descriptors: Type 0, 1, 2 and 4. Type 2 offers the most flexibility as the ring can hold up to 16M 32-bit words while Type 0, 1 and 3 can hold up to 64K 32-bit words.

Ring descriptors contain the head and tail pointer, Queue count, Ring Size, Locality, page and EOP.

This example will show how to build a Ring Descriptor, read descriptor in to the Queue Engine QArray, put an entry on the ring, and pop an entry off from the ring.

```

/*
 * init Ring queue descriptor in QArray from fields, nfp format. used to init
 * Ring to initial ( empty ) condition.
 *
 * _ddr_addr           - dram location to use for descriptor. must
 *                       be 16 byte aligned
 * _ddr_loc            - locality of mem w/queue desc
 * _ddr_ilnd           - island id of mem w/queue desc
 * _que_no             - que number, 0-1023
 * _que_addr           - base memory location of queue data.
 *                       must be lword aligned ( bits 1:0 = 0 )
 *                       and max of 24 bits ( types 0,1,3 ) or
 *                       max of 30 bits ( type 2 )
 * _page, _type, _loc - page, type, loc fields. 2 bits each
 * _size               - ring size. 3 bits (types 0,1,3) or
 *                       4 bits (type 2)
 *
 */
#define _queRingQDescInit( _ddr_addr, _ddr_loc, _ddr_ilnd, _que_no,\ 
                        _que_addr, _size, _page, _type, _loc )
.begin
    .sig s1
    .reg r_addr, r_que_no, r_head_ptr_mask, r_tail_ptr_mask, r_tmp, r_loc_ilnd
    .reg write $xfer0[4]
    .reg read $xfer1[4]
    .xfer_order $xfer0
    .xfer_order $xfer1

    r_head_ptr_mask = 0x03fffffc

```

```

r_tail_ptr_mask = 0xfffffffffc

// prepare descriptor
// word 0
r_tmp = _que_addr < r_head_ptr_mask ; Head pointer
.if (_type == 2)
    r_tmp = r_tmp | (_size << 28) ; size, type 2
.else
    r_tmp = r_tmp | (_size << 29) ; size, type 0,1,3
.endif
$xfer0[0] = r_tmp ; write word 0 to write transfer reg

// word 1
r_tmp = _que_addr < r_tail_ptr_mask // tail.
r_tmp = r_tmp | _type // type
$xfer0[1] = r_tmp

// word 2
r_tmp = 0
r_tmp = r_tmp | (_loc << 30) // loc
r_tmp = r_tmp | (_page << 24) // page
$xfer0[2] = r_tmp

// word 3
$xfer0[3] = 0 // reserved

// Write descriptor to ddr, 40-bit address,
// EXT MEM 0 Island ID 0x18
// MU Access mode (_ddr_loc= 2b10)
r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
r_loc_ilnd = r_loc_ilnd << 24
r_addr = _ddr_addr
mem[write, $xfer0[0], r_loc_ilnd, <<8, r_addr, 2], ctx_swap[s1]

// Read back to ensure data reaches DRAM
mem[read, $xfer1[0], r_loc_ilnd, <<8, r_addr, 2], ctx_swap[s1]
.end
#endifm

/*
 * read queue descriptor from memory into QArray
 *
 * _queue_no      - QArray number 0-1023
 * _queue_addr    - 16 byte aligned mem location of initialized QDesc
 * _buf_loc       - locality of mem w/queue desc
 * _buf_ilnd      - island ID of mem w/queue desc
 */
#macro _queReadDesc_nfp(_queue_no, _queue_addr, _buf_loc, _buf_ilnd)
.begin
    .sig s1
    .reg r_tmp, r_loc_ilnd
    .reg read $xdebug[4]
    .xfer_order $xdebug

    r_loc_ilnd = _buf_ilnd | (_buf_loc << 6)
    r_loc_ilnd = r_loc_ilnd << 24

    ; Use NFP6xxx Indirect reference where Queue no is written

```

```

; to the prev_alu Data16 field and OVE_DATA value is 8
; indicating load prev_alu DATA[13:0] into cpp_command.data_ref[13:0]
r_tmp = _queue_no << 16 ; Load queue no in register for next instruction
alu[--, 8, +, r_tmp]      ; Set prev_alu[7] and prev_alu DATA16
mem[rd_qdesc, --, r_loc_ilnd, <<8, _queue_addr], indirect_ref

; Can use push_qdesc to debug the QArray contents.
mem[push_qdesc, $xdebug[0], r_loc_ilnd, <<8, _queue_no], ctx_swap[s1]
nop
nop
.end
#endif

/*
 * Add entry to ring queue
 *
 * _ddr_qlink_addr - dram location to use for descriptor. must
 *                    be 16 byte aligned
 * _ddr_ilnd       - island id of mem w/queue desc
 * _ddr_loc        - locality of mem w/queue desc
 * _que_no         - que number, 0-1023
 *
 */
#endif

#define _enqueueBufRing(_ddr_qlink_addr, _ddr_ilnd, _ddr_loc, _que_no)
.begin
    .sig s1
    .reg r_tmp
    .reg r_loc_ilnd
    .reg r_buf_addr, r_qadr
    .reg $xd0[16]
    .xfer_order $xd0

    r_buf_addr = _ddr_qlink_addr
    r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
    r_loc_ilnd = r_loc_ilnd << 24

    ; Set SOP and EOP meaning that one packet is contained in 1 buffer.
    ; This way when dequeue is executed the QCount will decrement.
    r_tmp = 4
    alu[--, r_tmp, +, 8]
    mem[put, $xd0[0], r_loc_ilnd, <<8, _que_no, 1], indirect_ref, sig_done[s1]
    ctx_arb[s1]

.end
#endif

/*
 * Remove entry from the head of the ring queue
 * Returns the Qdesc head pointer, !EOP and !SOP
 * _ddr_ilnd - island id of mem w/queue desc
 * _ddr_loc  - locality of mem w/queue desc
 * _que_no   - que number, 0-1023
 *
 */
#endif

#define _dequeueBufRing(_ddr_ilnd, _ddr_loc, _que_no)
.begin
    .sig s1

```

```

.reg read $xref $xdebug[4]
.reg r_loc_ilnd
.xfer_order $xdebug

r_loc_ilnd = _ddr_ilnd | (_ddr_loc << 6)
r_loc_ilnd = r_loc_ilnd << 24

mem[dequeue, $xref, r_loc_ilnd, <<8, _que_no], ctx_swap[s1]

; Can use push_qdesc to debug the QArray contents.
mem[push_qdesc, $xdebug[0], r_loc_ilnd, <<8, _que_no], ctx_swap[s1]
nop

.end
#endif

; Main
.begin
.if(ctx() == 0)
.reg _ddr_addr, _ddr_loc, _ddr_ilnd
.reg _que_addr, _size, _page, _type, _loc _que_no

_ddr_ilnd = 0x18
_ddr_loc = 0x2
_ddr_addr = 0x0
_que_addr = 0x0
_que_no = 0x1
_page = 0x0
_size = 2 : 2K 32bit words
_loc = 0x0
_type = 0x02

; Initliaze the type 2 Ring Queue Descriptor in External memory
._queRingQDescInit( _ddr_addr, _ddr_loc, _ddr_ilnd, _que_no,\ 
                    _que_addr, _size, _page, _type, _loc)

; Read Queue descriptor into MU Queue Engine QArray
._queReadDesc_nfp(_que_no, _que_addr, _ddr_loc, _ddr_ilnd)

; Enqueue a buffer to the ring queue
._enqueueBufRing(_ddr_addr, _ddr_ilnd, _ddr_loc, _que_no)

; Dequeue a buffer from the ring queue
._dequeueBufRing(_ddr_ilnd, _ddr_loc, _que_no)

.endif
ctx_arb[kill]
.end

```

## 2.2.39.19 MEM (Rings Fast\_journal)

This command is only applicable to rings.

### Instruction Format

```
mem[cmd, --, src_op1, src_op2], opt Tok // 32-bit addressing
```

```
mem[cmd, --, src_op1, <<8, src_op2], opt_tok // 40-bit addressing
mem[cmd, --, src_op1, src_op2, << 8], opt_tok // 40-bit addressing
```

## Parameter Descriptions

Parameter	Command	Description
cmd	fast_journal	Write one 32-bit word of immediate data to a journal ring. All 32 bits of the journal data is specified by src_op1 + src_op2 and no pull operation is required on the pull bus.
	fast_journal_sig	Write one word to a journal ring that consists of 22 bits of the data specified by src_op1 + src_op2 together with 10 bits of tag data. The tag contains the island, signal master and signal ref. This is of limited use for debugging to know which context wrote the journal data.
--	fast_journal, fast_journal_sig	Must always be “--”.
src_op1, src_op2	fast_journal	32-bit data as specified by src_op1+src_op2 is journaled. The QueueArray is specified in the data_ref as detailed in Note 1 in <a href="#">Section 2.2.39.15</a> . The optional token must be indirect_ref.  Only one word is written and no signals are delivered.
	fast_journal_sig	Put the data along with tag onto the specified journal ring. The data has the following format: <ul style="list-style-type: none"> <li>• [31:28] = island_id[3:0] (tag)</li> <li>• [27:24] = signal_master[3:0] (tag)</li> <li>• [23:22] = signal_ref[6:5] (tag)</li> <li>• [21:0] = Journal data (from src_op1 + src_op2)</li> </ul> Only one word is written and no signals are delivered.
opt_tok	fast_journal, fast_journal_sig	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

## 2.2.39.20 MEM (Rings Queue Read and Write)

These commands allow the contents of a queue to be read or written at an offset from the base address. There are two types of read and writes – unbounded (read\_queue and write\_queue) and bounded (read\_queue\_ring and write\_queue\_ring). If an attempt is made to read or write beyond the end of the queue when using bounded operations, then it wraps to the beginning of queue treating it as a circular buffer.



### Note

MUConfigCPP[DirAccWays] has no effect on the Queue Engine Commands listed below. These commands use the QueueLoc field of the Queue Descriptor to determine whether memory access is to the External Memory or direct access (to DCache).

- read\_queue
- write\_queue
- read\_queue\_ring
- write\_queue\_ring



## Note

When reading the last 64 bit word in DRAM memory space, you must use an 64-bit-aligned read. DRAM reads that go beyond the end of physical memory, can cause cache corruption in later, unrelated, DRAM transactions.

## Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8, ref_cnt], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	read_queue	Unbounded read using queue engine at address:  (qdesc.tail & ~0x3FFFFFF)   (address & 0x3FFFFFF)
	read_queue_ring	Bounded read from queue at an offset from the base.
	write_queue	Unbounded write using queue engine at address:  (qdesc.tail & ~0x3FFFFFF)   (address & 0x3FFFFFF)
	write_queue_ring	Bounded write to queue at an offset from the base.
xfer	read_queue, read_queue_ring	Read transfer registers contain a ref_cnt number of 32-bit words of queue contents.
	write_queue, write_queue_ring	Write transfer registers contain a ref_cnt number of 32-bit words to be written into the queue.
src_op1, src_op2	read_queue, write_queue	Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>• [31:22] – Queue array entry number</li> <li>• [21:2] - address.</li> </ul>
	read_queue_ring, write_queue_ring	Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>• [31:22] – Queue array entry number</li> <li>• [21:2] - Offset from the queue base.</li> </ul>
ref_cnt	All commands	Reference count in increments of 4 byte words. Valid values are 1 to 16. Values above 8 must be specified using indirect reference.

Parameter	Command	Description
opt_tok	All commands	sig_done[sig_name] ctx_swap[sig_name] defer[n] where n = 1 or 2 indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2, ...]

## 2.2.39.21 MEM (STATS Operations)

The Stats Engine handles statistic processing for the Memory Unit. It allows logging of multiple statistics data to the local data cache in a single command, and can retrieve stats and push stats to requesting masters. Statistics data holding Packet count and Byte count is stored at same address in data cache (64 bit aligned). The statistics counters can be configured for wrapping or saturating.

Wrapping Statistics Format consists of Packet count in data[63:35] and Byte count in data[34:0]. Saturating Statistics Format consists of Packet count in data[62:35] and Byte count in data[33:0], where Packet Count Saturate Indicator is data[63] and Byte count Saturate Indicator is data[34]. Saturate Indicator bits remain set until statistic clear occurs. Statistics data is stored in the data cache on 64 bit addressable boundaries. To derive the data cache address from the the Stat command the Stat Engine bit wide ORs the write Transfer register value with MUSEBaseAddr[18:0] CSR. See NFP-6xxx Databook section MUSECsrXpb for how to select up to four MUSEBaseAddr registers (Base Address CSR for Statistic operations).

Stat commands issued to the Stats Engine are executed in the order they are received on a particular CPP bus, but across different CPP buses there is no order guaranteed.

### Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

### Parameter Descriptions

Parameter	Command	Description
cmd	stats_log	<p>Log statistics and wrap counters. Stats Engine retrieves statistics data stored in data cache, updates the packet and byte counters and writes it back to the data cache. Single stats_log command can support a maximum of sixteen statistic updates, where each statistic data is updated with the same byte count add value. Stats format is wrapping format where in data cache Packet Count is located in bits[63:35] and Byte Count is bits[34:0].</p> <p>Byte count add value added to statistic data is provided in command address[15:0]. The address where statistic data is stored in data cache is derived from Write Transfer registers.</p>

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
		How the pulled addresses are packeted into the pull data is determined by address[17:16] (AddrPackCfg). See <a href="#">Note 1</a> for more information.
	stats_log_event	Log statistics similar to stats_log and wrap counters. Raise event on half wrap or full wrap levels. Note that the continued counting will re-trigger half-wrap and full-wrap events when reached if the stat has not been cleared. Supports Packet and Byte Counter half-wrap (Event type = 3) and full-wrap (Event type = 2) thresholds.
	stats_log_sat	Log statistics similar to stats_log but with Saturate mode. Command will update stats and saturating stats will continue to update after the saturation point is reached and the saturate indicator will indicate the counter reached saturation point. Saturation indicator will remain set until command stats_push_clear is performed to the same address.  Stats format is Saturating format where in data cache Packet Count Saturate Indicator is bit[63], Packet Count is bits[62:35] and Byte Count Saturate Indicator is bit[34] and Byte Count is bits[33:0].
	stats_log_sat_event	Log statistics similar to stats_log_event with Saturate mode. Saturating stats will continue to update after the saturation point is reached and the saturate indicator will indicate the counter reached saturation point. Saturation indicator will remain set until command stats_push_clear to same address is issued. Stats format is Saturating format similar to stats_log_sat.
	stats_push	Read Stats data. Retrieve statistics from the data cache and return the values over the Push Bus to the requesting master.  The stats_push command provides the stat address in the command address[21:3] and address[31:30] indicates which MUSEBaseAddr[18:0] CSR to use when constructing the data cache address. The stat address format is unpacked. The stat data cache address construction is bit-wide OR of MUSEBaseAddr[18:0] and command address[21:3]. When command ref_cnt is N the supplied address is incremented and the statistics data for N sequential addresses are pushed to the master.
	stats_push_clear	Read Stats similar to stats_push, but also clear stats data in data cache on read. Command address parameters are similar to stats_push for deriving data cache address.
	stats_log, stats_log_event, stats_log_sat, stats_log_sat_event	Write transfer register holds the stat address that is bit ORed with MUSEBaseAddress[18:0] to derive the data cache address. Given the address packing configuration the packed address format can be 16 bits or 32 bits. See <a href="#">Note 1</a> for more information.
xfer	stats_push, stats_push_clear	Read transfer registers containing stats data.

Parameter	Command	Description
src_op1, src_op2	stats_log, stats_log_event, stats_log_sat, stats_log_sat_event	<p>Restricted source operands (src_op1+src_op2) combine to form preferred 40-bit address that encodes to meaningful command parameters. Command parameters: address[15:0] - Contains Byte Count value. address[17:16] - Defines how the stat addresses are packed in the pull data (Write Transfer Registers).</p> <p>Where address[17:16] options are: (2b0x) All addresses are 16-bit packed address, (2b10) All addresses are 32 bit unpacked addresses and (2b11) first address is 32-bit unpacked address and remaining addresses are 16-bit packed addresses. See <a href="#">Note 1</a></p>
	stats_push, stats_push_clear	<p>Restricted source operands (src_op1+src_op2) combine to form preferred 40-bit address that encodes to meaningful command parameters.</p> <p>Command parameters: address[18:0] - stat address, address[31:30] - Base Address Select one of four MUSEBaseAddr[18:0] CSRs.</p>
ref_cnt	stats_log, stats_log_event, stats_log_sat, stats_log_sat_event	<p>Reference count in 2-byte statistic address pointers to pull. Valid values are 1-16. Values above 8 must be specified using indirect reference. Refer to section <a href="#">Section 2.1.6.4.2</a></p> <p>Address pack configuration combined with the command length field determine the number of statistics data that the command will update. Length field specifies the number of 16 bit pulls associated with the command, but the way the addresses are packed affects the actual number of stats that will be updated by the command. See <a href="#">Note 1</a>.</p>
	stats_push, stats_push_clear	<p>Number of 8 byte stat data to read; valid values are 1-16. Use indirect_ref if ref_cnt is greater than 7. Refer to section <a href="#">Section 2.1.6.4.2</a></p>
opt Tok	All Commands	<p>sig_done[sig_name]</p> <p>ctx_swap[sig_name]</p> <p>defer[n] where n = 1 or 2</p> <p>indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a>.</p> <p>ind_targets[me1, me2,...]</p>

#### Notes:

1. Address pack configuration (specified by address[17:16]) combined with command length (ref\_cnt - 1) determines the number of statistics data that the command will update. The length field specifies the number of 16-bit pulls associated with the command, but the way the addresses are packed affects the actual number of stats that will be updated by the command.

The Stats Engine will round the length value up to an appropriate value to complete the command. However the source code must guarantee any and all pull data is valid in this scenario to ensure intended operation.

address[17:16] = 2b1x - Implies that the length of the associated command must be properly representative of the amount of data to pull. Further explanation is shown below.

- address[17:16] = 2b0x - All addresses are 16-bit packed address. Implies that the length field contains a value representing an even number of 16-bit pulls.
- address[17:16] = 2b10 - All addresses are 32-bit unpacked addresses. Implies that the length field contains a value greater than or equal to two 16-bit pulls.
- address[17:16] = 2b11 - First address is 32-bit unpacked address, remaining addresses are 16-bit packed addresses. Implies that the length field contains a value representing an even number of 16-bit pulls.

In example below Packed Address format for each statAdr[15:0] is:

- statAdr[15:14] - Base Address select, valid values are 0 to 3.
- statAdr[13:0] - Start address (64 bit aligned)

Packed Address format for each statAdr[31:0] is:

- statAdr[31:30] - Base Address select, valid values are 0 to 3.
- statAdr[18:0] - Start address (64 bit aligned)

```

Example (ref_cnt = 10, must use indirect_ref format):
[A] If command length = 9 // Indicating ten 16-bit pull elements
    address[17:16] = 2b0x // All addresses are packed
    Number of stats updated is 10
    Pull: Data:
        2      Data[63:0] = {                                     statdAdr9[15:0],statAdr8[15:0]}
        1      Data[63:0] = {statAdr7[15:0],statAdr6[15:0],statdAdr5[15:0],statAdr4[15:0]}
        0      Data[63:0] = {statAdr3[15:0],statAdr2[15:0],statdAdr1[15:0],statAdr0[15:0]}

[B] If command length = 9 // Indicating ten 16-bit pull elements
    address[17:16] = 2b10 // All addresses are unpacked
    Number of stats updated is 10
    Pull: Data:
        2      Data[63:0] = {                                     statAdr4[31:0]}
        1      Data[63:0] = {statAdr3[31:0],statAdr2[31:0]}
        0      Data[63:0] = {statAdr1[31:0],statAdr0[31:0]}

[C] If command length = 9 // Indicating ten 16-bit pull elements
    address[17:16] = 2b11 // First address is unpacked, remaining addresses packed
    Number of stats updated is 10
    Pull: Data:
        2      Data[63:0] = {                                     statdAdr8[15:0],statAdr7[15:0]}
        1      Data[63:0] = {statAdr6[15:0],statAdr5[15:0],statdAdr4[15:0],statAdr3[15:0]}
        0      Data[63:0] = {                                     statAdr2[15:0],statdAdr1[15:0],statAdr0[31:0]}

```

## 2.2.39.22 MEM (Ticket Release)

Release Ticket commands are used for reordering management, e.g. to reorder packets after processing. Using 128 bit memory data structure, these commands can help users to keep track of up to 96 packets. The first word of this structure contains a 10 bit number (or “ticket”) of the packet currently being processed. The remaining words contain a 96 bit map that marks tickets released prior to the current one.

## Instruction Format

mem[cmd, xfer, src_op1, src_op2], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <>8, src_op2], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <>8], opt Tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	release_ticket	If the released ticket number is greater than the current ticket, a bit equal to (released_ticket-current_ticket -1) is set in the bit map and value of 0 returned in the read transfer register. If the ticket number is equal to the current ticket, the current ticket number is set to the total number of already released tickets. The bit mask is updated and the number of released tickets is returned in the read transfer register. An error code of 0x00ff is returned in the read transfer register on attempt to release a ticket number smaller than the current ticket, or release a ticket number 96 more than the current ticket. Ticket numbers wrap around on the 10bit boundary (1022, 1023, 0, 1, ect...) See <a href="#">Note 1</a> .
	release_ticket_ind	This command is similar to release ticket but the returned data will be pushed to a different master. For an indirect release ticket command, the first word of the release ticket memory structure contains data master and data reference to direct data to a different master. See <a href="#">Note 2</a> .
xfer	release_ticket	A write transfer register contains a ticket number. A read transfer register contains a ticket release status information. See <a href="#">Note 1</a> .
	release_ticket_ind	A write transfer register contains a ticket number.
src_op1, src_op2	All commands	Restricted source operands that define a 32 or 40-bit byte address of 128-bit ticket release data structure.
opt Tok	release ticket	sig_done[sig_name]. (See <a href="#">Note 3</a> )
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]
	release_ticket_ind	sig_done[sig_name]
		ctx_swap[sig_name]
		defer[n] where n = 1 or 2
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]

### Notes:

1. 128-bit release ticket memory structure for release\_ticket command:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															Current ticket number																

<b>31</b>	<b>30</b>	<b>29</b>	<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>	<b>20</b>	<b>19</b>	<b>18</b>	<b>17</b>	<b>16</b>	<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
Bit map																															
Bit map																															
Bit map																															

### Example

Current ticket number is 13. Previously released tickets – 14 and 19.

<b>31</b>	<b>30</b>	<b>29</b>	<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>	<b>20</b>	<b>19</b>	<b>18</b>	<b>17</b>	<b>16</b>	<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
Reserved																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

After the release\_ticket command requested release of ticket number 15, then bit 15 in the bit map will be set as shown below and the read transfer register is set to zero.

<b>31</b>	<b>30</b>	<b>29</b>	<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>	<b>20</b>	<b>19</b>	<b>18</b>	<b>17</b>	<b>16</b>	<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
Reserved																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

After the release\_ticket command requested the release of the current ticket number 13, the number of contiguous tickets already released is calculated. In this example, this number is equal to 2 so the current ticket number is updated by this amount plus 1: 13 + 2 + 1 = 16. The bit map is shifted down by this amount. A read transfer register is set to 2 (number of released tickets) in the lower 8 bits and the upper 24 bits are set to the lower 24 bits of the address of the ticket release data structure. Thereafter the release ticket structure will look as follows:

<b>31</b>	<b>30</b>	<b>29</b>	<b>28</b>	<b>27</b>	<b>26</b>	<b>25</b>	<b>24</b>	<b>23</b>	<b>22</b>	<b>21</b>	<b>20</b>	<b>19</b>	<b>18</b>	<b>17</b>	<b>16</b>	<b>15</b>	<b>14</b>	<b>13</b>	<b>12</b>	<b>11</b>	<b>10</b>	<b>9</b>	<b>8</b>	<b>7</b>	<b>6</b>	<b>5</b>	<b>4</b>	<b>3</b>	<b>2</b>	<b>1</b>	<b>0</b>
Reserved																															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

When the release ticket command requested a ticket number < 16, then the lower 8 bits of a read transfer register is set to 0xff and the upper 24 bits are set to the lower 24 bits of the address of the ticket release data structure.

2. 128-bit release ticket memory structure for release\_ticket\_ind command:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																							
Data destination	Data master FPC	Data reference								Current ticket number																																												
Bit map																																																						
Bit map																																																						
Bit map																																																						

3. Two signals are delivered for release\_ticket command – one on pull and another on push.

### 2.2.39.23 MEM (Work Queue Operations)

These instructions enable work to be delivered to a pool of threads.

Work is added to the work queue with the qadd\_work and qadd\_work\_imm operations. These operations however do not check for ring overflow, and overflow will occur when more work is added than is consumed.

Threads (consumers) are added to the work queue with the qadd\_thread operation and a signal is delivered when work is available or becomes available.

These instructions operate on rings configured as type 2.

Work may consist of 1 to 16 32-bit words and should be the same size when added as when consumed by qadd\_thread. The producer and consumer of the work queue data must specify the same work queue data size.



#### Note

MUConfigCPP[DirAccWays] has no effect on the Queue Engine Commands listed below. These commands use the QueueLoc field of the Queue Descriptor to determine whether memory access is to the External Memory or direct access (to DCache).

- qadd\_work
- qadd\_work\_imm
- qadd\_thread

#### Instruction Format

mem[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok // 32-bit addressing
mem[cmd, xfer, src_op1, <<8, src_op2, ref_cnt], opt Tok // 40-bit addressing
mem[cmd, xfer, src_op1, src_op2, <<8, ref_cnt], opt Tok // 40-bit addressing

#### Parameter Descriptions

Parameter	Command	Description
cmd	qadd_work	Add work specified in transfer register(s) and place on the work queue. No work queue overflow checking is performed.

Parameter	Command	Description
	qadd_work_imm	Add a single long word describing work on the work queue. No work queue overflow checking is performed and no signals are used.
	qadd_thread	Fetch work from a work queue into transfer register(s). This call will block until there is work on the queue.
xfer	qadd_work	Write transfer registers containing the work to add.
	qadd_work_imm	Must be omitted; should be "--".
	qadd_thread	Read transfer registers containing the work to perform.
src_op1, src_op2	qadd_work, qadd_thread	Restricted source operands for address. The address is specified by src_op1 + src_op2. Restricted operands are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"><li>• [9:0] - Queue array entry number.</li></ul>
	qadd_work_imm	src_op1 + src_op2 = work to add
ref_cnt	qadd_work, qadd_thread	Number of words to put on ring. Valid values are 1-16. Values above 8 must be specified using indirect reference.
opt_tok	qadd_work, qadd_thread	sig_done[sig_name]
		ctx_swap[sig_name]
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
	qadd_work_imm	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . (See <a href="#">Note 1</a> )

#### Notes:

1. A queue array entry number has to be specified in the data reference field of indirect format. An example of using a queue array number 0x11 is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Encoding		Reserved								Queue array entry number								Reserved																
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0

## 2.2.40 SRAM

Each Internal and External Memory Unit in the NFP Architecture supports a VQDR target. This target supports the same set of commands as the QDR unit in the NFP-6xxx predecessor and is provided for compatibility. Thus below SRAM instructions are implemented as virtual QDR instructions in the memory unit (=DRAM).

## 2.2.40.1 SRAM (Atomic Operations)



### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Issue a memory reference to perform an atomic operation on data in VQDR memory. An atomic operation is one in which a read, modify, and write operation is performed on the memory location and it is assured that another atomic operation will not be allowed to access the data while the atomic operation is in progress. Non-atomic operations, such as a sram[read], can occur in the middle of a atomic operation.

The NFP-6xxx supports no\_pull atomic operations for the swap, set, clr, add, test\_and\_set, test\_and\_clr, and test\_and\_add commands. The no\_pull version is specified by using the no\_pull and the indirect\_ref tokens. The indirect reference also provides the write data, which is normally “pulled” from the write transfer register. Also, the no\_pull version of the instructions use one less signal.

By not using a transfer register for the data modifier, the MEM unit does not have to pull any data from the FPC, which in turn speeds up the execution of the no\_pull atomic operations in the MEM unit.

### Instruction Format

```
sram[cmd, xfer, src_op1, src_op2], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	swap	Swap the contents of a transfer register with the data at the address. See <a href="#">Note 2</a> .
	set	Set the bit(s) at the specified address according to a bit mask provided in the transfer register. A one in the bit position of the bit mask signifies that the bit should be set.
	clr	Clear the bit(s) at the specified address according to a bit mask provided in the transfer register. A one in the bit position of the bit mask signifies that the bit should be cleared.
	incr	Increment the value of the data at a specified address by 1. The value in memory rolls over to 0 after 0xFFFF FFFF.
	decr	Decrement the value of the data at the specified address by 1. The value in memory saturates at 0x0000 0000.
	add	Add the 2s complement value in the transfer register to the data at the address, which is treated as an unsigned number. Negative number addition whose results is < 0 will saturate the result in the value in memory at 0x0000 0000. Positive number addition whose results is > 0xFFFF FFFF will roll over.
	test_and_set	Same as the set instruction, but also return the premodified value to the transfer register that contained the bit mask. See <a href="#">Note 2</a> .

Parameter	Command	Description
	test_and_clr	Same as the clr instruction, but also return the premodified value to the transfer register that contained the bit mask. See <a href="#">Note 2</a> .
	test_and_incr	Same as the incr instruction, but also return the premodified value to the transfer register that contained the incr value. The premodified value can be used to test for saturation.
	test_and_decr	Same as the decr instruction, but also return the premodified value to the transfer register that contained the decr value. The premodified value can be used to test for saturation.
	test_and_add	Same as the add instruction, but also return the premodified value to the read transfer register specified. The premodified value can be used to test for saturation. See <a href="#">Note 2</a> .
no_pull cmd (See <a href="#">Note 1</a> )	swap	Replace the data at the address with the sign extended contents of the data field from the no_pull indirect reference (See <a href="#">Note 1</a> ). The original data at the address is placed in the read transfer register.
	set	Set one bit at the specified address. The bit to set is in the data field [4:0] specified in the no_pull indirect reference. (See <a href="#">Note 1</a> ). Data field [10:5] is ignored.
	clr	Clear one bit at the specified address. The bit to clear is in the data field [4:0] specified in the no_pull indirect reference (See <a href="#">Note 1</a> ). Data field [10:5] is ignored.
	add	Add the sign extended value in the data field of the no_pull indirect reference to the data at the address. (See <a href="#">Note 1</a> ) Saturates at 0x00000000 if the data field is negative number and the addition results is <0. If the data field is a positive number, then addition results > 0xFFFFFFFF will roll over.
	test_and_set	Same as the set instruction, but also return the premodified value to a read transfer register.
	test_and_clr	Same as the clr instruction, but also return the premodified value to a read transfer register.
	test_and_add	Same as the add instruction, but also return the premodified value to the read transfer register.
xfer	incr, decr	Not used and should be “--”.
	test_and_incr, test_and_decr	Transfer read register that holds the premodified data.
	swap & all other test commands	Transfer write register that holds the data modifier and the transfer read register that hold the premodified data.
	set, clr, add	An transfer write register that holds the data modifier.
	set, clr, add, with no_pull opt_tok	Not used and should be “--”.

Parameter	Command	Description
	swap, test_and_set, test_and_clr, test_and_add with no_pull opt_tok	An transfer read register to hold the premodified data.
src_op1, src_op2	All commands	<p>Restricted operands that define the byte address. The address is specified by src_op1 + src_op2. Bits[1:0] of the address is ignored by the SRAM channels. The base addresses of the SRAM channels are:</p> <ul style="list-style-type: none"> <li>• Channel 0: 0x0000 0000</li> <li>• Channel 1: 0x4000 0000</li> <li>• Channel 2: 0x8000 0000</li> <li>• Channel 3: 0xc000 0000</li> </ul>
opt_tok	test_and_incr, test_and_decr set, clr, add	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .
	test_and_set, test_and_clr, test_and_add, swap	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .
	incr, decr	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .
set, clr, add, with no_pull opt_tok	no_pull (required)	
	indirect_ref (required); refer to <a href="#">Section 2.1.6.4.2</a> .	
	ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .	
	swap, test_and_set, test_and_clr, test_and_add, with no_pull opt_tok	no_pull (required)
		sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
indirect_ref (required); refer to <a href="#">Section 2.1.6.4.2</a> .		
ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .		
ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .		

#### Notes:

1. NFP-6xxx supports a “no\_pull” version of these commands. The no\_pull version is specified by the indirect reference token no\_pull. The indirect reference also provides the write data, which is normally “pulled” from the write transfer register. The length field has to have bit[3] set, and then the byte mask contains bits [7:0] of the 11-bit immediate data, and length bits [2:0] contain the top 3 bits, which are sign-extended. The no\_pull commands do not require a write transfer register.

Refer to [Section 2.1.6.4.6](#) and [Section 2.1.6.4.5](#) for Indirect Reference usage.

- Two signals are required and this affects how the BR\_SIGNAL, BR\_!SIGNAL instructions are used. Refer to the BR\_SIGNAL, BR\_!SIGNAL instructions for details.

## 2.2.40.2 SRAM (Dequeue)



### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

The Memory Unit ConfigVQDRn[QueueType] register bits support the following three modes that determine the behavior of the dequeue command (Mode 2 is not supported). Refer to [Figure 2.3](#) for examples of these modes.

### Mode 0: Dequeue Segments & Count Packets

The seg\_cnt is decremented for each sram[dequeue] command. Only when seg\_cnt equals 0 is the q\_link removed from the linked list. The EOP is used by hardware to determine if it should decrement the q\_count, therefore it must be set on the last buffer of a packet for software designs that support multiple buffers per packet or on all buffers for software designs that support a single buffer per packet. The state of the EOP bit is returned with the data for each dequeue command. The state of the SOP bit is returned only with the data for the first dequeue command to a buffer. The SOP bit is clear for subsequent dequeue commands to the buffer.

### Mode 1: Dequeue Buffers & Count Packets

The seg\_cnt is ignored in this mode so a q\_link is removed from the linked list for each sram[dequeue] command. The EOP and SOP bits are treated the same as mode 0. The seg\_cnt is returned unchanged on each dequeue.

### Mode 3: Dequeue Buffers & Count Buffers

The seg\_cnt is ignored in this mode so a q\_link is removed from the linked list for each sram[dequeue] command. The EOP bit is also ignored by hardware so the q\_count is always decremented for each dequeue command. Note: In this mode the seg\_cnt is added to the q\_count on every enqueue. A seg\_cnt value of 0 is illegal for enqueue commands.

### Instruction Format

```
sram[cmd, xfer, src_op1, src_op2], opt_tok
```

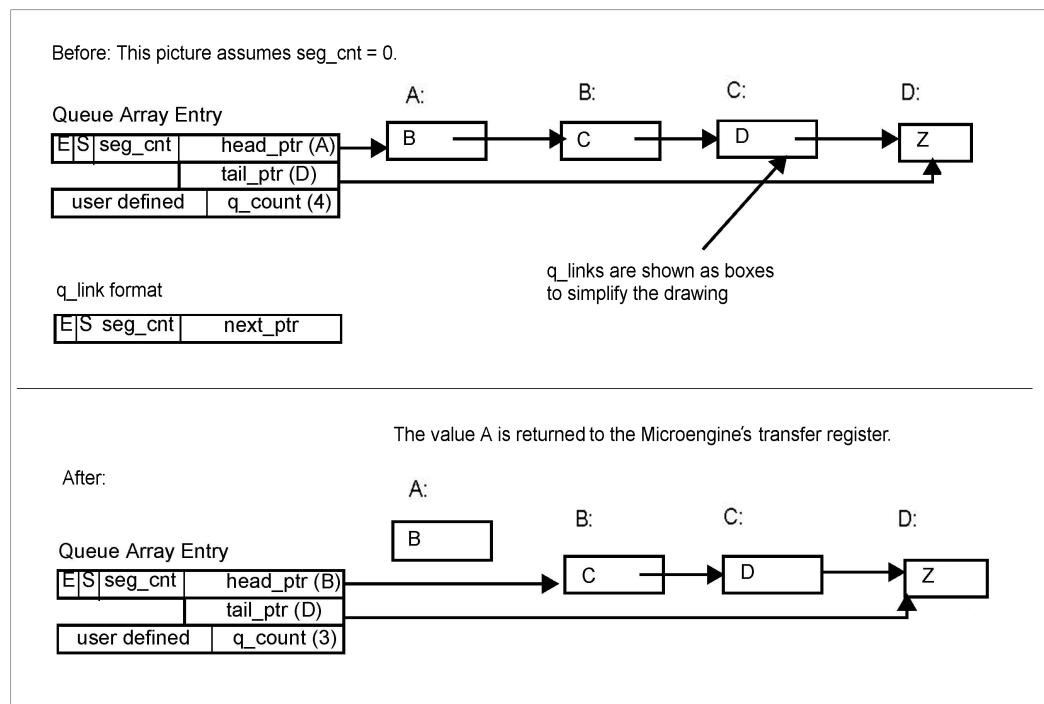
### Parameter Descriptions

Parameter	Command	Description
cmd	dequeue	This command is used to remove a q_link from the queue. If the q_count is 0 (nothing on the queue), the value of 0 is returned. If the q_count is not 0 (something on the queue), and the seg_cnt is 0 in the head q_link (buffer only has one segment), then the q_link is removed from the linked list and it is returned. If the q_count is

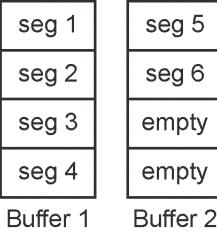
Parameter	Command	Description
		not 0 (something on the queue), and the seg_cnt is NOT 0 in the head q_link (buffer holds multiple segments), then the seg_cnt is decremented and the q_link is NOT removed from the linked list. If the q_count is not 0 then the entire q_link is returned.
xfer	dequeue	Transfer read register where the q_link is returned.
src_op1, src_op2	dequeue	Restricted operands that are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>[31:30] SRAM Channel</li> <li>[29:24] Queue Array Entry Number</li> <li>[23:0] ignored.</li> </ul>
opt Tok	dequeue	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> . ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> . indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . defer[n]; refer to <a href="#">Section 2.1.5</a> . ind_targets[fpc1, fpc2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

### Condition Codes Affected

N	Z	V	C
Not Affected			



**Figure 2.2. Dequeue Buffer**

<p>Mode 0: Dequeue Segments &amp; Count Packets</p>  <p>Buffer 1      Buffer 2</p>	<p>Q_link 1: seg_cnt = 4, decremented each dequeue SOP = 1, cleared on first dequeue EOP = 0, q_count not decremented Q_link is removed when seg_cnt = 0 (4 dequeues)</p> <p>Q_link 2: seg_cnt = 2, decremented each dequeue SOP = 0 EOP = 1, q_count decremented when seg_cnt = 0 Q_link is removed when seg_cnt = 0 (2 dequeues)</p> <p>q_count is incremented each enqueue (not enqueue_tail)</p>
<p>Mode 1: Dequeue Buffers &amp; Count Packets</p> <p>Example: One Packet in 2 Buffers</p>  <p>Buffer 1      Buffer 2</p>	<p>Q_link 1: seg_cnt is ignored SOP = 1 EOP = 0, q_count not decremented</p> <p>Q_link 2: seg_cnt is ignored SOP = 0 EOP = 1, q_count decremented</p> <p>Q_link is removed on each dequeue</p> <p>q_count is incremented each enqueue (not enqueue_tail)</p>
<p>Mode 3: Dequeue Buffers &amp; Count Buffers</p> <p>Example: One Packet in 2 Buffers</p>  <p>Buffer 1      Buffer 2</p>	<p>Q_link 1: seg_cnt = 2. This is added to the current q_count during the enqueue (not enqueue_tail) to indicate that two buffers are being enqueued. It is ignored on dequeue. SOP = 1 EOP = ignored, q_count not decremented</p> <p>Q_link 2: seg_cnt is ignored SOP = 0 EOP = ignored, q_count decremented</p> <p>Q_link is removed on each dequeue</p>

**Figure 2.3. Example of the Three Dequeue Modes**

### 2.2.40.3 SRAM (Enqueue)



#### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Two commands are provided for performing enqueue operations. The enqueue command is used to enqueue to a queue structure that supports a single buffer per packet (refer to [Figure 2.4](#) ). The enqueue\_tail (enqueue tail) command is used with the enqueue command for queue structures that support multiple buffers per packet (refer to [Figure 2.4](#) and [Figure 2.5](#) ). In this case the enqueue command links the first buffer to the queue array and the

`enq_tail` command updates the tail pointer in the SRAM Queue Array. The default is to set the segment count (`seg_cnt`) to 0 and set the SOP and EOP bits. This can be overridden using the indirect reference optional token.

This instruction uses `src_op1` and `src_op2` parameters to select the SRAM channel and SRAM Queue Array entry as well as to pass the address of the buffer descriptor to the SRAM Queue Array. The indirect reference is used to set the EOP, SOP, and `seg_cnt` fields. The indirect reference must override the byte mask using the DATA16 field, with the following format for the DATA16 field:

8h00 : EOP(1) : SOP(1) : seg\_cnt(6)

Note the following:

1. When performing SRAM Enqueues from the ARM processor, the EOP, SOP, and Segment Count fields are always written with values EOP = 1, SOP = 1, and Segment Count = 0.
2. The ARM processor cannot perform enqueue operations if the queue controller is in Mode 3 since a Segment Count of 0 on enqueue is illegal in this mode. Refer to [Section 2.2.40.2](#) for a description of Mode 3.

## Instruction Format

```
sram[cmd, --, src_op1, src_op2], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	enqueue	This command is used to add a single buffer to the queue or to add the Start-of-Packet buffer of a multi-buffer frame to the queue. It adds a buffer to the queue contained in the Q-array entry, and sets the tail to point to the buffer. If necessary, a link is established from the old tail buffer to the new buffer. If Memory Unit ConfigVQDRn[QueueType] = 3, the value of <code>seg_cnt</code> is added to <code>q_count</code> . Otherwise the <code>q_count</code> is incremented.
	enqueue_tail	This command updates the tail pointer only. This command must be proceeded by an enqueue command to the same entry. This adds the End-of-Packet buffer of a multi-buffer frame to the queue. There must be no intervening commands to a queue array entry between an enqueue and enqueue_tail command.
--	Both commands	Must always be “--”.
src_op1, src_op2	Both commands	Restricted operands that are added ( <code>src_op1 + src_op2</code> ) to define the following: <ul style="list-style-type: none"> <li>• [31:30] SRAM Channel</li> <li>• [29:24] Queue Array Entry Number</li> <li>• [23:0] Q_link Pointer.</li> </ul> The <code>Q_link</code> Pointer must be a 4-byte word address.
opt Tok	Both commands	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

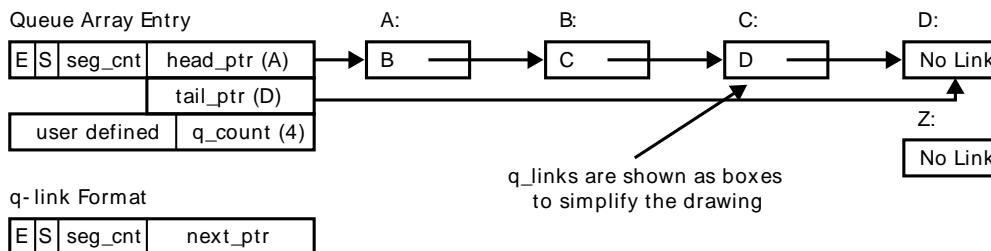
### Notes:

1. When an indirect ref is not specified, the parameters delivered to the SRAM controller are SOP =1, EOP =1, and SEG\_CNT = 0x3F (which is translated by the SRAM controller as a SEG\_CNT of 0).
2. If in Queue Array HW is in Mode 1, the segment count field is not used by hardware and can be used by software as a message field between the enqueuer and dequeuer.
3. If in Queue Array HW is in Mode 3, a segment count of 0 is illegal and is considered a programming error if used.

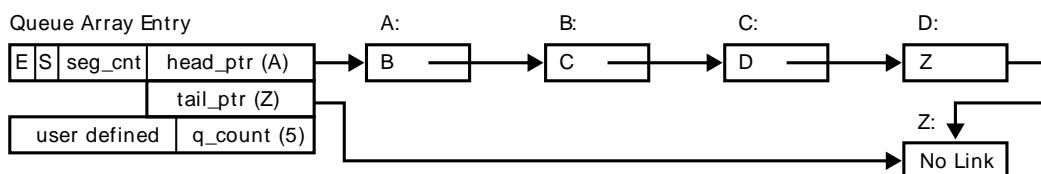
### Condition Codes Affected

N	Z	V	C
Not Affected			

Before

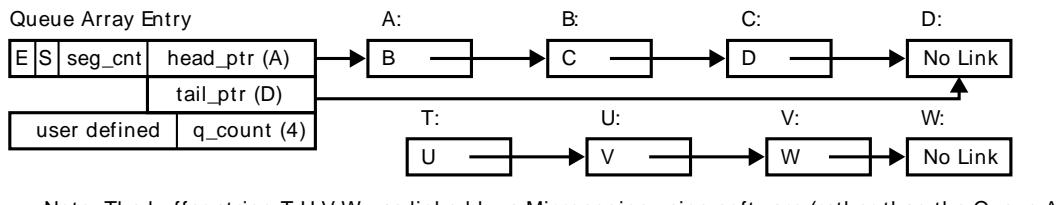


After



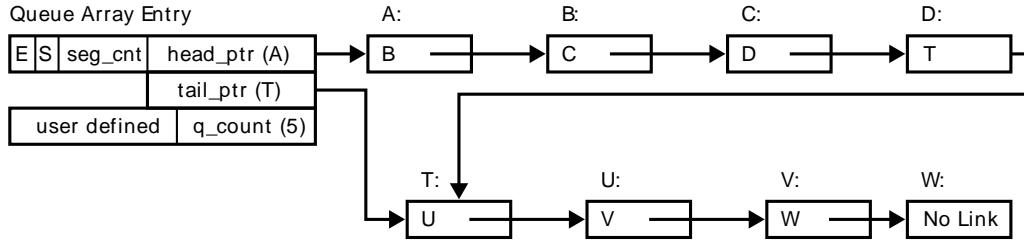
**Figure 2.4. Enqueue One Buffer at a Time Using the Enqueue Command**

Initial State of the Queue

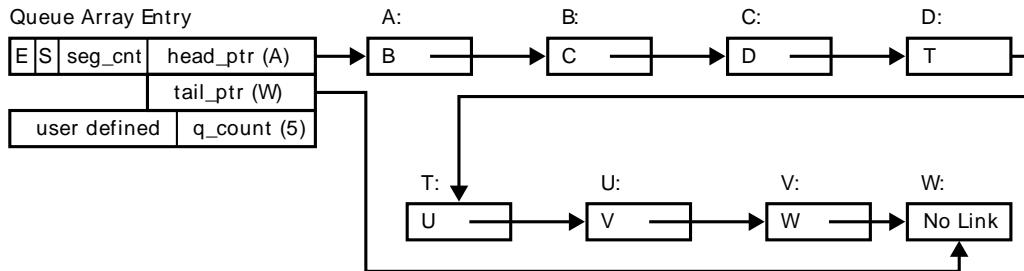


Note: The buffer string T,U,V,W was linked by a Microengine using software (rather than the Queue A)

First Step to Enqueue a String of Buffers to a Queue (sram[enqueue])



SECOND Step to Enqueue a String of Buffers to a Queue (sram[enqueue\_tail])



**Figure 2.5. Enqueue a String of Buffers to a Queue**

#### 2.2.40.4 SRAM (Journal Operations)



##### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Issue an SRAM Journal command to the SRAM channel. A Journal is similar to a Ring except that only the put operation is supported and the data written to the journal ring will wrap to the beginning when the Journal Ring fills. Data is read from the Ring using a standard sram read instruction. Each Journal Ring uses one of the 64 SRAM Queue Array Entries in the SRAM controller. Note that the maximum number of Journal rings is equal to 64 times the number of SRAM channels supported by the Network Flow Processor. Also, note that the journal descriptor must be read from SRAM using the Read Queue Descriptor command before issuing any journal operations to it. The format of the journal descriptor is similar to that of the ring descriptor shown in [Table 2.43](#).

The journal command puts data onto the Journal Ring from the transfer registers. The number of 4- byte words put onto the Journal Ring is specified by the ref\_cnt field. The fast\_journal command puts immediate data onto

the Journal Ring. The fast\_journal command always writes a tag that tells the SRAM Channel to insert the Cluster, FPC and thread numbers into bits [31: 24] of the data. (FPC Cluster bit [31], FPC number bits [30: 27] and thread number bits [26: 24]).

The journal area will always start at an address aligned to the journal size (for example, if the size is 1K the journal area will start on a 1K address boundary).

The sram[rd\_qdesc] commands must be used to initialize the queue array entry before the Journal can be used.

An example use for the journal is for debug:

- The FPC threads write to the journal area at specific points in the code. Examples of information written to the journal area might be packet header/timestamp/buffer pointer, tagged with FPC name.
- A FPC hits a breakpoint based on an error and all the FPCs are stopped. For example, when it receives an illegal buffer\_pointer.
- The breakpoint routine results in all the FPCs to be stopped.
- The user reads the journal area to help unravel what went wrong.

## Instruction Format

```
sram[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	journal	Put the data in the specified transfer registers onto the specified journal ring.
	fast_journal	Put the data along with a tag onto the specified journal ring. The data has the following format: <ul style="list-style-type: none"> <li>• [31:27] = FPC number (tag)</li> <li>• [26:24] = Context number (tag)</li> <li>• [23:0] = Journal data.</li> </ul> The FPC and context number is that which wrote the data or if the indirect option is used it is the FPC and context specified by the fields in the indirect reference. Journal data is specified by src_op1 + src_op2. Note that the journal data is provided with the command and no pull operation is required on the pull bus.
xfer	journal	Transfer write register specifies the first of a contiguous set of registers that hold the data that is to be put onto the journal ring.
	fast_journal	Not required and should be “--”
src_op1, src_op2	journal	Restricted operands that are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>• [31:30] = SRAM Channel</li> </ul>

Parameter	Command	Description
		<ul style="list-style-type: none"> <li>• [29:24] = Queue Array Entry (journal number)</li> <li>• [23:0] = Not used.</li> </ul>
	fast_journal	Restricted operands that are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>• [31:30] = SRAM Channel</li> <li>• [29:24] = Queue Array Entry (journal number)</li> <li>• [23:0] = Journal data.</li> </ul>
ref_cnt	journal	Reference count. Specifies the number of transfers (1 to 8) in increments of 4 byte words. Values above 8 must be specified using indirect reference. If the indirect_ref token is specified, ref_cnt can be a keyword max_nn (where nn = 1-16). Refer to <a href="#">Section 2.1.6.4.3</a> .
	fast_journal	Not required and should be omitted from the parameter list.
opt_tok	journal	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
	fast_journal	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

## 2.2.40.5 SRAM (Read and Write)



### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Move data between the FPCs and SRAM(VQDR) memory.

#### Instruction Format

```
sram[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

#### Parameter Descriptions

Parameter	Command	Description
cmd	read	Read sram memory starting at the specified address into the specified transfer registers.
	write	Write sram memory starting at the specified address from the specified transfer registers.

Parameter	Command	Description
xfer	read	Transfer read register.
	write	Transfer write register.
src_op1, src_op2	Both commands	Restricted operands that define the byte address. The address is specified by src_op1 + src_op2. Bits[1:0] of the address is ignored by the SRAM channels. The base addresses of the SRAM channels are: <ul style="list-style-type: none"> <li>• Channel 0: 0x0000 0000</li> <li>• Channel 1: 0x4000 0000</li> <li>• Channel 2: 0x8000 0000</li> <li>• Channel 3: 0xc000 0000</li> </ul>
ref_cnt	Both commands	Reference count in increments of 4 byte words. Valid values are 1 to 8.
opt Tok	read	ignore_data_error
	Both commands	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.5</a> .
		ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .

#### Condition Codes Affected

N	Z	V	C
Not Affected			

### 2.2.40.6 SRAM (Read Queue Descriptor)



#### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Issue a memory reference to an SRAM Channel to read the Queue descriptor into the SRAM Queue Array (see [Figure 2.6](#)). Three commands are provided to read the head and q\_count, the tail and q\_count, or the other (which reads the head if it is not currently in the array or the tail if it is currently not in the array). Optionally, additional data (as defined by the application) that begins at the specified address + 3 can be read from the SRAM into the FPC transfer registers.

#### Instruction Format

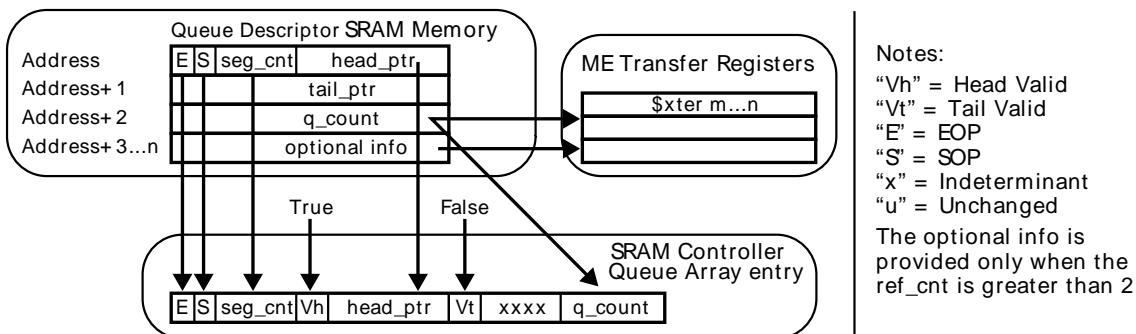
```
sram[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

## Parameter Descriptions

Parameter	Command	Description
cmd	rd_qdesc_head	This command is used to partially load a new queue_descriptor into the queue_array at an assigned entry number. Dequeues can be performed once the head is loaded. Follow rd_qdesc_head with rd_qdesc_other to completely load the queue_array entry. Refer to <a href="#">Figure 2.6</a> for a description of fields moved between the queue descriptor, SRAM Queue Array and FPC.
	rd_qdesc_tail	This command is used to partially load a new queue_descriptor into the queue_array at an assigned entry number. Enqueues can be performed once the tail is loaded. Follow rd_qdesc_tail with rd_qdesc_other to completely load the queue_array entry. Refer to <a href="#">Figure 2.6</a> for a description of fields moved between the queue descriptor, SRAM Queue Array and FPC.
	rd_qdesc_other	This command is used to finish loading a new queue_descriptor into the queue_array at an assigned entry number. This command should only be used when either the head or tail is valid in the same queue_array entry. Refer to <a href="#">Figure 2.6</a> for a description of fields moved between the queue descriptor, SRAM Queue Array and FPC.
xfer	rd_qdesc_head	Transfer read register where the q_count and optional data is returned.
	rd_qdesc_other	Must be --
src_op1,sr c_op2	All commands	<p>Restricted operands that are added (src_op1 + src_op2) to define the following:</p> <ul style="list-style-type: none"> <li>• [31:30] SRAM Channel</li> <li>• [29:24] Queue Array Entry Number</li> <li>• [23:0] Queue Descriptor data block.</li> </ul> <p>The Address of Queue Descriptor block specifies a 4-byte word address (not a byte address) that must be the start of the 16-byte aligned address of the Queue descriptor (i.e bits [1:0] should always be 0). The queue descriptor consists of the head, tail, q_count and optional data.</p>
ref_cnt	rd_qdesc_head	Reference count in increments of 4 byte words. Valid values are 2 to 8. The number of words returned to transfer registers is one less than the ref_cnt specified. For example, if the ref_cnt were 3, then three words would be read from sram, the first would be used by the SRAM Queue Array hardware, and the last two (q_count and optional word) would be returned to transfer registers.
	rd_qdesc_other	Not required and must be omitted from the parameter list.
opt Tok	All commands	<p>indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .</p> <p>ind_targets[me1, me2, ...]; refer to <a href="#">Section 2.1.6.4.4</a> .</p>

Parameter	Command	Description
	rd_qdesc_head	sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
	rd_qdesc_tail	ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		defer[n] (n = 1 to 2); refer to <a href="#">Section 2.1.6.4.2</a> .
		ignore_data_error

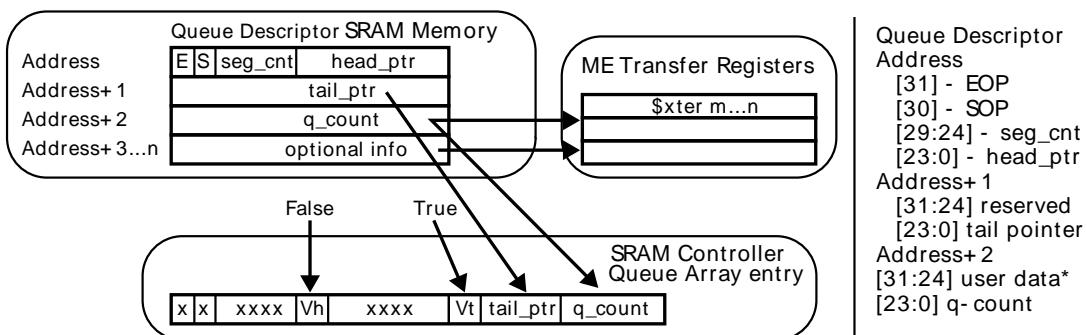
#### Head Command



#### Notes:

"Vh" = Head Valid  
 "Vt" = Tail Valid  
 "E" = EOP  
 "S" = SOP  
 "x" = Indeterminant  
 "u" = Unchanged  
 The optional info is provided only when the ref\_cnt is greater than 2

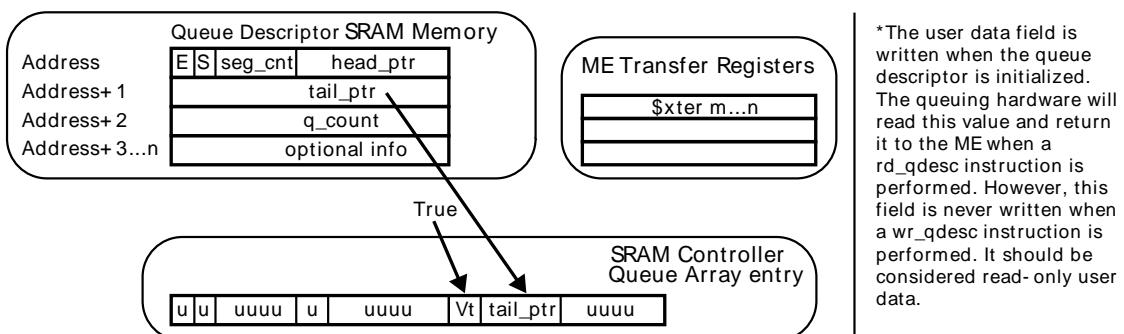
#### Tail Command



#### Queue Descriptor

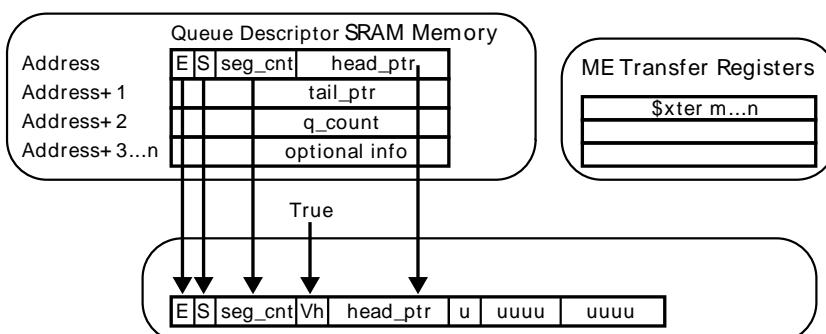
Address	[31] - EOP
	[30] - SOP
	[29:24] - seg_cnt
	[23:0] - head_ptr
Address+1	[31:24] reserved
	[23:0] tail pointer
Address+2	[31:24] user data*
	[23:0] q-count

#### Other Command with Head Valid



\*The user data field is written when the queue descriptor is initialized. The queuing hardware will read this value and return it to the ME when a rd\_qdesc instruction is performed. However, this field is never written when a wr\_qdesc instruction is performed. It should be considered read-only user data.

#### Other Command with Tail Valid



**Figure 2.6. Read Queue Descriptor Commands**

## Condition Codes Affected

N	Z	V	C
Not Affected			

## 2.2.40.7 SRAM (Ring Operations)



### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Issue an SRAM Ring put or get command to the SRAM. Each Ring uses one of the 64 SRAM Queue Array Entries in the SRAM controller. Note that the maximum number of SRAM rings is equal to 64 x the number of SRAM channels supported by the Network Processor. Also, note that the ring descriptor must be read from SRAM using the Read Queue Descriptor command before issuing any put or get operations to it. The format of the ring descriptor is shown in [Table 2.43](#).

**Table 2.43. SRAM Ring Descriptor Format**

Name	Longword #	Bit #	Definition
head_ptr	0	23:0	Head pointer.
tail_ptr	1	23:0	Tail pointer.
Ring Count	2	23:0	Number of longwords on the ring.

**Note:** For Ring/Journal, Head and Tail must be initialized to the same address.

Journals and Rings can be configured to be one of eight sizes, as shown in [Table 2.44](#).

**Table 2.44. SRAM Ring Size Encoding**

Ring Size Encoding	Size of Journal/ Ring Area	Head/Tail Field Base	Head and Tail Field Increment
000	512 Longwords	23:9	8:0
001	1K	23:10	9:0
010	2K	23:11	10:0
011	4K	23:12	11:0
100	8K	23:13	12:0
101	16K	23:14	13:0
110	32K	23:15	14:0
111	64K	23:16	15:0

The put command puts data onto the Ring while the Get command gets data from the Ring. The number of 4-byte words moved between the FPC and the Ring is specified by the ref\_cnt field. For the Put command, a 4-

byte Status word is returned that indicates the current number of 4- byte words on the Ring and whether the put was successful. The put command will be unsuccessful if there are 64 or fewer free entries. The Status word is written to read transfer register specified by xfer field. If the Put operation is unsuccessful, the data is dropped (not put onto the ring) and the FPC should retry the put command. A value of zero will be returned if the ring is empty.

The ring area will always start at an address aligned to the ring size (example if the size is 1K the ring area will start on a 1K address boundary).

The sram[rd\_qdesc] commands must be used to initialize the queue array entry before the Ring can be used.

## Instruction Format

```
sram[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

### Parameter Descriptions

Parameter	Command	Description
cmd	get	Get the data from the ring specified in the address and return it to the specified transfer registers.
	put	Put the data into the ring specified in the address from the specified transfer registers.
xfer	get	Transfer read register.
	put	The write transfer registers holds the data that is put onto the ring, the read transfer register holds the Full Status word which has the format: <ul style="list-style-type: none"> <li>• [31] = Success = 1, Fail = 0</li> <li>• [30:16] = Always 0</li> <li>• [15:0] = Number of 4-byte words on the Ring before the PUT operation.</li> </ul>
src_op1, src_op2	Both commands	Restricted operands that are added (src_op1 + src_op2) to define the following: <ul style="list-style-type: none"> <li>• [31:30] = SRAM Channel</li> <li>• [29:8] = Ignored</li> <li>• [7:2] = Ring Number</li> <li>• [1:0] = Ignored.</li> </ul>
ref_cnt	Both commands	Reference count. Specifies the number of transfers (1 to 8) in increments of 4 byte words. Values above 8 must be specified using indirect reference. If the indirect_ref token is specified, ref_cnt can be a keyword max_nn (where nn = 1-16). Refer to <a href="#">Section 2.1.6.4.3</a> .
opt_tok	put	sig_done[sig_name2]; refer to <a href="#">Section 2.1.6.5</a> .

Parameter	Command	Description
	get	indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .
		sig_done[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		ctx_swap[sig_name]; refer to <a href="#">Section 2.1.6.5</a> .
		defer[n]; refer to <a href="#">Section 2.1.5</a> .
		indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> .

## 2.2.40.8 SRAM (Write Queue Descriptor)



### Note

All SRAM/VQDR instructions are deprecated in NFP-6xxx and programmers should not use them. SRAM/VQDR instructions are ONLY available for porting old code that used QDR memory.

Issue a memory reference to an SRAM Channel to move data from the SRAM Queue Array into SRAM.

### Instruction Format

```
sram[cmd, --, src_op1, src_op2]
```

### Parameter Descriptions

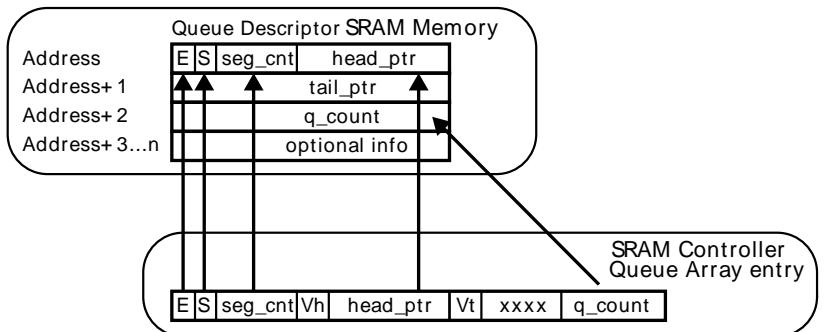
Parameter	Command	Description
cmd	wr_qdesc	This command is used to evict an entry in the SRAM Q_array and returns its contents to SRAM memory. Only the fields that are valid in the Q_descriptor are written to minimize SRAM bandwidth utilization. If the head valid bit is set in the Q_array, the head, seg_cnt, SOP, EOP, and q_count fields are returned to SRAM. If the tail valid bit is set in the Q_array, the tail, and q_count fields are returned to SRAM. If head and tail are valid, the head, seg_cnt, SOP, EOP, tail, and q_count fields are returned to SRAM. If neither the head and tail are valid, nothing is returned to SRAM. Refer to <a href="#">Figure 2.7</a> for a description of fields moved between the Q_array and the SRAM.
	wr_qdesc_count	This command is used to evict the q_count entry in the SRAM Q_array and return its contents to SRAM memory. Refer to <a href="#">Figure 2.7</a> for a description of fields moved between the Q_array and the SRAM.
--	Both commands	Must always be “--”.
src_op1, src_op2	Both commands	Restricted operands that are added (src_op1 + src_op2) to define the following:

Parameter	Command	Description
		<ul style="list-style-type: none"> <li>• [31:30] SRAM Channel</li> <li>• [29:24] Queue Array Entry Number</li> <li>• [23:0] Queue Descriptor data block.</li> </ul> <p>The Address of Queue Descriptor block specifies a 4-byte word address (not a byte address) that must be the start of the 16- byte aligned address of the Queue descriptor (i.e bits [1:0] always should be 0). The queue descriptor consists of the head, tail, q_count and optional data.</p>

### Condition Codes Affected

N	Z	V	C
Not Affected			

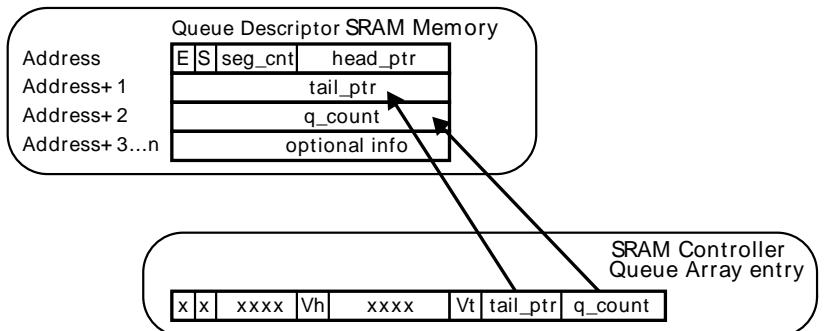
Wr\_qdesc Command with Head Valid



Notes:

"Vh" = Head Valid  
"Vt" = Tail Valid  
"E" = EOP  
"S" = SOP

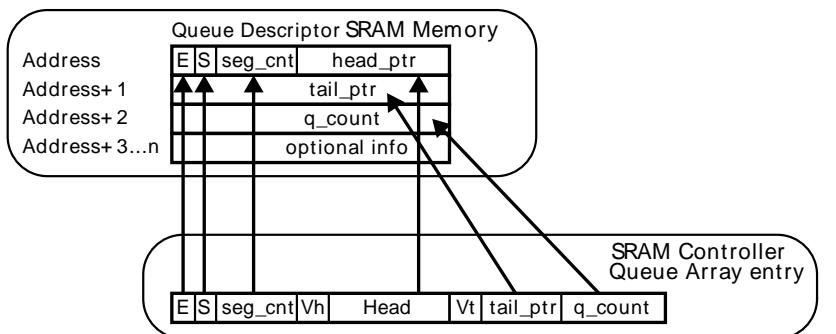
Wr\_qdesc Command with Tail Valid



Queue Descriptor

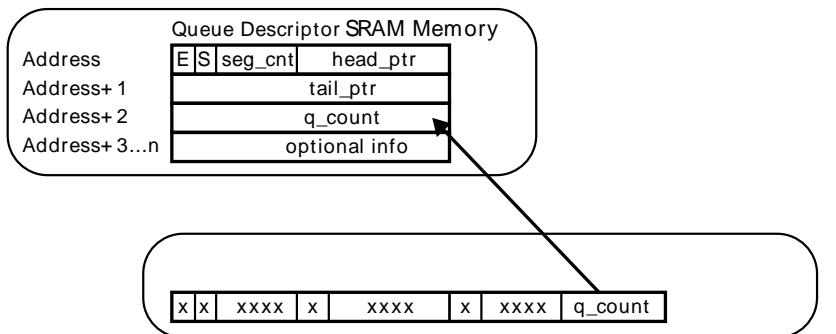
Address  
 [31] - EOP  
 [30] - SOP  
 [29:24] - seg\_cnt  
 [23:0] - head\_ptr  
 Address+1  
 [31:24] reserved  
 [23:0] tail pointer  
 Address+2  
 [31:24] user data\*  
 [23:0] q-count

Other Command with Head and Tail Valid



\*The user data field is written when the queue descriptor is initialized. The queuing hardware will read this value and return it to the ME when a rd\_qdesc instruction is performed. However, this field is never written when a wr\_qdesc instruction is performed. It should be considered read-only user data.

Wr\_descr\_count Command



**Figure 2.7. Write Queue Descriptor Commands**

## 2.2.41 Interlaken Look Aside (ILA)

### 2.2.41.1 ILA (Interlaken Look-Aside)

ILA commands are used to interface to FPGA-based accelerator attachments and communicate between Netronome FPA based devices.

The ILA mode must be configured before the ILA logic is taken out of reset.

Accelerator mode supports 2 ILA channels and 4 DMA queues and is the mode used for inter-Netronome-FPA communication.

ILA block supports two methods of moving data between the Netronome FPA-based devices and the attached device. First method uses the CPP Target interface and second method uses the DMA capabilities of the ILA block for higher performance data transfers.

#### Instruction Format

```
ila[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

#### Parameter Descriptions

Parameter	Command	Description
cmd	read	Read memory of attached ILA device. Signal back immediate with Read Response from ILA Target.
	read_check_error	Read memory of attached ILA device. Signal back immediate with Read Response from ILA Target. If an ILA Rx error is encountered, data is returned with both even and odd signal.
	read_int	Internal Read operation.
	write	Post Write data to ILA attached device. Signal back immediate with Write Response from ILA Target
	write_check_error	Post Write data to ILA attached device. Use even signal when write response is received from ILA attached device. If an ILA Rx error is encountered, 4 bytes data is returned with both even and odd signals.
	write_int	Internal Write operation.
xfer	read, read_check_error, read_int	Read transfer registers containing the data from the ILA.
	write, write_check_error, write_int	Write transfer registers containing the data to be sent to the ILA.
src_op1, src_op2	All commands	Restricted source operands that define byte address. The address is specified by src_op1 + src_op2.

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
		Refer to <a href="#">Section 2.1.6.2</a> . 32-bit or 40-bit address mode is configured by IlaCtrlCfg Register.
read, read_check_error, write, write_check_error	Reference count in increments of 4 byte words. Valid values are 1 to 32. Values above 8 must be specified using indirect reference.	
opt Tok	read_check_error, write_check_error	sig_done[sig_name]  indirect_ref; Refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]
		sig_done[sig_name]  ctx_swap[sig_name]  defer[n] where n = 1 or 2
	read, read_int, write, write_int	indirect_ref; Refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]
		sig_done[sig_name]  ctx_swap[sig_name]  defer[n] where n = 1 or 2
		indirect_ref; Refer to <a href="#">Section 2.1.6.4.2</a> .
		ind_targets[me1, me2,...]

## 2.2.42 Packet Processing Subsystem

### 2.2.42.1 (NBI) Network Block Interface

The Packet Processing subsystem or NBI subsystem is a high performance component of the Netronome NFP Architecture. NBI blocks include the Packet Preclassifier, DMA, Traffic Manager (TM) and Packet Modifier. These instructions below will allow the FPC to transmit/read packets via the TM and write packets to the Cluster Target Memory Unit (CTM) or Memory Unit (MU).

The Packet Engine in the CTM will inform the TM when the packet processing has been completed by the FPC on a particular packet and the packet is ready for transmission using one of the packet\_ready\_\* instructions. The Packet Engine does this by sending one of the packet\_ready\_\* commands to the Egress NBI indicating that the packet is ready for transmission.

Note: Network Block Interface (NBI) is a synonym for Packet Processing Subsystem

#### Instruction Format

```
nbi[cmd, xfer, src_op1, src_op2, ref_cnt], opt Tok
```

#### Parameter Descriptions

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
cmd	packet_ready_drop	This instruction is used for keeping track of sequence numbers only. No memory buffers are freed as a result.
	packet_ready_unicast	For unicast packets, "free on last transfer" is set. The TM will be responsible for freeing the CTM on last Pull ID request.

Parameter	Command	Description
	packet_ready _multicast_dont_free	For common copy multicast packets, the "free on last transfer" is clear. The TM will not free the CTM buffer.
	packet_ready _multicast_free_on_last	For unique copy multicast packets, the "free on last transfer" is set. The TM will be responsible for freeing the CTM on last PULL ID request.
	read	Read a packet from CTM or MU. The NBI uses PullIds for reading the CTM, and Read64 commands for reading the MU.
	write	Write a packet to CTM or MU. The NBI used PullIds for writing to the CTM, and Write64 commands for writing to the MU.
xfer	read	Read transfer registers containing data from the CPP Target Control.
	write	Write transfer registers for CPP Target Control.
	packet_ready_drop, packet_ready_unicast, packet_ready _multicast_dont_free, packet_ready _multicast_free_on_last	Write transfer registers containing data for Traffic Manager.
src_op1, src_op2	packet_ready_drop, packet_ready_unicast, packet_ready _multicast_dont_free, packet_ready _multicast_free_on_last, read, write	Restricted source operands that define byte address. The address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a>
ref_cnt	read, write	Reference count in increments of 8 byte words. Valid values are 1 to 16. Values above 8 must be specified using indirect reference.
opt Tok	packet_ready_drop, packet_ready_unicast, packet_ready _multicast_dont_free, packet_ready _multicast_free_on_last, read, write	sig_done[sig_name] ctx_swap[sig_name] defer[n] where n = 1 or 2 indirect_ref; refer to <a href="#">Section 2.1.6.4.2</a> . ind_targets[me1, me2,...]

## 2.2.43 PCI Express (PCIe)

### 2.2.43.1 PCIE

Move data between PCIe memory or PCIe target internal memory and data masters.

#### Instruction Format

pcie[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok // 32-bit addressing
pcie[cmd, xfer, src_op1, <>8, src_op2, ref_cnt], opt_tok // 40-bit addressing
pcie[cmd, xfer, src_op1, src_op2, <>8, ref_cnt], opt_tok // 40-bit addressing

## Parameter Descriptions

Parameter	Command	Description
cmd	read	32-bit read from PCIe memory to transfer registers.
	read_int	Read n 4-byte words from PCIe internal target to transfer registers. Targets include Internal SRAM, Expansion BAR Configuration registers, Queue Controller, Event Manager, Interrupt Manager, DMA Controller, CPP Master Engine.
	read_pci	See: read_int
	read_rid	Read n 4-bytes words of data from PCIe buffer. Override PCIe Requester ID of PCIe transactions. Using indirect reference to set the Byte_mask[7:0] specifies Requester ID.
	write	32-bit write from transfer registers to PCIe memory.
	write_int	Write n 4-byte words from transfer registers to PCIe internal target. Targets include Internal SRAM, Expansion BAR Configuration registers, Queue Controller, Event Manager, Interrupt Manager, DMA Controller, CPP Master Engine.
	write_pci	See: write_int
xfer	read, read_int, read_pci, read_rid	Read transfer registers containing n number of 4-byte words.
	write, write_int, write_pci, write_rid	Write transfer registers containing n number of 4-byte words.
src_op1, src_op2	read, read_rid, write, write_rid	Restricted source operands that define 4-byte address and encoded data. The address is specified by src_op1 + src_op2. Address[37:35] is the PCIe BAR, Address[34:0] is starting address within PCIe buffer aligned to an 4-byte boundary. Address[1:0] must be 2b00.
	read_int, read_pci, write_int, write_pci	Restricted source operands that define byte address and encoded data. The address is specified by src_op1 + src_op2. Address[19:16] is the Internal target number and Address[15:0] is Internal target address.
ref_cnt	read_int, read_pci, write_int, write_pci	Reference count in 4-byte words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
	read, read_rid, write, write_rid	Reference count in 4-byte words. Valid values are 1-32. Values above 8 must be specified using indirect reference.
opt_tok	All commands	sig_done[sig_name]

Parameter	Command	Description
		ctx_swap[sig_name]
		indirect_ref
		defer[n] where n = 1 or 2
		ind_targets[me1, me2, ...]

### Example 1. pcie[write\_int] and pcie[read\_int]

```
;Issue DMA transaction using pcie[write_int].
.reg $xfer[4]
.xreg_order $xfer
.sig sig1
.areg sig1 1

.reg dmaq_addr_descr
.reg dmaq_addr_upper
.reg dmaq_addr_status
.reg cl_num

; Select the address for DMA enqueue operations, where the
; address is within the PCIe block for accessing a given DMA queue.
; DMA Queue Index is 0, Address of DmaCmdInsertHiToPci register is selected.
immed[dmaq_addr_descr, 0x4000]
alu_shf[dmaq_addr_descr, --, B, dmaq_addr_descr, <<4]
alu_shf[dmaq_addr_upper, --, B, cl_num, <<30]

; Setup Descriptor
alu[$xfer[0], --, B, 0x00000000] ;DMADescrLegacy bits[31:0]
alu[$xfer[1], --, B, 0x00000085] ;DMADescrLegacy bits[63:32]
alu[$xfer[2], --, B, 0x00000000] ;DMADescrLegacy bits[95:64]

immed[tmp, 0x04]
alu_shf[tmp, tmp, or, 0x3f, <<20]
alu[$xfer[3], tmp, or, 0] ;DMADescrLegacy bits[127:96]

; Issue DMA 40-bit Address (DmaCmdInsertHiToPci)
pcie[write_int, $xfer[0], dmaq_addr_upper, <<8, dmaq_addr_descr, 4], sig_done[sig1]
ctx_arb[sig1]

; Read DMA status 40-bit Address (DMAQstatToPCI0, DMAQstatToPCI1)
; Note ref_cnt is equal to 2, $xfer[0] will contain contents of DMAQstatToPCI0
; and $xfer[1] will contain the contents of DMAQstatToPCI1
; DMA Status provides the following:
; Get the maximum number of available entries in LO, MED and HI Queues,
; state of DMA (run/stop), DMA error status bits.
alu[dmaq_addr_status, dmaq_addr_descr, OR, 0xe0]
pcie[read_int, $xfer[0], dmaq_addr_upper, <<8, dmaq_addr_status, 2], sig_done[sig1]
ctx_arb[sig1]
```

### Example 2. pcie[write]

```
; Issue pcie[write] to enable bus mastering.
```

```
.sig sig1
.reg $xfer
.reg addr_msb
.reg cl_num, me_num
.reg r_bar0

.if (ctx() == 0)

; Get Island ID and ME number this code is running on.
; The ME_NUM is in bits [6:3]
; The Island ID or cl_num is in bits [30:25]
local_csr_rd[ACTIVE_CTX_STS]
immed[cl_num,0]
alu_shf[me_num,0xF, AND,cl_num,>>3]
alu_shf[cl_num,0x3F, AND,cl_num,>>25]

; Configure egress BAR to enter EP PCIe config space
alu_shf[addr_msb, --, B, cl_num, <<30]
alu_shf[$xfer, --, B, 2, <<30] ; MapType=2, address=0, overrideRID=0
immed[r_bar0, 0x180] ; Offset cppToPcieBar CPP to PCIe translator BAR
alu_shf[reg_bar0, reg_bar0, OR, 0x30, <<12] ; NFP_PCIE_BAR_CONFIG_BASE=0x30000
;
pcie[write_int, $xfer, addr_msb, <<8, r_bar0, 1], sig_done[sig1]
ctx_arb[sig1]

; set PCIe command_status register to enable bus mastering by EP
; Enable IO Space and MEM Space accesses through the core
alu[$xfer, --, B, 7]
;
pcie[write, $xfer, addr_msb, <<8, 4, 1], sig_done[sig1]
ctx_arb[sig1]
```

## 2.2.44 Crypto

### 2.2.44.1 crypto(Operations)

The Cryptography Engine supports Bulk Cryptography plus Authentication. The Cryptography Engine receives Security Context, Packet data and instructions for processing over the system bus. The instructions setup the processing modes as well as perform the processing steps.

The Crypto Engine has a simple queuing structure for instructions feeding six bulk cryptography blocks and one DMA block. Each bulk build cryptography core operates on a set of control information and data, which are supplied through the queues. The different classes of cores maintain their own queues of commands that consist of core class, action, operand and trigger.

#### Instruction Format

```
crypto[cmd, xfer, src_op1, src_op2, ref_cnt], opt_tok
```

#### Parameter Descriptions

<b>Parameter</b>	<b>Command</b>	<b>Description</b>
cmd	read	Read data from crypto buffer.
	write	Write data into the crypto buffer.
	write_fifo	Provide a command for the crypto command FIFO.
xfer	read	Read transfer registers.
	write, write_fifo	Write transfer registers.
src_op1, src_op2	read, write	Restricted source operands that define the byte address. The 32-bit address is specified by src_op1 + src_op2. Refer to <a href="#">Section 2.1.6.2</a>
	write_fifo	Restricted source operands that define the byte address. The 40-bit address is specified by src_op1 and src_op2. Refer to <a href="#">Section 2.1.6.2</a>
ref_cnt	All commands	Reference count in 8-byte words. Valid values are 1-16. Values above 8 must be specified using indirect reference.
opt_tok	All commands	sig_done[sig_name]
		ctx_swap[sig_name]
		indirect_ref
		defer[n] where n = 1 or 2
		ind_targets[me1, me2, ...]

## 3. FPC Address Map

---

Unlike older network processors, the Netronome NFP does not have a flat, hardcoded memory map. This provides more flexible and dynamic memory space.

This section describes the FPC memory view of the functional units. The FPCs support commands that allow access to the functionality listed in [Table 3.1](#).

For ARM view of the memory and address map refer to [Section 3.2.1](#) and PCIe refer to [Section 3.2.2](#).

For lower-bandwidth interaction, the units communicate via the Expansion Peripheral Bus (XPB) interconnect with the bus master in each island. Refer to [Section 3.2.10](#) for XPB Target Addressing.

**Table 3.1. FPC I/O Access**

Unit	Functionality	Instruction/ Commands	Comments
FPC	FPC's Own Local CSRs	LOCAL_CSR_RD LOCAL_CSR_WR	CSR names are used to specify the CSRs. Refer to the <a href="#">LOCAL_CSR_RD</a> and <a href="#">LOCAL_CSR_WR</a> instructions for a list of supported CSR names.
	Other FPC Transfer Registers	ct[reflect_read_none] ct[reflect_read_sig_both] ct[reflect_read_sig_init] ct[reflect_read_sig_remote] ct[reflect_write_none] ct[reflect_write_sig_both] ct[reflect_write_sig_init] ct[reflect_write_remote]	Refer to <a href="#">Section 3.2.10.3</a> for XPB CTM Reflect XFER memory maps.
	Other FPC Local CSRs	ct[reflect_read_none] ct[reflect_read_sig_both] ct[reflect_read_sig_init] ct[reflect_read_sig_remote] ct[reflect_write_none] ct[reflect_write_sig_both] ct[reflect_write_sig_init] ct[reflect_write_remote]	Refer to <a href="#">Section 3.2.10.2</a> for XPB CTM Reflect CSR memory maps.
	Other FPC GPRs	No Access	No access to other FPC GPRs.
PCIe	PCIe Memory	pcie[read], pcie[read_int], pcie[write], pcie[write_int]	Refer to <a href="#">Section 3.2.2</a> PCIe memory maps.
	PCIe CSRs	pcie[read], pcie[read_int], pcie[write], pcie[write_int]	Refer to <a href="#">Section 3.2.2</a> PCIe memory maps.
SRAM Deprecated	SRAM Memory	sram[read], sram[write], sram[swap], sram[set], sram[clr], sram[incr], sram[decr], sram[add] sram[test_and_set],	The base addresses of the SRAM channels are:

<b>Unit</b>	<b>Functionality</b>	<b>Instruction/ Commands</b>	<b>Comments</b>
		sram[test_and_clr], sram[test_and_incr], sram[test_and_decr], sram[test_and_add]	<ul style="list-style-type: none"> <li>• Channel 0: 0x0000 0000</li> <li>• Channel 1: 0x4000 0000</li> <li>• Channel 2: 0x8000 0000</li> <li>• Channel 3: 0xc000 0000.</li> </ul> <p>The size of the memory installed determines the upper bound of the address.</p> <p>The Virtual-QDR uses bits 31:30 of address to select the channel. FPCs using SRAM command use top 2 bits of address to select the channel.</p>
	SRAM Queue Array	sram[rd_qdesc_head] sram[rd_qdesc_tail] sram[rd_qdesc_other] sram[wr_qdesc] sram[wr_qdesc_count] sram[enqueue] sram[enqueue_tail] sram[dequeue] sram[get] sram[put] sram[journal] sram[fast_journal]	The Queue Array entry and SRAM Channel and SRAM address is determined by src_opA + src_opB. Refer to the instruction definition for more details.
Ext-MU, Int-MU, CTM	External MU CSRs	No Access	
	Memory Unit	mem[add], mem[add_imm], mem[add_imm_sat], mem[add_sat], mem[add_tail], mem[add64], mem[add64_imm], mem[add64_imm_sat], mem[add64_sat], mem[addsat], mem[addsat_imm], mem[addsat64], mem[addsat64_imm], mem[atomic_write], mem[atomic_write_imm], mem[cam_lookup24_add_inc], mem[clr], mem[clr_imm], mem[compare_write], mem[decr], mem[decr64], mem[incr], mem[incr64], mem[read], mem[read_atomic], mem[read_be], mem[read_le], mem[read_swap], mem[read_swap_be], mem[read_swap_le], mem[read8], mem[read8_be], mem[set], mem[set_imm], mem[sub], mem[sub_imm], mem[sub_imm_sat], mem[sub_sat], mem[sub64], mem[sub64_imm], mem[sub64_imm_sat],	Refer to <a href="#">Section 3.2.5</a> for MU External memory maps.  Refer to <a href="#">Section 2.2.39</a> for complete list of instructions.

Unit	Functionality	Instruction/ Commands	Comments
		mem[sub64_sat], mem[subsat], mem[subsat_imm], mem[subsat64], mem[subsat64_imm], mem[swap], mem[swap_imm], mem[test_add], mem[test_add_imm], mem[test_addsat], mem[test_add64], mem[test_addsat_imm], mem[test_add64_imm], mem[test_addsat64], mem[test_addsat64_imm], mem[test_and_add], mem[test_and_add_imm], mem[test_and_add_imm_sat], mem[test_and_add_sat], mem[test_and_add64], mem[test_and_add64_imm], mem[test_and_add64_imm_sat], mem[test_and_add64_sat], mem[test_and_clr], mem[test_and_clr_imm], mem[test_and_compare_write], mem[test_and_set], mem[test_and_set_imm], mem[test_and_sub], mem[test_and_sub_imm], mem[test_and_sub_imm_sat], mem[test_and_sub_sat], mem[test_and_sub64], mem[test_and_sub64_imm], mem[test_and_sub64_imm_sat], mem[test_and_sub64_sat], mem[test_and_xor], mem[test_and_xor_imm], mem[test_clr], mem[test_clr_imm], mem[test_set], mem[test_set_imm], mem[test_sub], mem[test_sub_imm], mem[test_sub64], mem[test_sub64_imm], mem[test_subsat], mem[test_subsat_imm], mem[test_subsat64], mem[test_subsat64_imm], mem[test_xor], mem[test_xor_imm], mem[write], mem[write_atomic], mem[write_atomic_imm], mem[write_be], mem[write_le], mem[write_swap], mem[write_swap_be], mem[write_swap_le], mem[write8], mem[write8_be], mem[write8_le], mem[write8_swap],	

<b>Unit</b>	<b>Functionality</b>	<b>Instruction/ Commands</b>	<b>Comments</b>
		mem[write8_swap_be], mem[write8_swap_le], mem[xor], mem[xor_imm] .....	
Cluster Local Scratch	Cluster Local Scratch Memory	cls[add], cls[add_imm], cls[add_imm_sat], cls[add_sat], cls[add_tail], cls[add64] cls[add64_imm], cls[addsat_imm], cls[tcam_lookup16], cls[tcam_lookup16_add], cls[tcam_lookup24], cls[tcam_lookup24_add], cls[tcam_lookup24_add_inc], cls[tcam_lookup24_add_or_inc], cls[tcam_lookup32], cls[tcam_lookup32_add], cls[tcam_lookup8], cls[tcam_lookup8_add], cls[clr], cls[clr_imm] cls[cmp_rw], cls[decr], cls[decr64], cls[get], cls[get_safe], cls[hash_mask], cls[hash_mask_clear], cls[incr], cls[incr64], cls[journal], cls[pop], cls[pop_safe], cls[put], cls[queue_lock], cls[queue_unlock], cls[read], cls[read_be], cls[read_le], cls[reflect_read_sig_both], cls[reflect_read_sig_local], cls[reflect_read_sig_remote], cls[reflect_from_sig_both], cls[reflect_from_sig_dst], cls[reflect_from_sig_src], cls[reflect_read], cls[reflect_to_sig_both], cls[reflect_to_sig_dst], cls[reflect_to_sig_src], cls[reflect_write], cls[reflect_write_sig_both], cls[reflect_write_sig_local], cls[reflect_write_sig_remote], cls[ring_add_to_tail_ptr], cls[ring_get], cls[ring_get_freely], cls[ring_journal], cls[ring_pop], cls[ring_pop_freely], cls[ring_put], cls[ring_workq_add_thread], cls[ring_workq_add_work], cls[set] cls[set_imm], cls[sub] cls[sub_imm], cls[sub_imm_sat], cls[sub_sat], cls[sub64] cls[sub64_imm], cls[subsat], cls[subsat_imm], cls[swap], cls[tcam_lookup], cls[tcam_lookup_byte], cls[tcam_lookup16], cls[tcam_lookup24], cls[tcam_lookup32], cls[tcam_lookup8], cls[test_addsat], cls[test_addsat_imm], cls[test_and_add_imm_sat],	Refer to <a href="#">Section 3.2.9</a> for CLS Direct Access Memory map.  Refer to <a href="#">Section 2.2.37</a> for complete list of instructions.

<b>Unit</b>	<b>Functionality</b>	<b>Instruction/ Commands</b>	<b>Comments</b>
		cls[test_and_add_sat], cls[test_and_add64_sat], cls[test_and_clr], cls[test_and_clr_imm], cls[test_and_compare_write], cls[test_and_set], cls[test_and_set_imm], cls[test_and_sub_imm_sat], cls[test_and_sub_sat], cls[test_and_sub64_imm_sat], cls[test_clr], cls[test_clr_imm], cls[test_compare_write], cls[test_set], cls[test_set_imm], cls[test_subsat], cls[test_subsat_imm], cls[write], cls[write_be], cls[write_le], cls[write8], cls[write8_be], cls[write8_le], cls[xor] .....	
Cluster Target	Cluster Target	ct[write], ct[xpb_read], ct[xpb_write], ct[ring_put], ct[ring_get], ct[ctnn_write], ct[interthread_signal], ct[reflect_read_sig_init], ct[reflect_read_sig_remote], ct[reflect_read_sig_both], ct[reflect_read_none], ct[reflect_write_sig_init], ct[reflect_write_sig_remote], ct[reflect_write_sig_both], ct[reflect_write_none]	Refer to <a href="#">Section 3.2.10</a> for XPB CTM memory map.
ILA	Interlaken Look-Aside	ila[read], ila[read_check_error], ila[read_int], ila[write], ila[write_check_error], ila[write_int]	Refer to <a href="#">Section 3.2.3</a> for ILA memory map.
NBI	Network Block Interface	nbi[packet_ready_drop], nbi[packet_ready_unicast], nbi[packet_ready_multicast_dont_free], nbi[packet_ready_multicast_free_on_last], nbi[read], nbi[write]	Refer to <a href="#">Section 3.2.4</a> for NBI memory map.
Crypto	Crypto Unit	crypto[read], crypto[write], crypto[write_fifo]	Refer to <a href="#">Section 3.2.11</a> for Crypto memory map.

**Note:**

1. The top two bits of a 40-bit memory unit address are encoded to specify the memory addressing type as in the following table:

<b>Bit value</b>	<b>Addressing Mode</b>
00	High locality of reference data. Access through cache system, where cache lines are used as write-back on eviction and marked as “preferred to keep”.

Bit value	Addressing Mode
01	Low locality of reference data. Access through cache system, where cache lines are used as write-back soon and marked as “preferred to evict”.
10	Direct access. Uses bottom 22-bits of supplied address as data cache address to access, avoiding the use of the cache tag system, and is therefore faster than cached accesses. It must be used with care, as it would be unwise to use a location in cache memory for both direct access and cached accesses. Refer to the NFP-6xxx Databook.
11	Discard after read. Any writes that have been performed to the memory will be lost because the cache line is marked as clean and not written back to DDR memory. This operating mode should only be used for transient data, such as packets on their final transmission from the system. Refer to the NFP-6xxx Databook.



### Note

In the event that the DDR channels are not populated for any given External Memory Unit, that Memory Unit is still available in the system as a pseudo 'Internal' Memory Unit. It is software's responsibility to configure the unit for and address it only in direct access mode for this scenario.

## 3.1 Functional Units

The NFP's functional units are the following:

### ARM:

ARM11 MP (single core), primarily little endian, connects to the CPP bus via a gateway (memory-mapped into the ARM's physical address space, so that CPP targets occupy different addresses on the ARM's memory bus). Resides within a FPC cluster. The ARM's clock speed is the speed of the CPP bus, while in contrast, a FPC's speed is twice the clock frequency of the CPP bus.

### Cryptography

NFP-6xxx includes a cryptography block includes 50Gbps cryptography block that supports Bulk cryptography plus authentication. The cryptography is accessible to all FPCs, all PCIe units, all NPPC/TM units, all ILA and ARM11. The bulk cryptography engines can run the following: AES, DES, 3DES, 3GPP UEA2(Snow3g f8) stream cipher, 3GPP UEA1(Kasumi f8) stream cipher, UIA2 Snow3G f9 (3GPP MAC), UIA1 Kasumi f9 (3GPP MAC) SHA-1, SHA-2, MD5 and ARC4

### High Speed Network/Switch Fabric

The NFP-6xxx's interface and backplane connectivity is controlled by its highly configurable Media Access Controller (MAC) blocks. Each of the NFP's two MAC blocks incorporates multiple Ethernet and Interlaken (including Look-Aside) logic units that provide connectivity through 24 high-speed SerDes lanes in various combinations of 100GE(CAUI), and multiples of 40GE(XLAUI), 10GE(XFI) including KR with FEC and Auto-Negotiation, 1GE(SGMII and 1000BaseX) and up to 100Gb Interlaken. The Ethernet MACs provide both Pause and Priority Flow

Control(PFC) capability. The Interlaken cores can both be operated in standard Interlaken mode or one of the Interlaken cores can also be reassigned as an Interlaken Look-Aside interface for TCAM access.

**ILA:**

Interlaken Look-Aside is an interface to external devices such as off the shelf TCAMS, and FPGA-base accelerators. The ILA block can also serve as a communication interface between devices in the Netronome Flow Processing Architecture. The major components of the ILA are a Queue Controller, Buffer Manager and DMA Engine, all of which exist to ensure efficient data throughput over SERDES lanes of rates of 6.375Gbps, 10.3125Gbps, 11.1 Gbps and 12.5Gbps.

**Memory unit:**

Cluster Target memory (CTM) Unit: Intra-cluster memory resource, 256KB local SRAM with ECC. Highly flexible and high performance bulk memory transaction acceleration engine. Atomic transaction acceleration engine supplies broad selection of atomic operations. Packet acceleration engine, supplies coordinated DMA data transfers to the local SRAM via a CPP master interface and supporting packet addressing and translation. Misc acceleration engine that serves as the XPB master, handling translation from CPP commands to XPB master operations for the cluster. A CPP master interface for DMA operations.

Internal Memory Unit: Two CPP target interfaces, supporting both a DDR and 'virtual QDR' target which remap CPP commands to internal memory unit commands. 4MB direct-access SRAM memory arranged as 64k 64 byte lines with ECC. Build memory transaction acceleration engines, which provides hight flexible very high performance bulk data transfers. Atomic transaction acceleration engine that supplies a broad selection of atomic operations. Lookup acceleration engines that performs many lookup table and data matching functions as needed by the packet processing and classification requirements of the NFP. Statistics acceleration engine provides for fast packet and byte count statistical data capture. Load balancing adaptive link aggregation group (LAG) acceleration engine that provides a dynamic hash-based mechanism for managing and rebalancing microflows based on traffic flow. Direct connection to the NPPC/TM to SRAM for lookup table storage and access.

External (DDR3 DRAM) Memory Unit, 2MB cache, 64-byte cache line. Unaligned bulk data transfer engine. Atomic operation engine, which works on cache lines, and even includes CAM lookups. Queue engine. Two CPP target interfaces, and two *virtual QDR* targets that remap CPP-encoded IXP compatible QDR commands into memory unit commands.

**Flow Processing Clusters (Islands):**

The Flow Processing Cluster represents the programmable processing center of the NFP-6xxx. There are two different types of FPC clusters. One type contains a service accelerator/processor (the ILA, PCIe, or Crypto accelerators or ARM processor) and 4 multi-threaded FPCs which serve as the command center for the service accelerator/processor.

The other type is a pure FPC cluster with 12 multi-threaded FPCs.

FPCs in all clusters operate at 1.2GHz and provide complex processing capability such as deep packet walking and parsing, flow state analysis and adjustments, including packet insertion and reordering, and deep packet classification and action processing. Packets are served to and from the FPCs without using FPC processing cycles through an innovative Packet Engine located in the 256KB Cluster Target Memory unit at the periphery of the Flow Processing Cluster. Each Flow Processing Cluster is fitted with 64KB of scratchpad memory with accelerators for up to 16 circular buffer rings, atomic transactional and locking actions, CAM and TCAM searches through the scratchpad, and Hash operations with up to a 128-bit remainder.

**Island target:**

Access an FPC's next-neighbor registers as a FIFO. The cluster target/XPB address mode specifies the island in the target CPP command address bits [6;24] (thus partial address decode handled by the CPP bus). For remaining address modes refer to the *NFP-6xxx Databook*

**Cluster Local Scratch:**

Mid-tier storage for FPC programs (64kB SRAM, 64-bits wide with ECC, with unaligned read and write). It offers lower latency access than QDR, but higher-latency than FPC registers, 64-bit words gives highest performance. Offers atomic operations on memory, and locking transactions for synchronization of threads, with signaling. Also provides CAM mechanism with TCAM and string search options. 16 rings (circular buffers) for communication between masters within the cluster. Master to master data and signal reflect, hash computations with up to 128-bit remainder, event manager, interrupt manager, and a true random number generator.

**FPC core:**

Posted CPP transactions.

**Packet Processing SubSystem:**

The Packet Processing Subsystem provides a wide array of dedicated hardware functionality to support the high bandwidth processing and packet manipulation requirements of the FPA (Flow Processor

Architecture). This hardware includes the Netronome Packet Processing Cores (NPCCs) on Ingress, and the Packet Modifier and Traffic Manager on Egress. The Packet Processing Subsystem resides between the Media Access Controller (MAC) subsystem and the FPA DSF CPP Fabric, which provides ready access to the Flow Processing Cores.

**PCIe unit:**

Provides four PCIe Gen-3 controllers and PHYs. Each PCIe component provides both a control plane and a data plane interconnect through PCI Express transaction to internal registers, and any CPP target such as Internal Memory Units and External Memory Units, for access to DDR3 memory. It also generates PCI Express transactions from an internal DMA Engine or in response to specific CPP master transactions. So it provides a bidirectional gateway between the PCIe bus and the CPP bus. It has various modes for mapping CPP requests to PCIe requests, and vice versa. It initiates PCIe DMA. It maps NFP events to PCIe interrupts. It can serve as a PCIe endpoint or root complex. As a root complex, it can receive and fulfill DMA requests.

## 3.2 Target Memory Maps

Target Memory Maps section shows the address mapping for the functional units in the NFP Architecture. The following memory targets exist on the NFP-6xxx:

- ARM11 Target
- PCIe Target
- ILA Target
- NBI Target
- Crypto Target
- Memory Unit Targets (External MU, Internal MU, and CT Memory)
- Cluster Local Scratch Target
- XPB Target (Chip Level XPB addressing and Cluster Target Memory addressing) and further broken down into specific sub-modules that are shown below.



### Note

Not all Targets have HW support on every Island type.

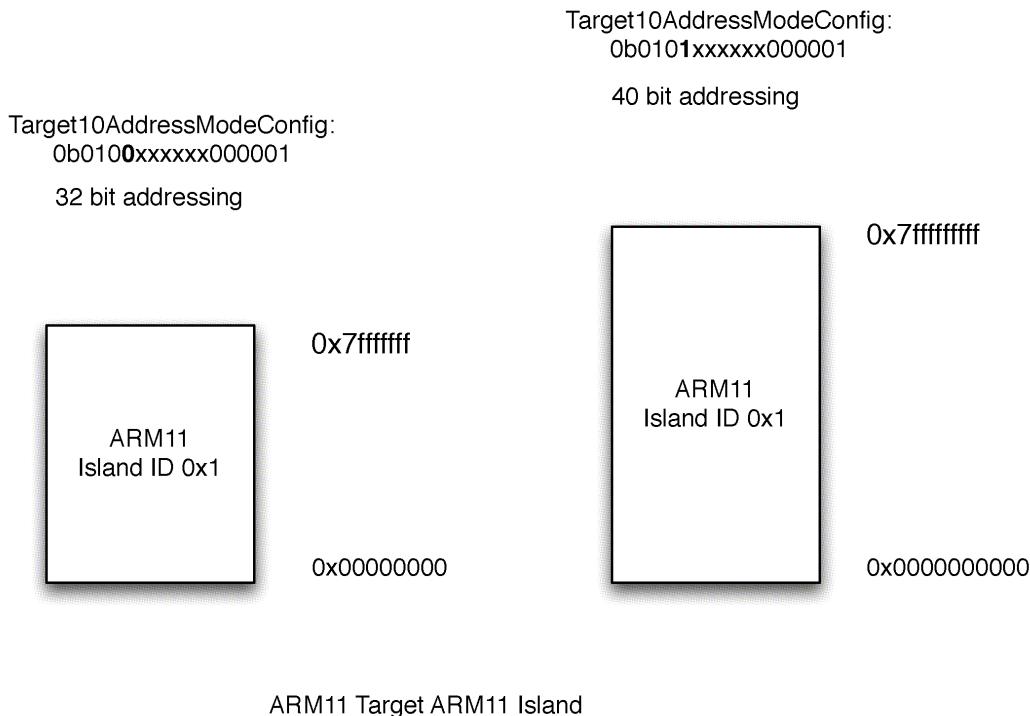


### Note

For all targets other than the XPB Target, two different modes of addressing are shown: 32-bit mode and 40-bit mode. The 32-bit addressing modes are being deprecated in the NFP-6xxx architecture. They are illustrated here for completeness and as a point of reference to previous versions of the NFP processor, and for backwards compatibility support. 40-bit addressing mode is the recommended method of operation.

### 3.2.1 ARM Memory Map

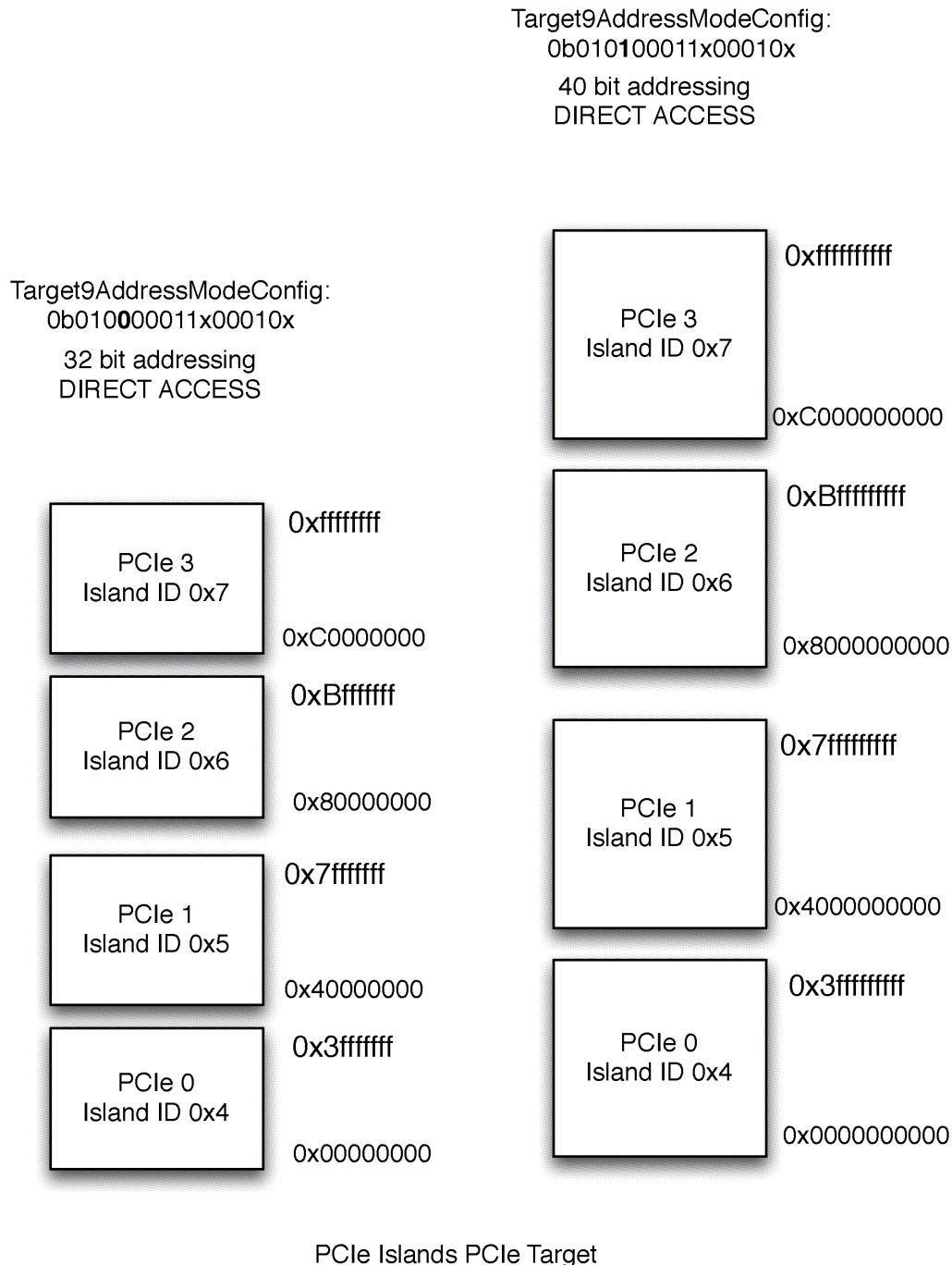
The ARM11 memory space is completely configurable. A default configuration does exist. For more information regarding the memory usage on the ARM11 core, and details of the partitioning to enable processing of CPP commands, see the ARM11 chapter in the Databook, and specifically the subsection Address Mapping.



**Figure 3.1. ARM Memory Map**

### 3.2.2 PCIe Memory Map

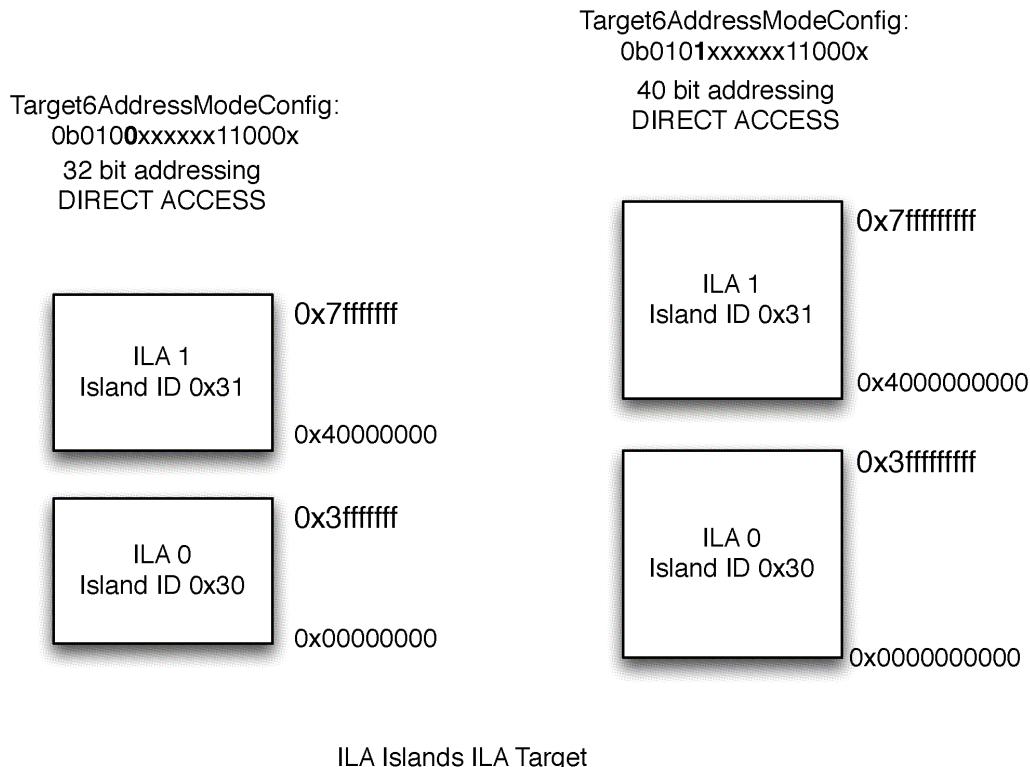
In addition to the general space shown in the figure, details of the hierarchical partitioning of the addressable PCIe space are shown in the PCIeInternalTargetsCPPAddressMap Address Map Table in the NFP-6xxx Databook.



**Figure 3.2. PCIe Memory Map**

### 3.2.3 ILA Memory Map

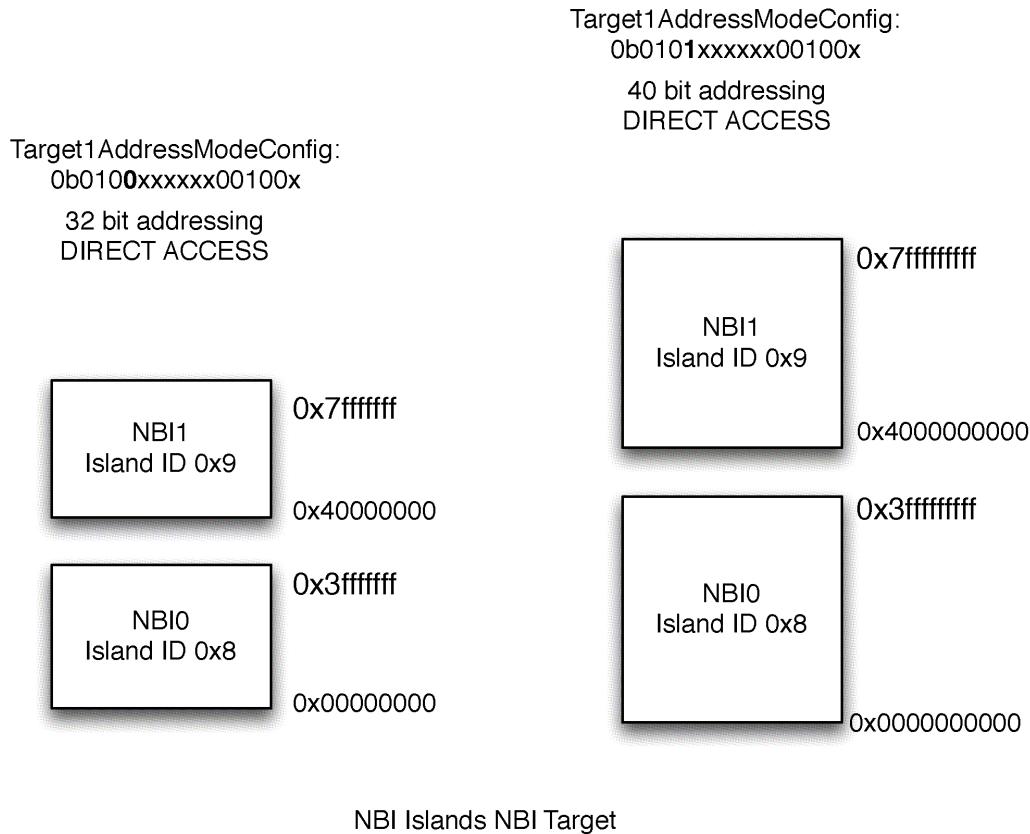
The general ILA Target address ranges for the ILA Islands are shown in the figures below. The first 64kB of these ranges are Local Scratch SRAM. ILA related CSRs for functional configuration of ILA HW supported functionality can be addressed as shown in the table ILACPPAddressMap Address Map and associated maps found in the NFP-6xxx Databook.



**Figure 3.3. ILA Memory Map**

### 3.2.4 NBI Memory Map

In addition to the general space shown in the figure, details of the hierarchical partitioning of the addressable NBI space are shown in the NBIAccessMap Address Map Table in the NFP-6xxx Databook.



**Figure 3.4. NBI Memory Map**

### 3.2.5 MU External Memory Map

The three external memory units are the only ones which can be accessed outside of Direct Access mode. There are different types of cache access locality which govern the cache behavior when accessing locations in these external MUs. Different locality memory ranges are shown for each of the three external MUs. Refer to the NFP-6xxx Databook “Hierarchical Memory Sub-System” section for further details of operating and memory access to the MEM Targets.

### 3.2.5.1 MU External 0 Memory Map

Target7AddressModeConfig:  
0b0110xxxxx0110xx

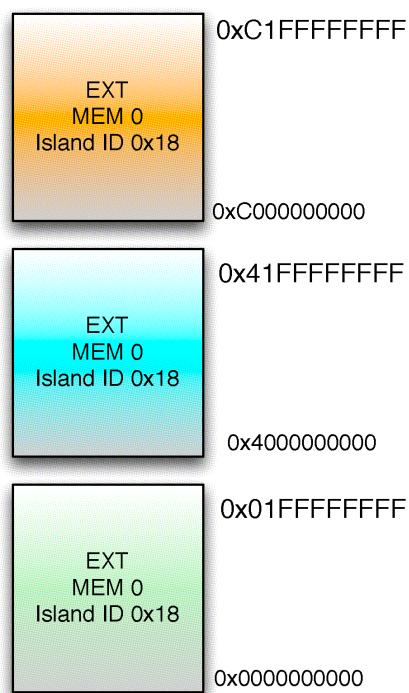
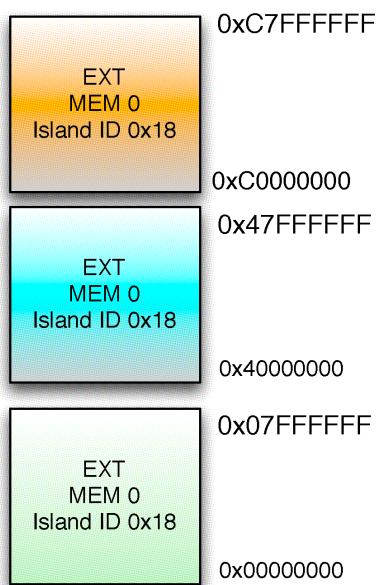
Target7AddressModeConfig:  
0b0111xxxxx0110xx

External Memory Unit 0  
Target Address  
Range of Values - Note 1



40 bit addressing

32 bit addressing



0x01FFFFFF  
0x0000000000

Note 1: The lowest (green) address values are for the case of High Locality of Reference Data access mode, which affects the cache behavior upon these memory accesses. If Low Locality of Reference Data is to be used, use the middle (blue) range of shown values. If Discard After Read access mode is to be used, use the highest (orange) range of shown values.

**Figure 3.5. MU External 0 Memory Map**

### 3.2.5.2 MU External 1 Memory Map

Target7AddressModeConfig:  
0b0110xxxxxx0110xx

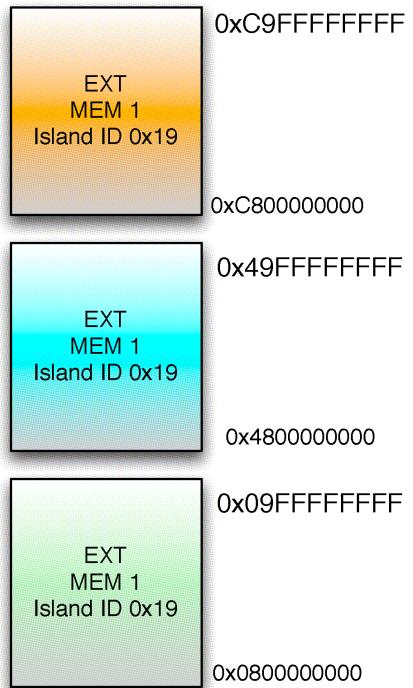
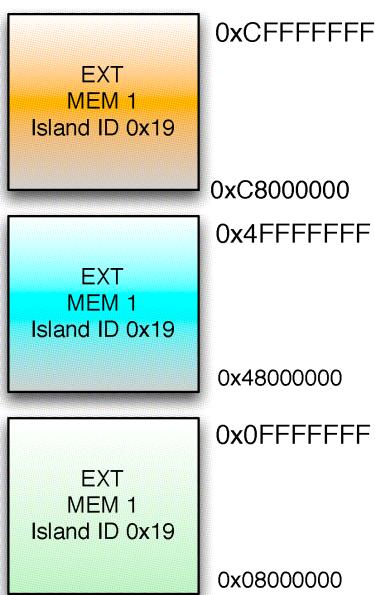
Target7AddressModeConfig:  
0b0111xxxxxx0110xx

External Memory Unit 1  
Target Address  
Range of Values - Note 1



40 bit addressing

32 bit addressing



Note 1: The lowest (green) address values are for the case of High Locality of Reference Data access mode, which affects the cache behavior upon these memory accesses. If Low Locality of Reference Data is to be used, use the middle (blue) range of shown values. If Discard After Read access mode is to be used, use the highest (orange) range of shown values.

**Figure 3.6. MU External 1 Memory Map**

### 3.2.5.3 MU External 2 Memory Map

Target7AddressModeConfig:  
0b0110xxxxxx0110xx

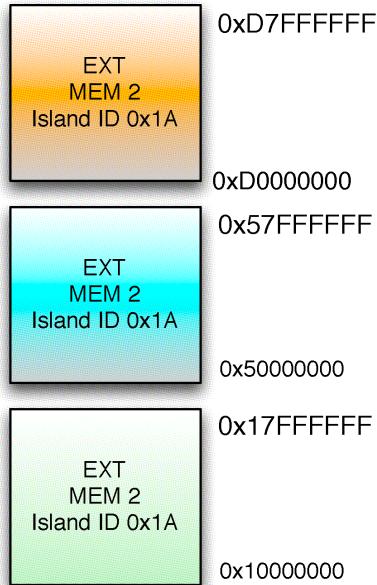
Target7AddressModeConfig:  
0b0111xxxxxx0110xx

External Memory Unit 2  
Target Address  
Range of Values - Note 1



40 bit addressing

32 bit addressing

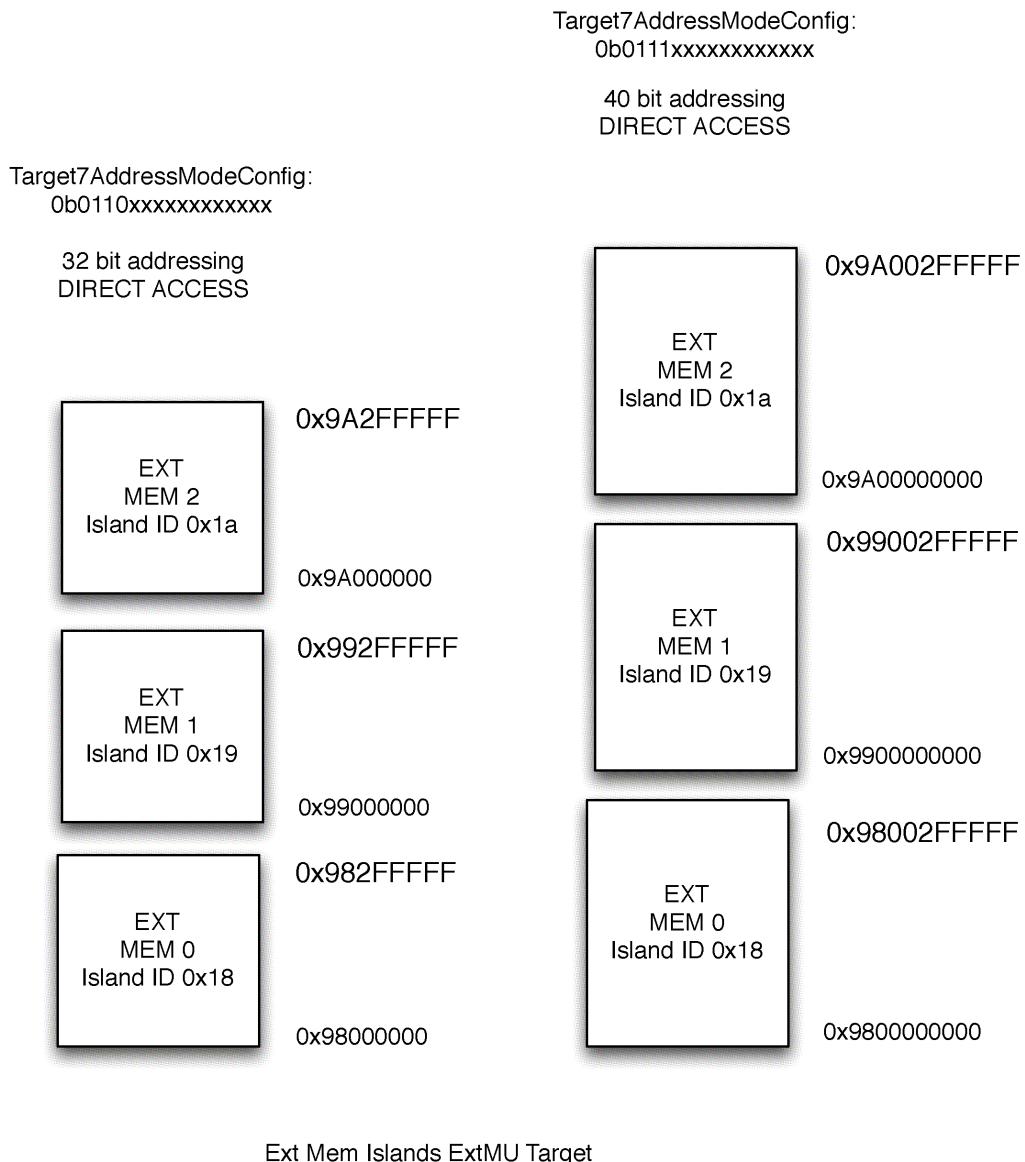


Note 1: The lowest (green) address values are for the case of High Locality of Reference Data access mode, which affects the cache behavior upon these memory accesses. If Low Locality of Reference Data is to be used, use the middle (blue) range of shown values. If Discard After Read access mode is to be used, use the highest (orange) range of shown values.

**Figure 3.7. MU External 2 Memory Map**

### 3.2.6 MU Direct Access Memory External Memory Map

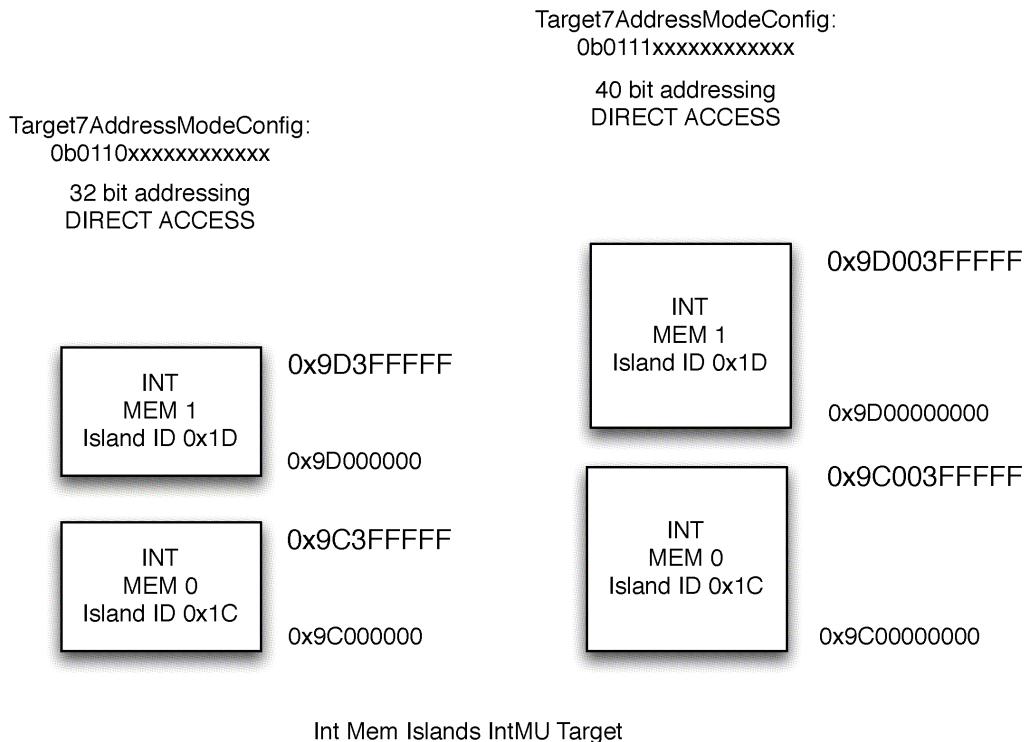
The External Memory Unit can be accessed through Direct Access Mode as well, as shown in the figure below.



**Figure 3.8. MU Direct Access Memory External Memory Map**

### 3.2.7 MU Direct Access Memory Internal Memory Map

The Internal Memory Unit can only be accessed through Direct Access Mode.



**Figure 3.9. MU Direct Access Memory Internal Memory Map**

### 3.2.8 MU Direct Access Target CTM Memory Map

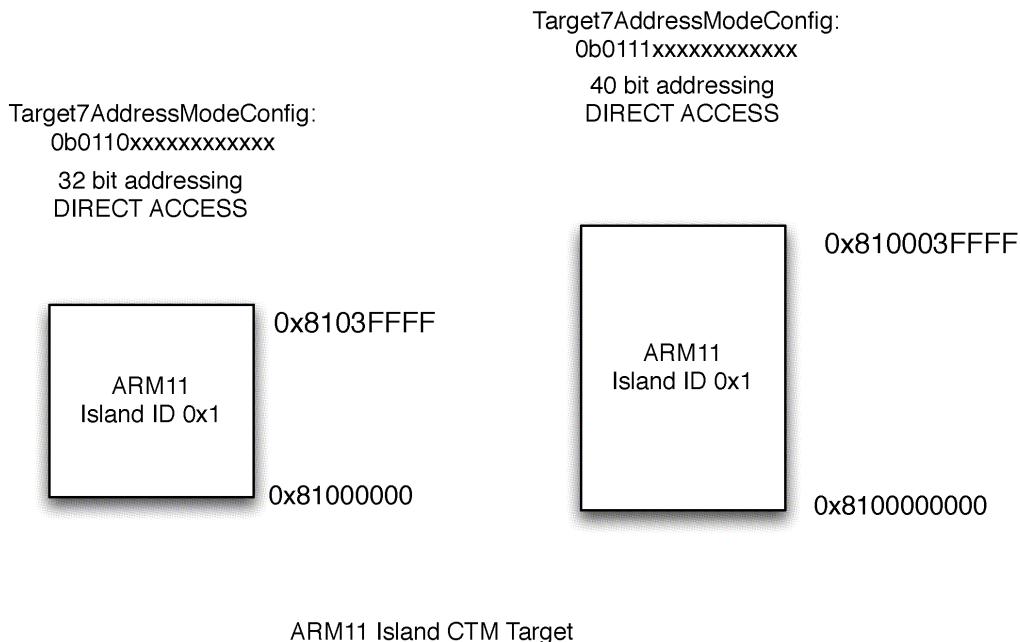
A number of different service master islands support Cluster Target Memory as an addressable Target. These are:

- ARM11 Island
- PCIe Islands
- Crypto Islands
- FPC Islands
- ILA Islands

The addressable memory spaces for each of these Island Family types are shown in the sub-sections [Section 3.2.8.1](#) through [Section 3.2.8.4](#).

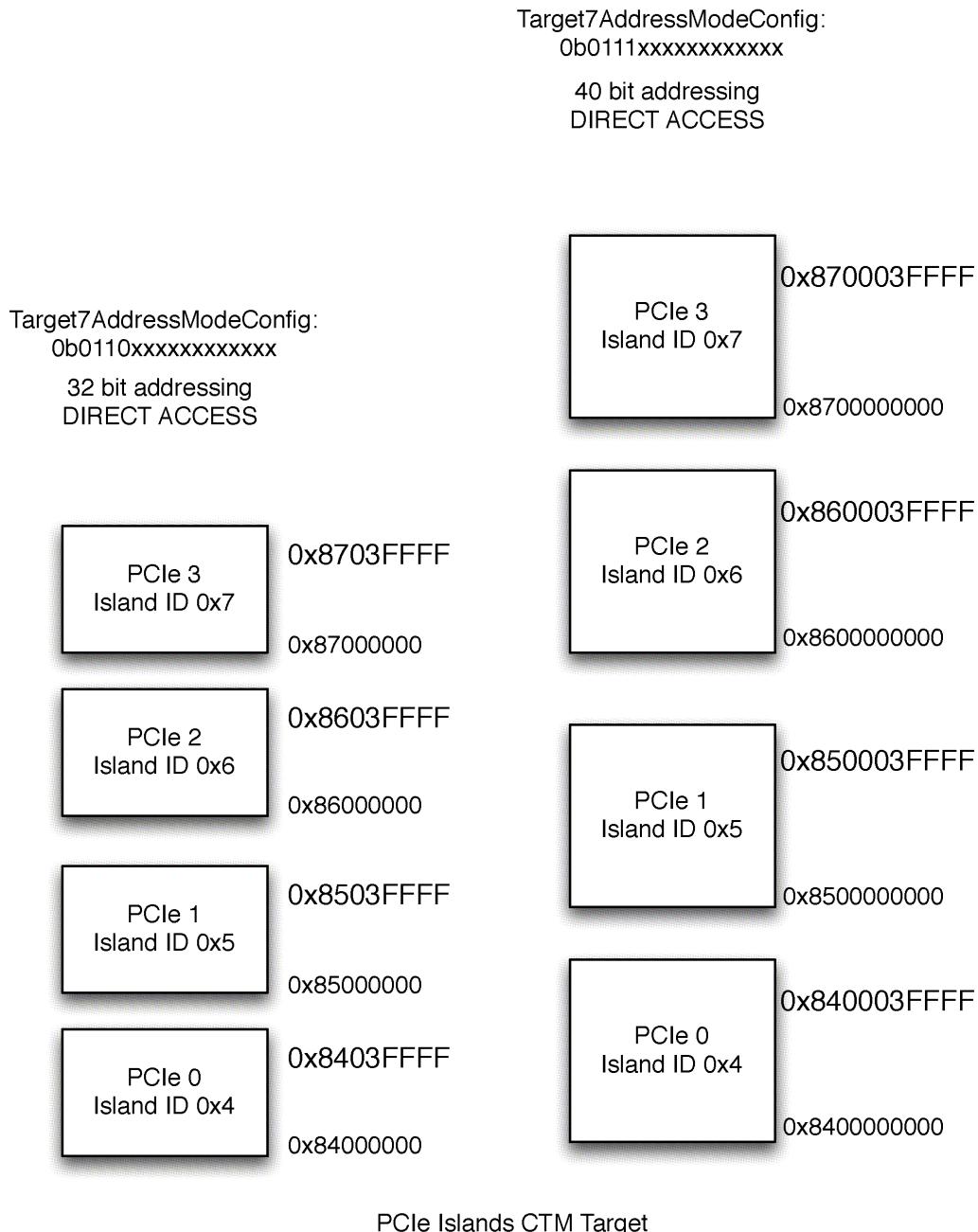
It is also possible to reference the Cluster Target Memory on the Island in which the FPC is located. This is referred to as the “Self Island”. The Self Island will always be addressed using island ID 0x0. The addressable memory space for Self Island addressing is shown in sub-section [Section 3.2.8.5](#).

### 3.2.8.1 MU Direct Access ARM CTM Memory Map



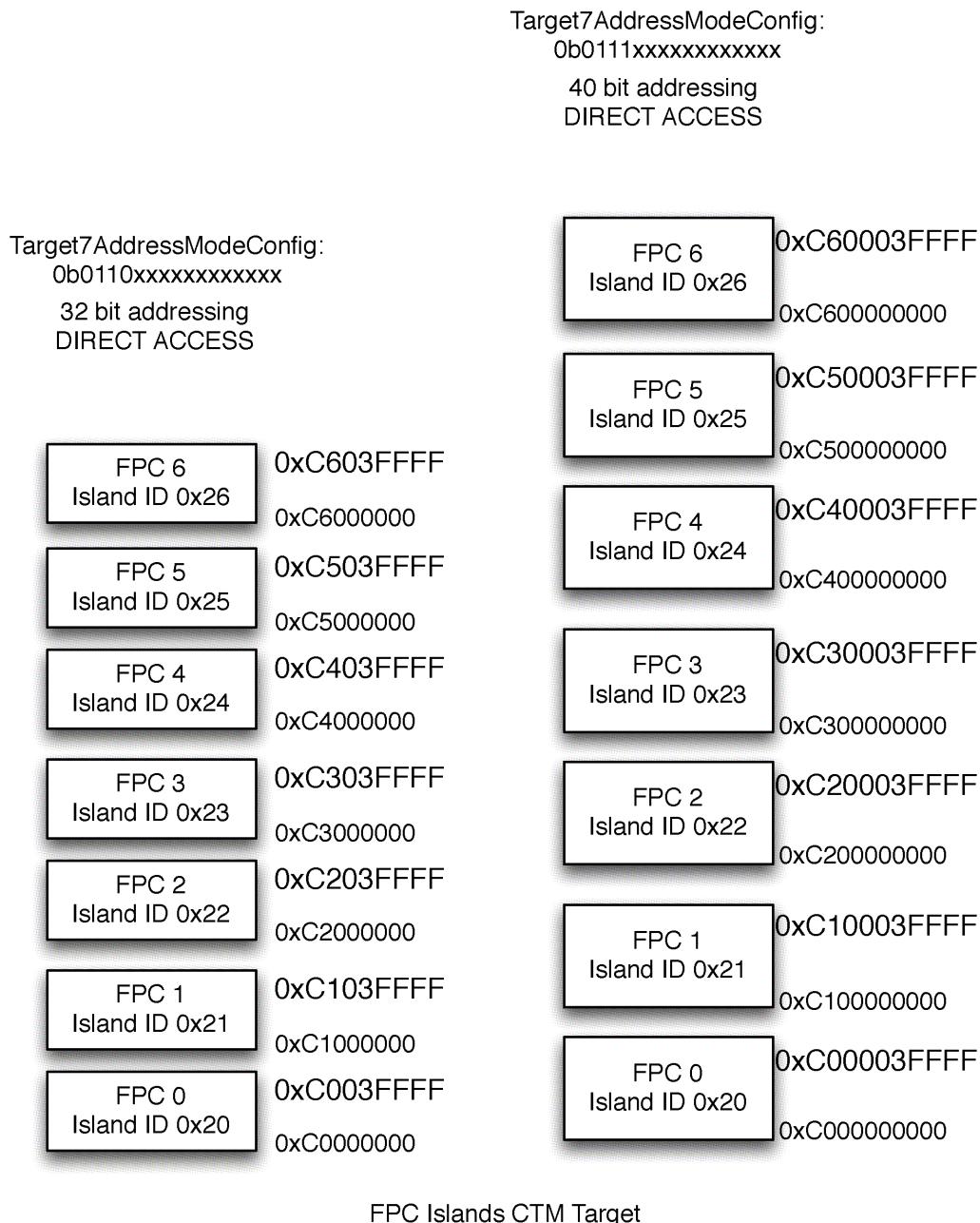
**Figure 3.10. MU Direct Access ARM CTM Memory Map**

### 3.2.8.2 MU Direct Access PCIe CTM Memory Map



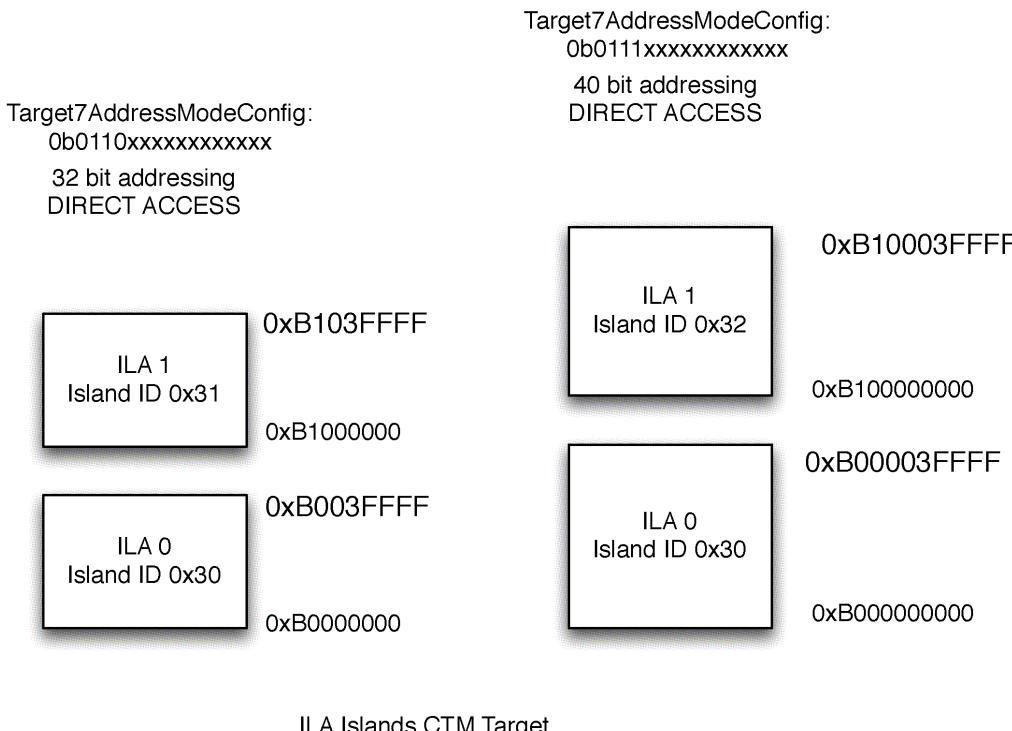
**Figure 3.11. MU Direct Access PCIe CTM Memory Map**

### 3.2.8.3 MU Direct Access FPCx CTM Memory Map



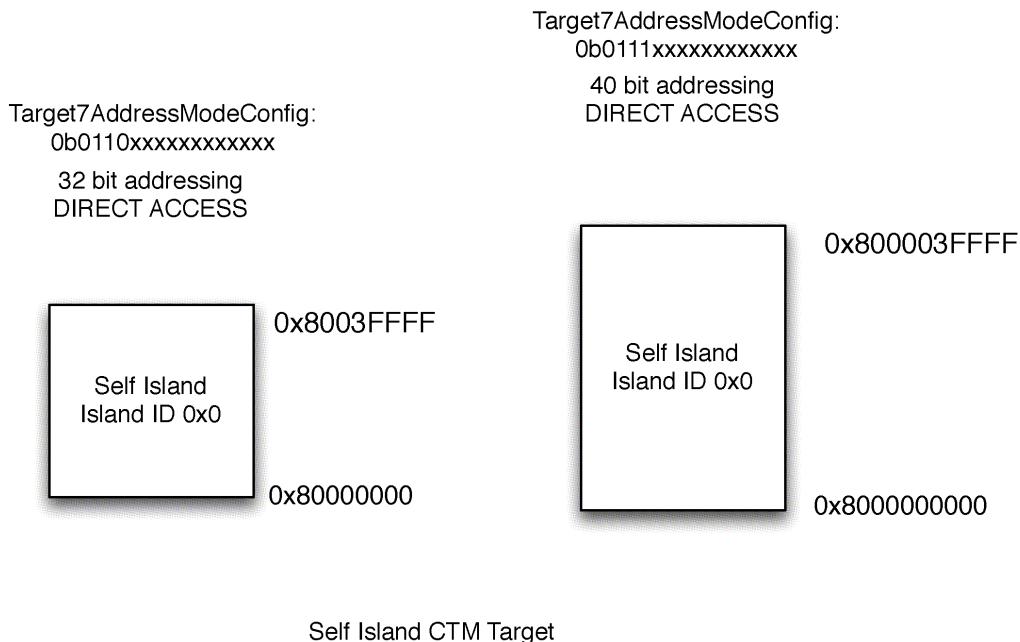
**Figure 3.12. MU Direct Access FPCx CTM Memory Map**

### 3.2.8.4 MU Direct Access ILA CTM Memory Map



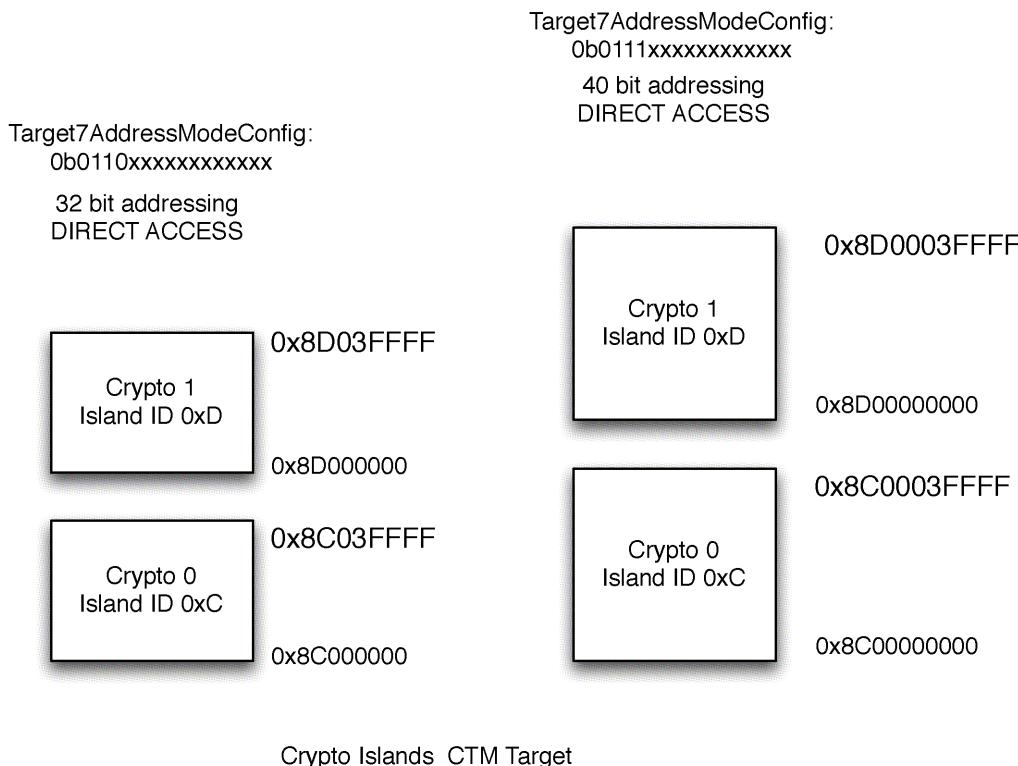
**Figure 3.13. MU Direct Access ILA CTM Memory Map**

### 3.2.8.5 MU Direct Access Self Island CTM Memory Map



**Figure 3.14. MU Direct Access Self Island CTM Memory Map**

### 3.2.8.6 MU Direct Access Crypto CTM Memory Map



**Figure 3.15. MU Direct Access Crypto CTM Memory Map**

### 3.2.9 CLS Direct Access Memory Map

A number of different Island types on the NFP have Cluster Local Scratch as a Target. These are:

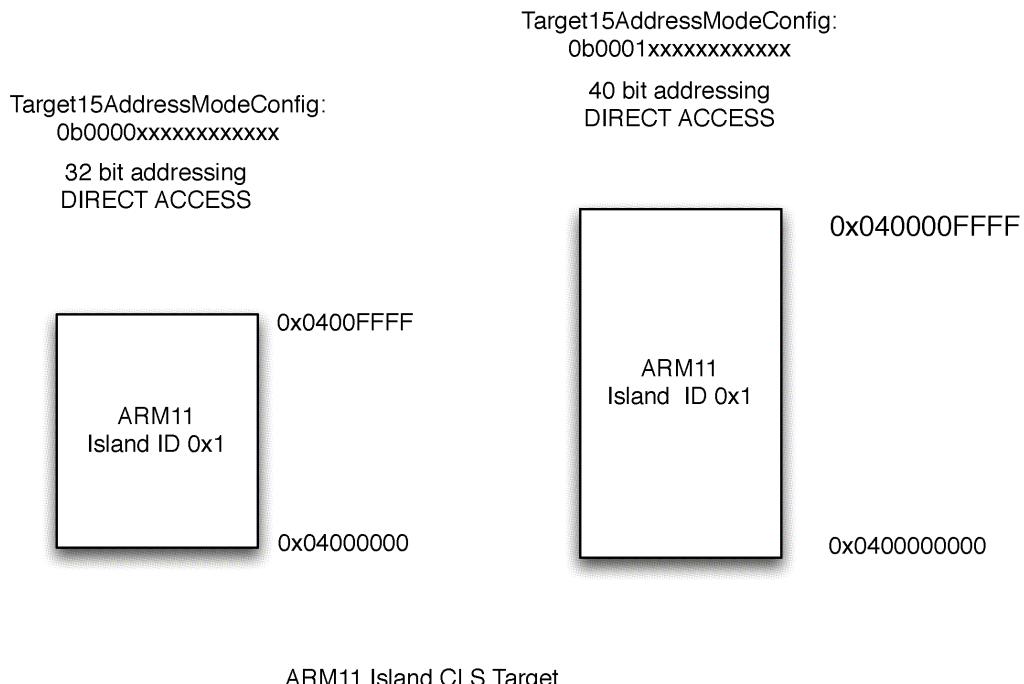
- ARM11 Island
- PCIe Islands
- Crypto Islands
- FPC Islands
- ILA Islands

The addressable memory spaces for each of these Island Family types are shown in the sub-sections [Section 3.2.9.1](#) through [Section 3.2.9.4](#).

It is also possible to reference the Cluster Local Scratch on the Island in which the FPC is located. This is referred to as the “Self Island”. The Self Island will always be addressed using island ID 0x0. The addressable memory space for Self Island addressing is shown in sub-section [Section 3.2.9.5](#).

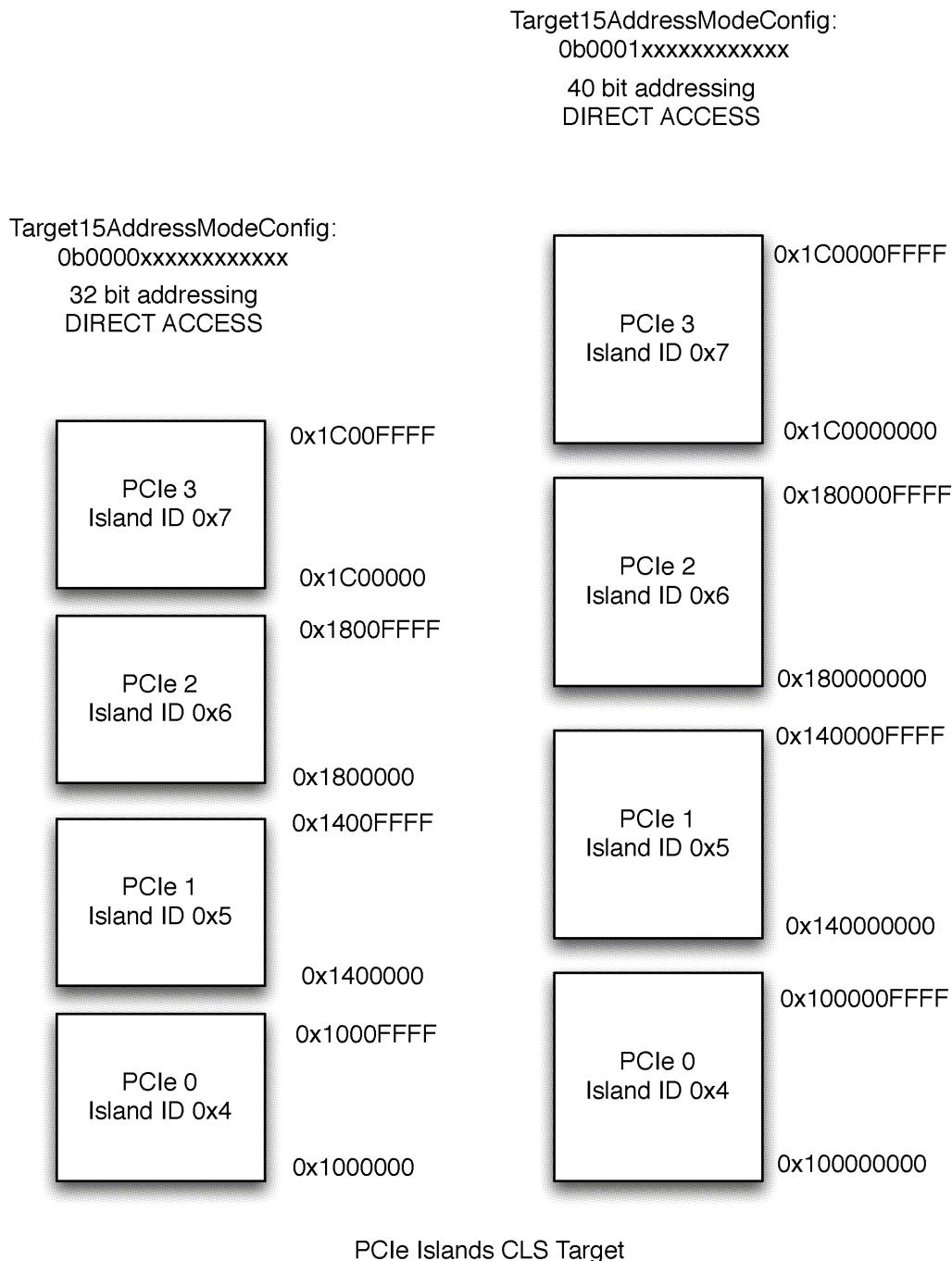
The general CLS Target address ranges for these islands are shown in the figures below. The first 64kB of these ranges are Local Scratch SRAM. CLS related CSRs for functional configuration of HW supported features (Rings, Hash, etc) can be addressed at the addresses shown in the ClusterScratchSSB Address Map and associated maps found in the NFP-6xxx Databook.

### 3.2.9.1 CLS Direct Access ARM Memory Map



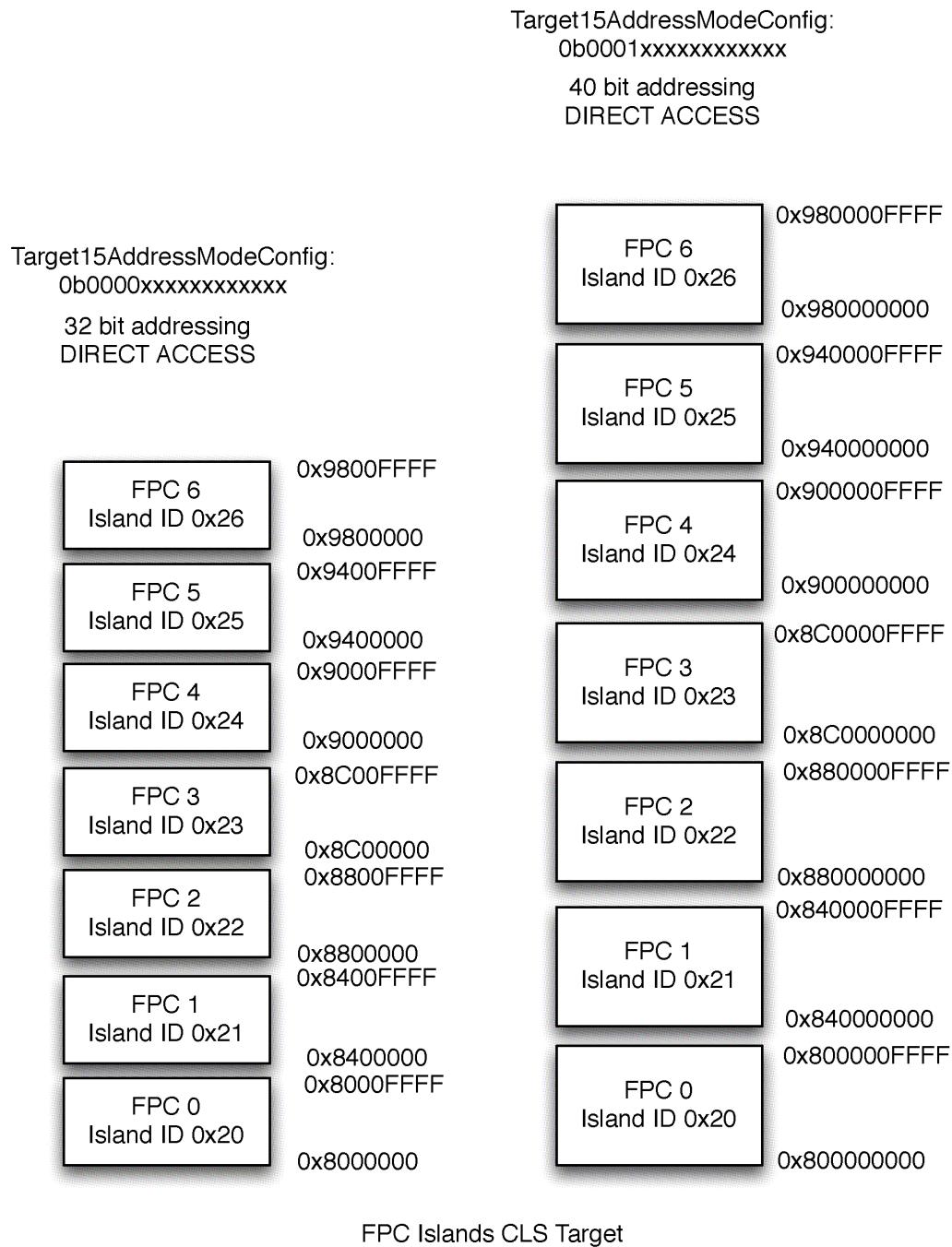
**Figure 3.16. CLS Direct Access ARM Memory Map**

### 3.2.9.2 CLS Direct Access PCIe Memory Map



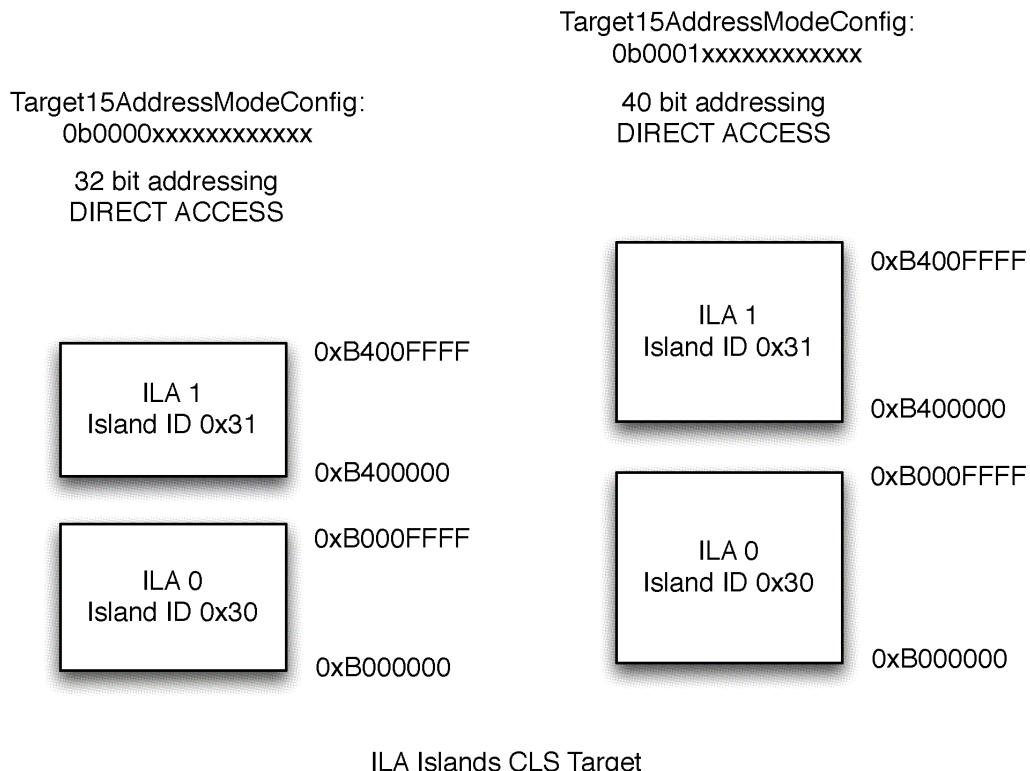
**Figure 3.17. CLS Direct Access PCIe Memory Map**

### 3.2.9.3 CLS Direct Access FPC Memory Map



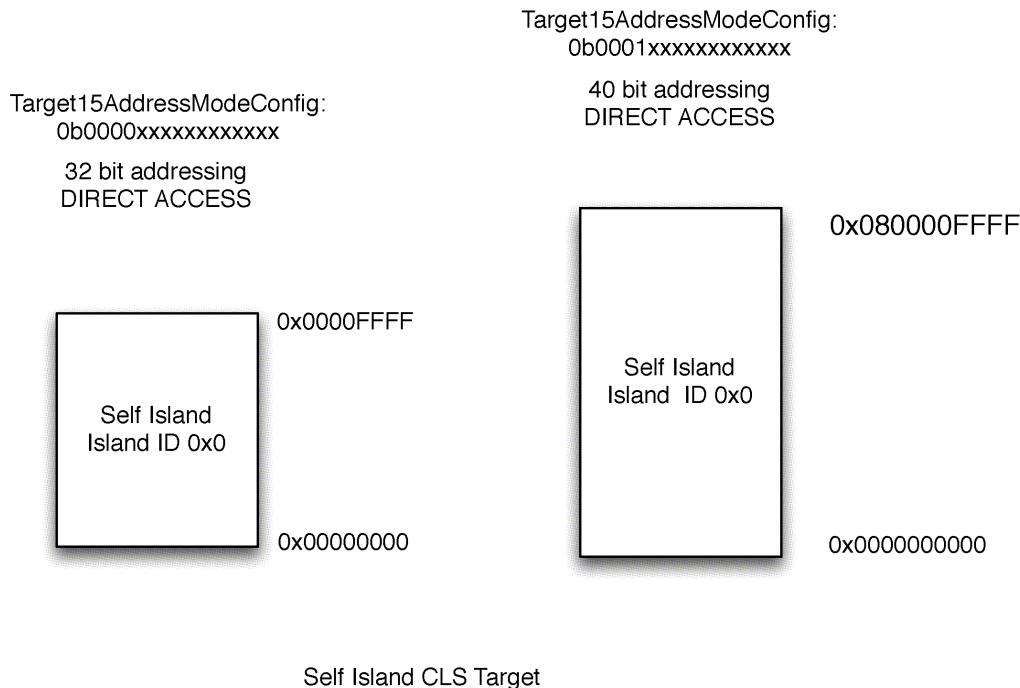
**Figure 3.18. CLS Direct Access FPC Memory Map**

### 3.2.9.4 CLS Direct Access ILA Memory Map



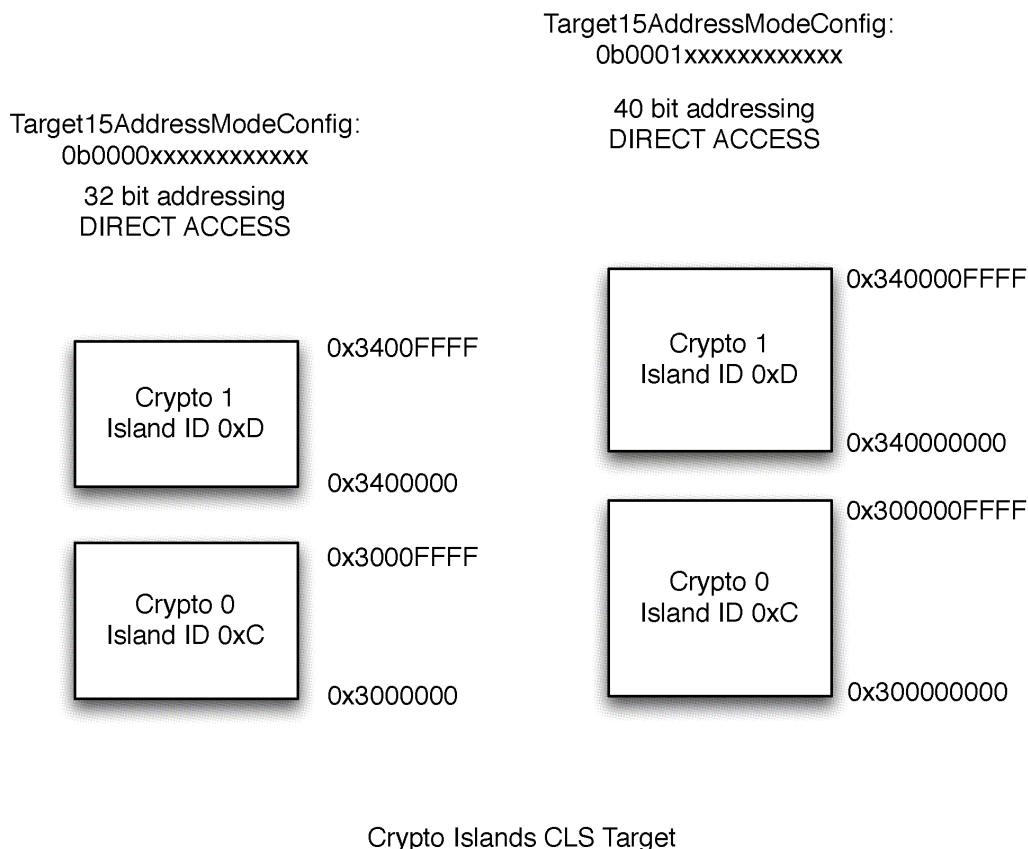
**Figure 3.19. CLS Direct Access ILA Memory Map**

### 3.2.9.5 CLS Direct Access Self Island Memory Map



**Figure 3.20. CLS Direct Access Self Island Memory Map**

### 3.2.9.6 CLS Direct Access Crypto Memory Map



**Figure 3.21. CLS Direct Access Crypto Memory Map**

### 3.2.10 XPB Target Addressing

The schematics presented here for XPB Target Addressing are not memory “maps” in the sense that ranges of usable, addressable memory for a given target on a given island are not portrayed in the same fashion as for the other Target maps. Rather, they represent graphic depictions of how to construct the correct address for accessing any given register over the XPB/Cluster Target bus – whether through the global chip-wide overlay XPBMini bus master or local XPB bus masters.

The method for using the XPB addressing schemes is as follows:

Begin at the left of any one of the addressing schematics for the six different functional categories:

- FPC Inter-Thread Signaling addressing
- FPC CSR access
- FPC Transfer Register access

- FPC Next Neighbor Register access
- Cluster Target Ring access
- Chip Level XPB addressing

This value is the starting base address for a given specific register or memory location to be referenced.

Traversing the appropriate line from that value to the right, towards the correct Island, the value encountered in the next vertical stack of locations is to be summed with the previous base address, forming a new base address. The process continues to the right along the appropriate line towards the desired register grouping, the next value within the subsequent vertical stack is to be summed to the existing base address. This iterative hierarchical approach continues until the exact address of a specific register or memory location is fully constructed.

Some of these schematics do not traverse the entire hierarchy of base locations to be summed, as this would not be practical. Where this is the case, the relevant Table(s) within the NFP-6xxx Databook must be consulted, providing the subsequent value(s) to be added to the existing base address being constructed, for a specific register/memory location.

This hierarchical “construction” of the address for a specific XPB addressed location, together in some sub-modules with address information implicit in the instruction given e.g. signal number, thread number, etc allow for complete, unambiguous address decoding by the NFP assembler and HW support.

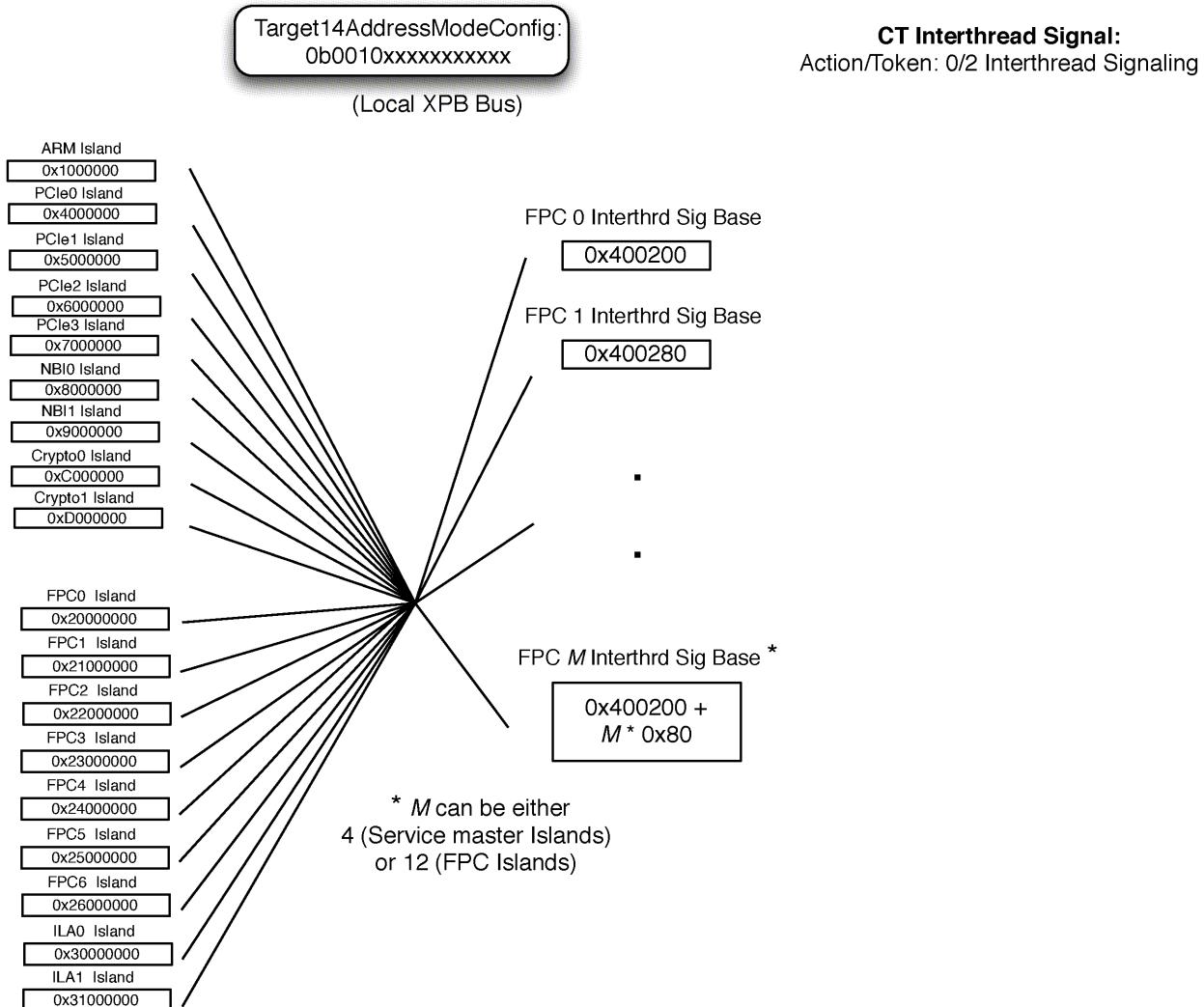
For the XPB Chip Level Addressing scheme it has been logically divided into two overall groupings of Islands: Memory Unit Islands and NON Memory Unit Islands. This is due to the fact that the MU Islands can only be addressed through Global XPBM bus commands. All other Islands can be addressed using either the Global XPBM bus or the local XPB bus, since they possess their own XPB Bus master. Using the local XPB command scheme for these islands would be the typical runtime operational scheme.



### Note

All XPB Target Addressing is a 32-bit value, while all other Target Addressing is a 40-bit value.

### 3.2.10.1 XPB CTM Inter-Thread



**Figure 3.22. XPB CTM Inter-thread Memory Map**

### 3.2.10.2 XPB CTM Reflector CSR

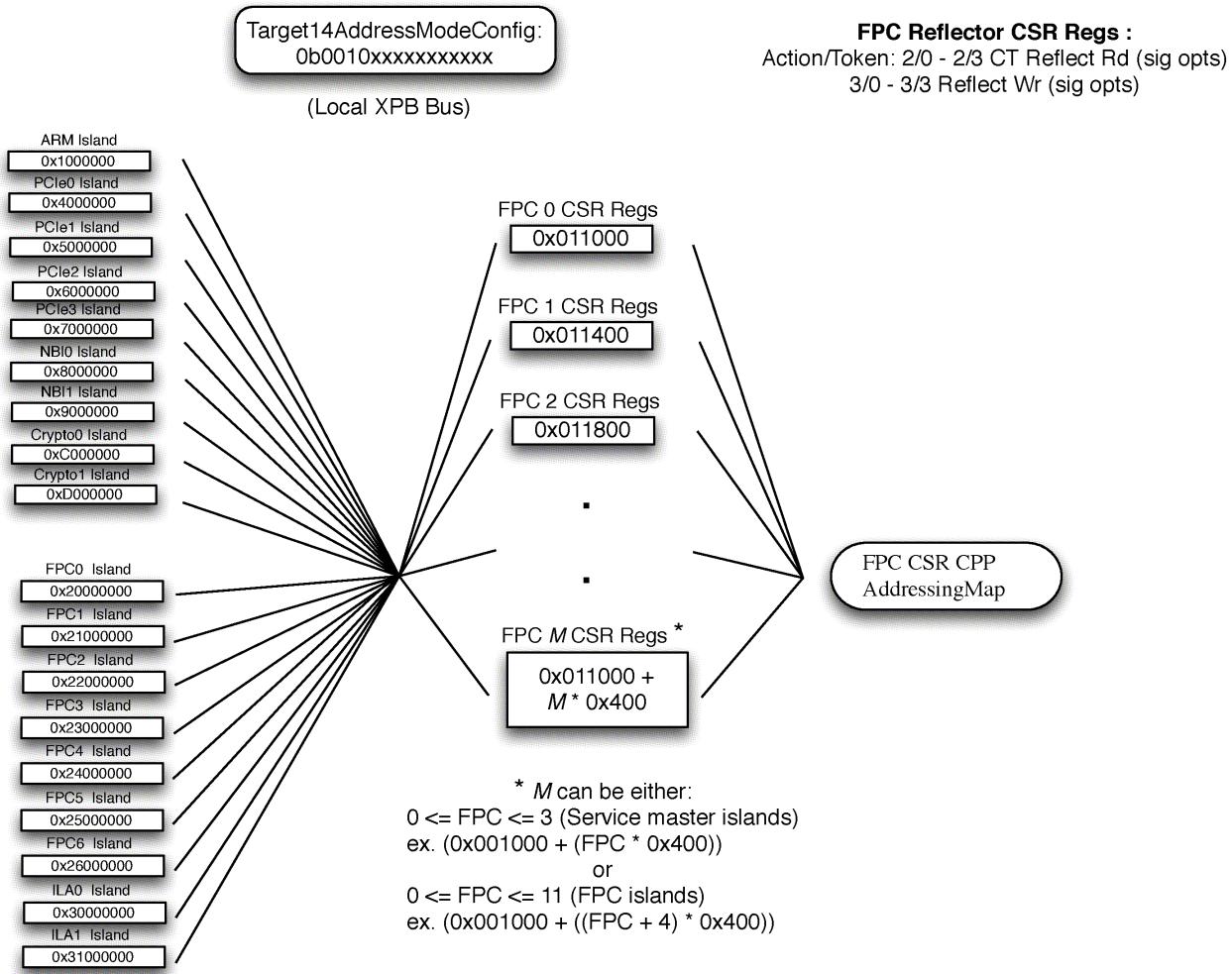


Figure 3.23. XPB CTM Reflector CSR Memory Map

### 3.2.10.3 XPB CTM Reflector XFER

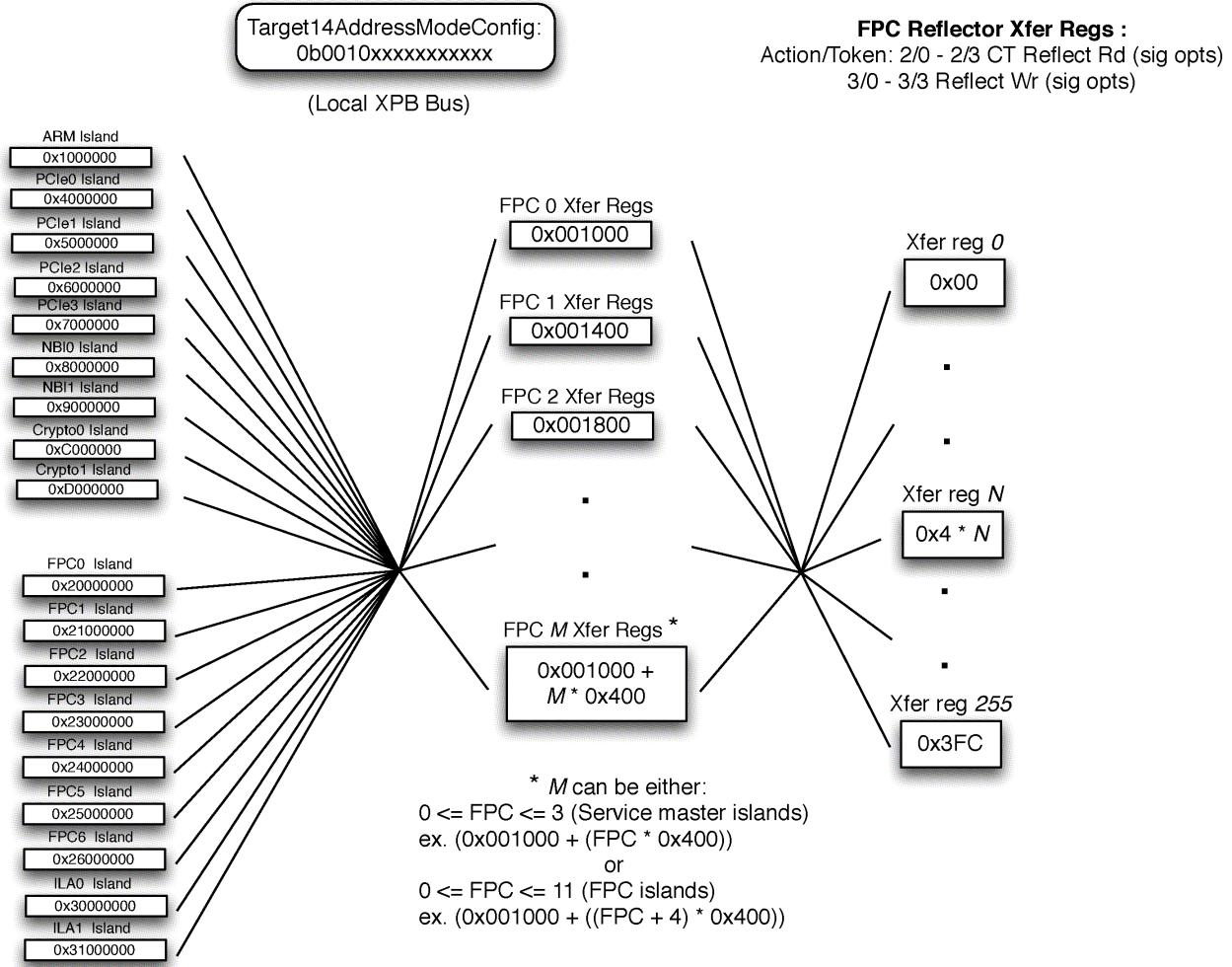


Figure 3.24. XPB CTM Reflector XFER Memory Map

### 3.2.10.4 XPB CTM Next-Neighbor

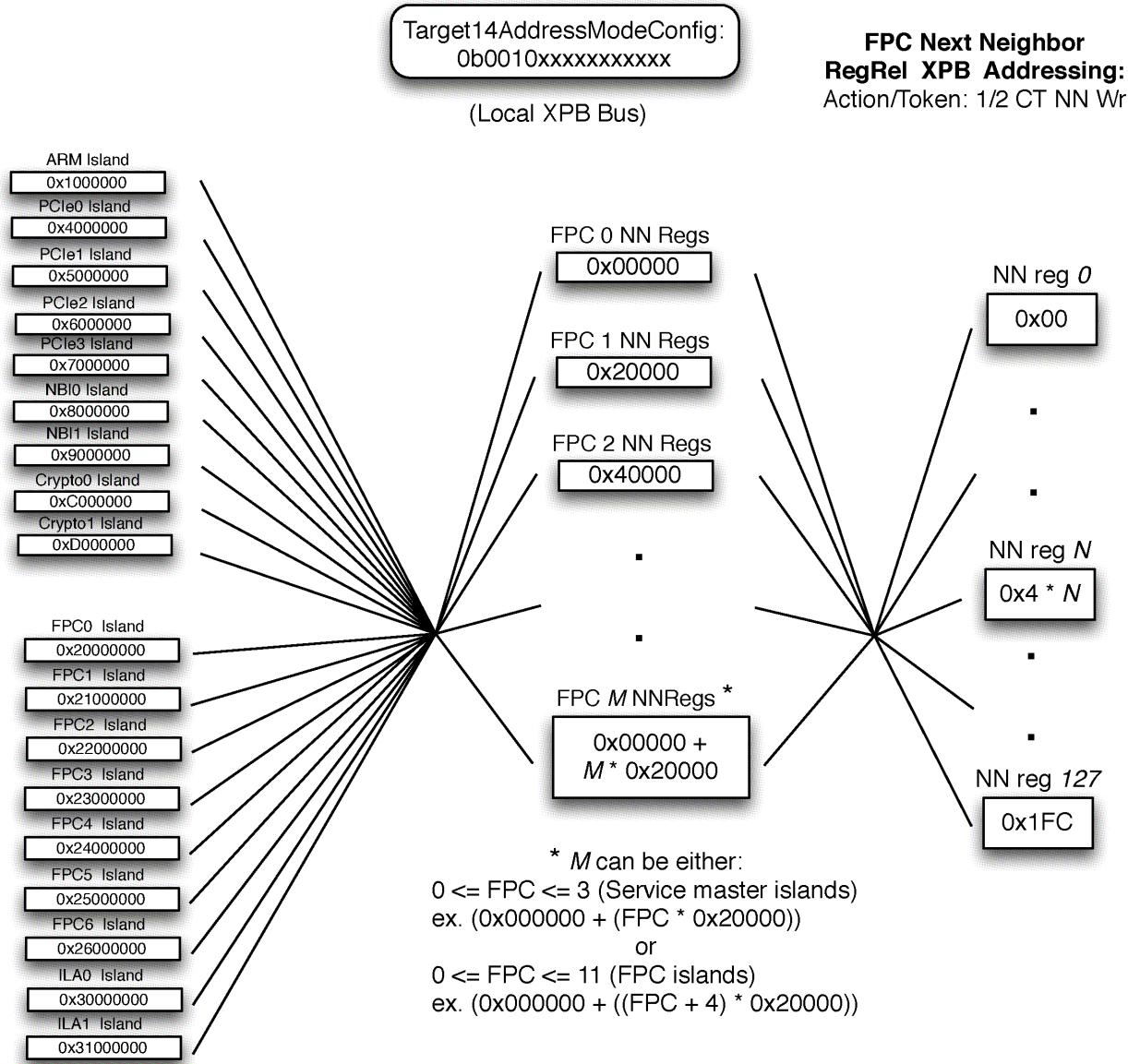


Figure 3.25. XPB CTM Reflector Next-Neighbor Memory Map

### 3.2.10.5 XPB CTM Rings

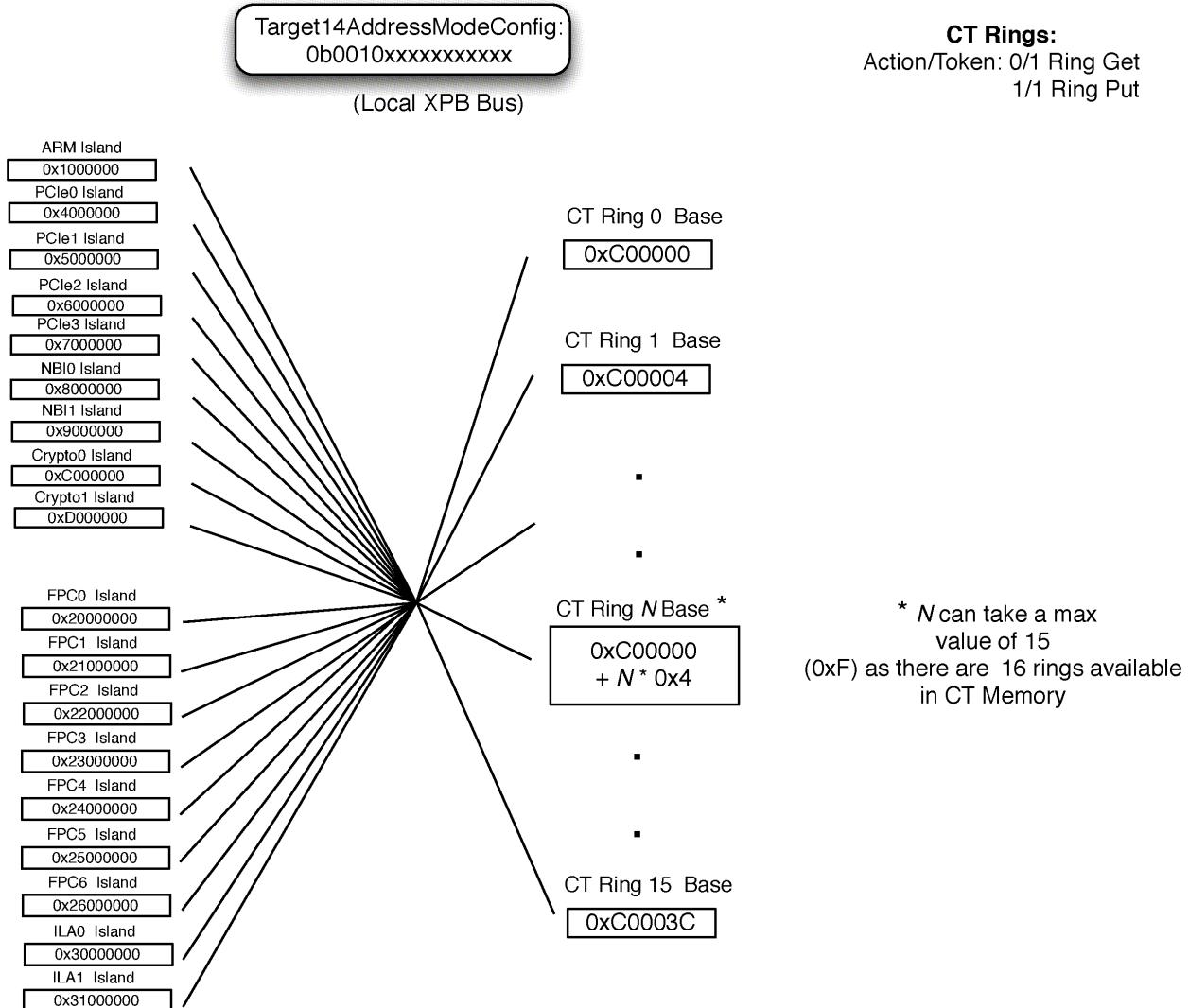
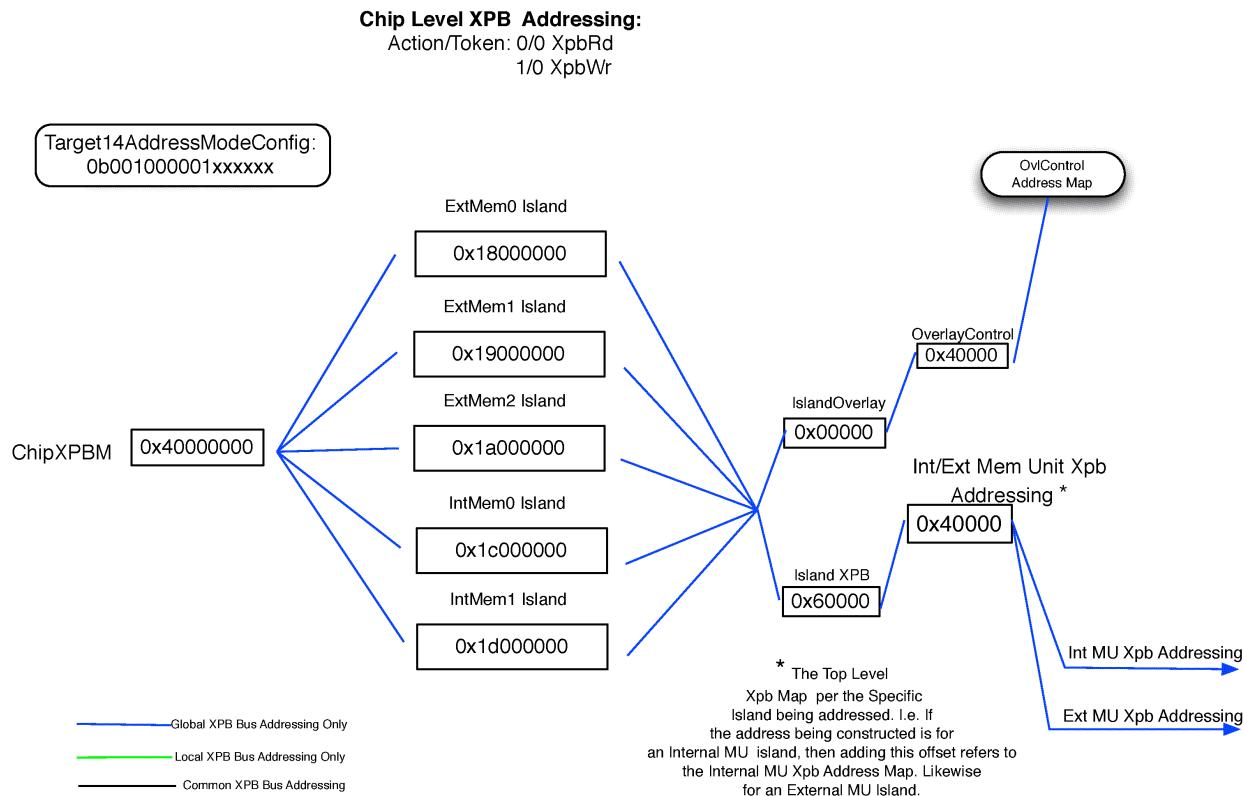


Figure 3.26. XPB CTM Reflector Rings Memory Map

### 3.2.10.6 XPB Chip Level Memory Units

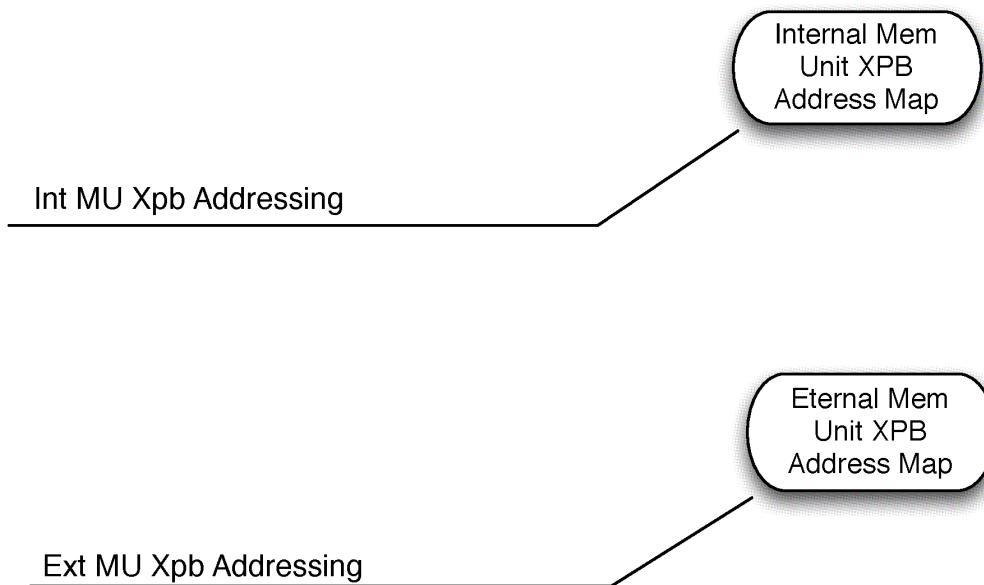


**Figure 3.27. XPB Chip Level Memory Units Only Part 1**

Continued in next figure below.

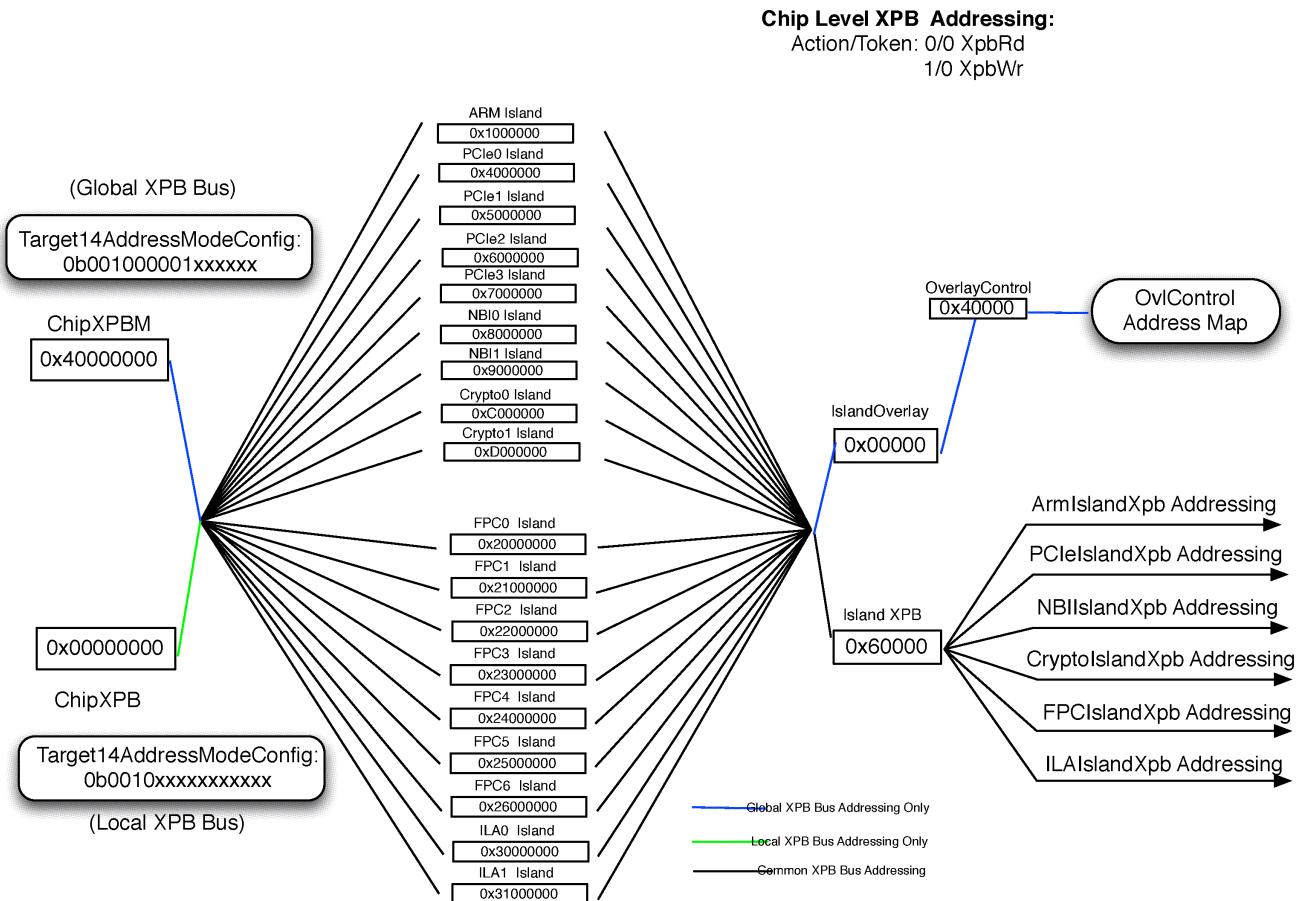
**Chip Level XPB Addressing:**

Action/Token: 0/0 XpbRd  
1/0 XpbWr



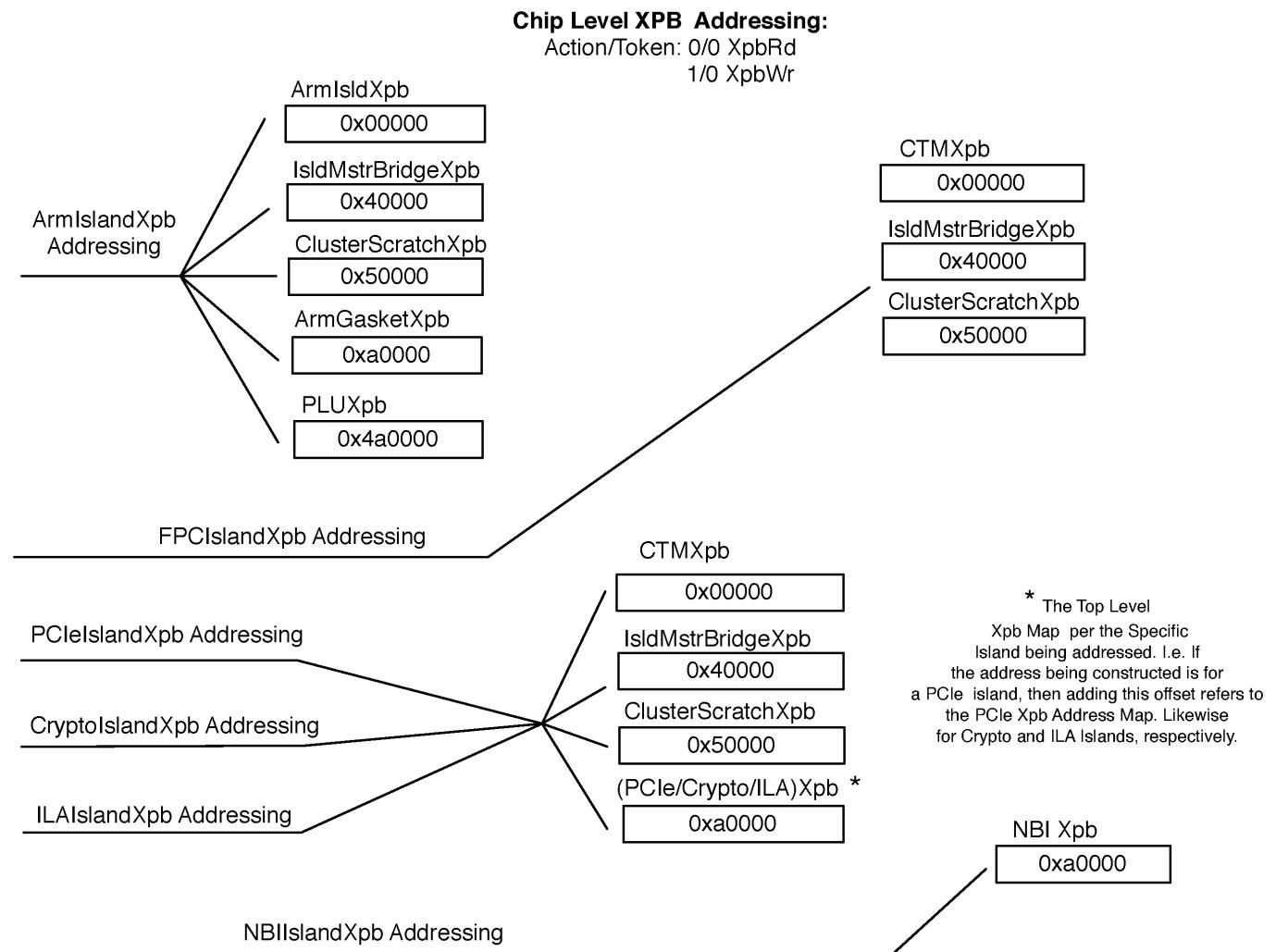
**Figure 3.28. XPB Chip Level Memory Units Only Part 2**

### 3.2.10.7 XPB Chip Level Non Memory Islands



**Figure 3.29. XPB Chip Level Non Memory Islands Only Part 1**

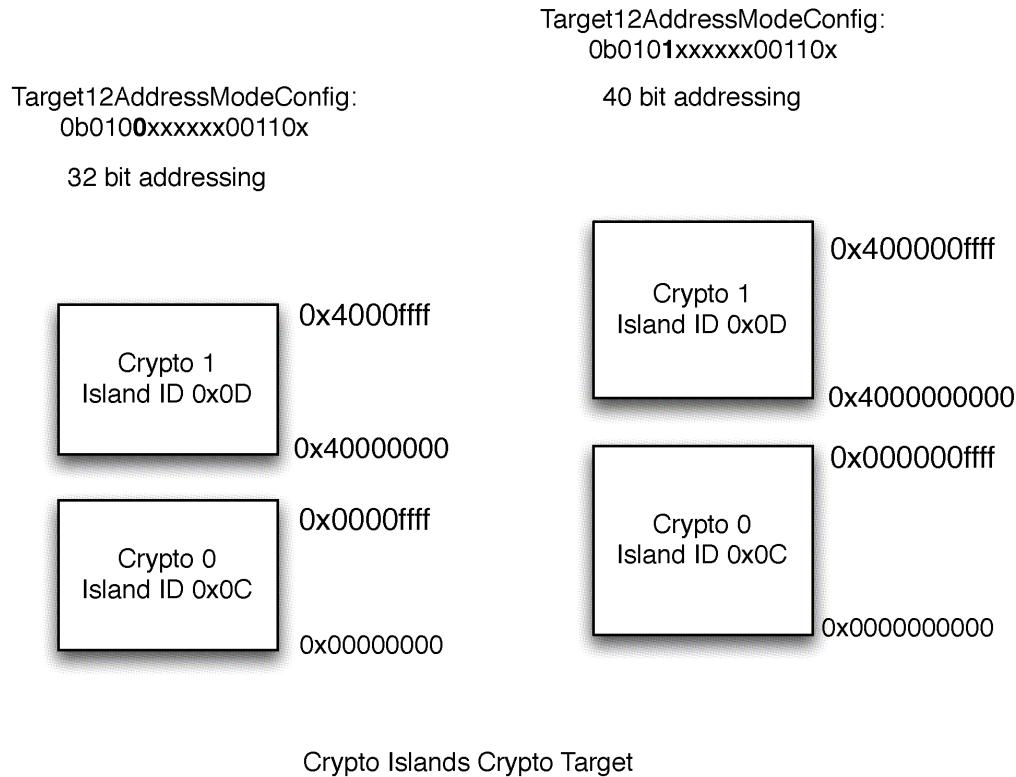
Continued in next figure below.



**Figure 3.30. XPB Chip Level Non Memory Islands Only Part 2**

### 3.2.11 Crypto Memory Map

The Crypto target has 64kB of addressable SRAM, as shown in the figure below.



**Figure 3.31. Crypto Memory Map**

## 4. Control and Status Registers

---

### 4.1 Local Control and Status Registers

Local Control and Status Registers (CSRs) are external to the Execution Datapath, and hold specific purpose information. They can be read and written by special instructions (local\_csr\_rd and local\_csr\_wr) and are typically accessed less frequently than datapath registers. Because Local CSRs are not built in the datapath, there is a write to use delay of either three or four cycles, (depending on the local CSR), and a read to consume penalty of one cycle.

### 4.2 Microengine Local Control and Status Registers

The table below shows the offset addresses of the Microengine Local CSRs. These CSRs can be accessed from the ARM core or another Microengine via reflect transactions. A Microengine can access its own Local CSRs using the local\_csr\_write and local\_csr\_read instructions.

Some local CSRs listed in the table are actually eight registers: one for each Microengine context. These registers are accessed using a common address and an indirection pointer that specifies the context number. The indirection pointer is defined in a field within the CSRCtxPtr CSR (see [Section 4.2.1.1.9](#)

All CSR fields are cleared with the assertion of the T-domain ME Reset input. There are three types of latencies associated with a local\_csr\_write instruction; these latencies vary depending on the actual CSR being targeted by the CSR write instruction. CSR Latencies are shown in [Table 4.1](#).

- Write Latency:** The number of instructions from local\_csr\_wr to when the Local CSR is actually written.
- Read Latency:** The number of instructions between a local\_csr\_wr to a local\_csr\_rd of the same register, to get the newly written value.
- Usage Latency:** The number of instructions between a local\_csr\_wr to when a function is performed by the hardware.

**Table 4.1. Microengine Local Control and Status Register Latencies**

Register Name	CSR Address Offset	Latency		
		Write	Read	Usage
UstorAddr	0x00	NA	NA	NA
UstorDataLwr	0x04	NA	NA	NA
UstorDataUpr	0x08	NA	NA	NA
UstorErrStat	0x0C	NA	NA	NA
ALUOut	0x10	NA	NA	NA
CtxArbCtrl	0x14	4	3	NA
CtxEnables	0x18	4	3	8

Register Name	CSR Address Offset	Latency		
		Write	Read	Usage
CondCodeEn	0x1C	4	3	1
CSRCtxPtr	0x20	3	2	3
PcBreakpoint0	0x24	4	3	NA
PcBreakpoint1	0x28	4	3	NA
PcBreakpointStatus	0x2c	NA	NA	NA
RegErrStatus	0x30	NA	NA	NA
LMErrStatus	0x34	NA	NA	NA
IndCtxStatus	0x40	4	3	NA
ActCtxStatus	0x44	4	3	NA
IndCtxSglEvt	0x48	4	3	NA
ActCtxSglEvt	0x4C	4	3	NA
IndCtxWkpEvt	0x50	4	3	NA
ActCtxWkpEvt	0x54	4	3	NA
IndCtxFtrCnt	0x58	4	3	NA
ActCtxFtrCnt	0x5C	4	3	NA
IndLMAAddr0	0x60	3	3	3
ActLMAAddr0	0x64	3	2	3
IndLMAAddr1	0x68	3	3	3
ActLMAAddr1	0x6C	3	2	3
ByteIndex	0x70	3	2	3
XferIndex	0x74	3	2	3
IndFtrCntSgl	0x78	4	3	NA
ActFtrCntSgl	0x7C	4	3	NA
NNPut	0x80	3	2	0
NNGet	0x84	3	2	3
IndLMAAddr2	0x90	3	3	3
ActLMAAddr2	0x94	3	2	3
IndLMAAddr3	0x98	3	3	3
ActLMAAddr3	0x9C	3	2	3
IndPredCC	0xB0	4	3	5
TimestampLow	0xC0	4	3	NA
TimestampHgh	0xC4	4	3	NA
IndLMAAddr0BytIdx	0xE0	3	3	3
ActLMAAddr0BytIdx	0xE4	3	2	3

Register Name	CSR Address Offset	Latency		
		Write	Read	Usage
IndLMAAddr1BytIdx	0xE8	3	3	3
ActLMAAddr1BytIdx	0xEC	3	2	3
XfrAndBytIdx	0xF4	3	2	3
NxtNghbrSgl	0x100	3	2	12
PrvNghbrSgl	0x104	3	2	12
SameMESignal	0x108	3	2	8
CRCRemainder	0x140	3	2	0
ProfileCnt	0x144	NA	NA	NA
PseudoRndNum	0x148	3	2	3
MiscControl	0x160	3	2	NA
PcBreakpoint0Mask	0x164	NA	NA	NA
PcBreakpoint1Mask	0x168	NA	NA	NA
Mailbox0	0x170	4	3	NA
Mailbox1	0x174	4	3	NA
Mailbox2	0x178	4	3	NA
Mailbox3	0x17C	4	3	NA
CmdIndirectRef0	0x190	3	2	0

#### Notes:

1. CtxEnables: A ctx\_arb instruction should be issued 8 cycles after an instruction is issued to clear the context-enable bits to ensure that the disabled contexts will not run. Note: this latency may change in future versions of the Microengine.
2. CondCodeEn: Must wait 1 cycle before issuing an instruction that updates the Condition Codes.
3. CSRCtxPtr: Must wait 3 cycles before issuing an instruction that uses the CSRCtxPtr CSR.
4. IndLMAAddr0, ActLMAAddr0, IndLMAAddr1, ActLMAAddr1, IndLMAAddr2, ActLMAAddr2, IndLMAAddr3, ActLMAAddr3: Must wait 3 cycles before issuing an instruction that uses one of these LM indices.
5. ByteIndex, ActLMAAddr0BytIdx, IndLMAAddr0BytIdx, IndLMAAddr1BytIdx, ActLMAAddr1BytIdx, ActLMAAddr2BytIdx, IndLMAAddr2BytIdx, ActLMAAddr3BytIdx, IndLMAAddr3BytIdx, XfrAndBytIdx, XferIndex: Must wait 3 cycles before issuing an instruction that uses one of these indices.
6. NNGet: Must wait 3 cycles before issuing an instruction that uses the NNGet CSR.
7. SameMESignal: After issuing a local\_csr\_write instruction targeting the SameMESignal CSR, 8 cycles will elapse before the context arbitration logic sees such write as a signal event setting.
8. CRCRemainder: A usage of 0 indicates that it is permitted to issue a CRC instruction immediately after issuing a local\_csr\_wr instruction targeting the CRCRemainder CSR.
9. MiscControl: This register has control fields which should only be written when ME is in Idle; please see [Section 4.2.1.1.52](#) for usage details on each of the MiscControl fields.

10.PseudoRndNum: When the CtxEnables[PRN\_MODE] bit is set to update on a read, the new PSN will be generated 3 cycles after issuing local\_csr\_write of PSN.

11.

## 4.2.1 Detailed Register Descriptions

The Local ME CSRs materialize on both the CPP and PCIe spaces. [Table 4.2](#) shows the address offsets of each of the ME CSRs.

**Table 4.2. MeCsrCPP Address Map**

Offset from base	Name	Type	Comment
00h	UstorAddr	<a href="#">UstorAddr</a>	Used to load programs into the Control Store
04h	UstorDataLwr	<a href="#">UstorDataLwr</a>	Control Store Data - lower
08h	UstorDataUpr	<a href="#">UstorDataUpr</a>	Control Store Data - upper
0ch	UstorErrStat	<a href="#">UstorErrStat</a>	ECC errors during Control Store reads
10h	ALUOut	<a href="#">ALUOut</a>	Debug to show state of ALU
14h	CtxArbCtrl	<a href="#">CtxArbCtrl</a>	Context Arbiter Control - used by the context arbiter and for debug
18h	CtxEnables	<a href="#">CtxEnables</a>	Context Enables - used by the context arbiter and for debug
1ch	CondCodeEn	<a href="#">CondCodeEn</a>	Condition Code Enable
20h	CSRCtxPtr	<a href="#">CSRCtxPtr</a>	CSR Context Pointer
24h	PcBreakpoint0	<a href="#">PcBreakpoint</a>	PC Breakpoint 0 - PCB system
28h	PcBreakpoint1	<a href="#">PcBreakpoint</a>	PC Breakpoint 1 - PCB system
2ch	PcBreakpointStatus	<a href="#">PcBreakpointStatus</a>	PC Breakpoint - Status register associated with the PCB system
30h	RegErrStatus	<a href="#">RegErrStatus</a>	Information about parity errors detected on Datapath Regs
34h	LMErrStatus	<a href="#">LMErrStatus</a>	Status on ECC errors recorded on Local Memory reads
38h	LMeccErrorMask	<a href="#">LMeccErrorMask</a>	Controls Error Injection bits into an LM data-path word.

Offset from base	Name	Type	Comment
40h	IndCtxStatus	IndCtxStatus	Indirect Context Status Register
44h	ActCtxStatus	ActCtxStatus	Active Context Status Register
48h	IndCtxSglEvt	CtxSglEvt	Indirect Context Signal Events Register
4ch	ActCtxSglEvt	CtxSglEvt	Active Context Signal Events Register
50h	IndCtxWkpEvt	CtxWkpEvt	Indirect Context Wakeup Events Register
54h	ActCtxWkpEvt	CtxWkpEvt	Active Context Wakeup Events Register
58h	IndCtxFtrCnt	CtxFtrCnt	Indirect Context Future Count Register
5ch	ActCtxFtrCnt	CtxFtrCnt	Active Context Future Count Register
60h	IndLMAAddr0	LMAddr	Indirect Local Memory Address 0 Register
64h	ActLMAAddr0	LMAddr	Active Local Memory Address 0 Register
68h	IndLMAAddr1	LMAddr	Indirect Local Memory Address 1 Register
6ch	ActLMAAddr1	LMAddr	Active Local Memory Address 1 Register
70h	ByteIndex	ByteIndex	Byte Index Register
74h	XferIndex	XferIndex	Transfer Index Register
78h	IndFtrCntSgl	FtrCntSgl	Which signal to set when FUTURE_COUNT == TIMESTAMP
7ch	ActFtrCntSgl	FtrCntSgl	Which signal to set when FUTURE_COUNT == TIMESTAMP
80h	NNPut	NNPutGet	Next Neighbor Put Register
84h	NNGet	NNPutGet	Next Neighbor Get Register
90h	IndLMAAddr2	LMAddr	Indirect Local Memory Address 2 Register
94h	ActLMAAddr2	LMAddr	Active Local Memory Address 2 Register

Offset from base	Name	Type	Comment
98h	IndLMAAddr3	LMAAddr	Indirect Local Memory Address 3 Register
9ch	ActLMAAddr3	LMAAddr	Active Local Memory Address 3 Register
a0h	IndLMAAddr2BytIdx	LMAAddrBytIdx	Alias of IndLMAAddr2 and ByteIndex
a4h	ActLMAAddr2BytIdx	LMAAddrBytIdx	Alias of ActLMAAddr2 and ByteIndex
a8h	IndLMAAddr3BytIdx	LMAAddrBytIdx	Alias of IndLMAAddr3 and ByteIndex
ach	ActLMAAddr3BytIdx	LMAAddrBytIdx	Alias of ActLMAAddr3 and ByteIndex
b0h	IndPredCC	IndPredCC	Indirect Predicate CC select
c0h	TimestampLow	TimestampLow	Timestamp is 64 bits. It counts up by one every sixteen cycles
c4h	TimestampHgh	TimestampHgh	Timestamp is 64 bits. It counts up by one every sixteen cycles
e0h	IndLMAAddr0BytIdx	LMAAddrBytIdx	Alias of IndLMAAddr0 and ByteIndex
e4h	ActLMAAddr0BytIdx	LMAAddrBytIdx	Alias of ActLMAAddr0 and ByteIndex
e8h	IndLMAAddr1BytIdx	LMAAddrBytIdx	Alias of IndLMAAddr1 and ByteIndex
ech	ActLMAAddr1BytIdx	LMAAddrBytIdx	Alias of ActLMAAddr1 and ByteIndex
f4h	XfrAndBytIdx	XfrAndBytIdx	This register is used when Transfer registers are accessed via indexed mode
100h	NxtNghbrSgl	NxtNghbrSgl	Signal a Context in Next Neighbor
104h	PrvNghbrSgl	PrvNghbrSgl	Signal a Context in Previous Neighbor
108h	SameMESignal	SameMESignal	Signal another Context in same Microengine.
140h	CRCRemainder	CRCRemainder	Result of the CRC operation after a crc instruction.
144h	ProfileCnt	ProfileCnt	The profile count is used for code profiling and tuning

Offset from base	Name	Type	Comment
148h	PseudoRndNum	PseudoRndNum	Random number generator
160h	MiscControl	MiscControl	Miscellaneous Control Register
164h	PcBreakpoint0Mask	PcBreakpointMask	Mask register associated with PC Breakpoint 0
168h	PcBreakpoint1Mask	PcBreakpointMask	Mask register associated with PC Breakpoint 1
170h	Mailbox0	Mailbox	Mailbox Register 0
174h	Mailbox1	Mailbox	Mailbox Register 1
178h	Mailbox2	Mailbox	Mailbox Register 2
17ch	Mailbox3	Mailbox	Mailbox Register 3
190h	CmdIndirectRef0	CmdIndirectRef0	Command Indirect Reference Register Type 0

## 4.2.1.1 ME - Local CSRs - Detailed Description

### 4.2.1.1.1 Offset 00h: UstorAddr - Control Store Data Register

Assembler Register Name: USTORE\_ADDRESS

Used to load programs into the Control Store. This register and the [UstorDataUp](#) and [UstorDataLwr](#) registers are used to program the Microengine while all Contexts are in Inactive state. An external source (such as the ARM core) writes the Control Store address to this register and follows it with reads or writes to the [UstorDataUp](#) and [UstorDataLwr](#) registers.

After a write to this register, the data from the Control Store at the UADR location can be read by reading the [UstorDataUp](#) and [UstorDataLwr](#) registers.

To write the Control Store do the following (Note that the Microengine reset signal must be deasserted before this):

1. Write the address to be written into UstorAddr[Uadr]. Note that the EnableCS bit must be a '1' to write the Control Store.
2. Write the data for bits [31:0] into [UstorDataLwr](#).
3. Write the data for bits [39:32] into [UstorDataUp](#). The write to [UstorDataUp](#) also causes the write into the Control Store, and UstorAddr[Uaddr] to increment.
4. If writing consecutive addresses of the Ustore, repeat steps 2 and 3 for each location to be loaded. If writing to non-consecutive locations, repeat steps 1 through 3 for each to be loaded.
5. Write the UstorAddr[EnableCS] to '0' to enter normal mode.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EnableCS	UstoreWrStrb	UstoreDataInvert	reserved																								Uaddr				

Bits	Width; Start	Field name	Access	Reset value	Description				
31	1;31	EnableCS	rw	0	<p>Enable Control Store: The Microengine should be in Idle state (no contexts running). The address in Uaddr field specifies the Control Store address where the data written to UstorDataLwr and UstorDataUpr will be written. Also set in debug mode. This bit can be used to dump data from Microengine GPRs and Read Transfer registers. The Microengine should be in an idle state (no contexts running). This forces the Microengine to continuously execute an instruction at the address specified by Uaddr. Only the ALU instruction is supported in this mode and the result of the execution is written to ALUOut CSR rather than a destination register.</p> <table border="1"> <tr> <td>0x0</td><td>Cleared during normal execution.</td></tr> <tr> <td>0x1</td><td>Set when reading or writing the control store.</td></tr> </table>	0x0	Cleared during normal execution.	0x1	Set when reading or writing the control store.
0x0	Cleared during normal execution.								
0x1	Set when reading or writing the control store.								
30	1;30	UstoreWrStrb	rw	0	<p>UstoreWriteStrobe</p> <table border="1"> <tr> <td>0x0</td><td>Normal use</td></tr> <tr> <td>0x1</td><td>Each time the UstorAddr CSR is written with this bit set to 1, the ME will write the contents of the Ustore Data CSR's into the Control Store address specified by the UADR field. This bit is for Netronome use only.</td></tr> </table>	0x0	Normal use	0x1	Each time the UstorAddr CSR is written with this bit set to 1, the ME will write the contents of the Ustore Data CSR's into the Control Store address specified by the UADR field. This bit is for Netronome use only.
0x0	Normal use								
0x1	Each time the UstorAddr CSR is written with this bit set to 1, the ME will write the contents of the Ustore Data CSR's into the Control Store address specified by the UADR field. This bit is for Netronome use only.								
29	1;29	UstoreDataInvert	rw	0	<p>UstoreDataInvert</p> <table border="1"> <tr> <td>0x0</td><td>Normal use</td></tr> <tr> <td>0x1</td><td>When 1, the Data value written to the Ustore will be the logical inverse of the data in the Ustore DATA CSR's. This functionality can be used in conjunction with the Ustore Write Strobe feature described in bit[30]. This bit is for Netronome use only.</td></tr> </table>	0x0	Normal use	0x1	When 1, the Data value written to the Ustore will be the logical inverse of the data in the Ustore DATA CSR's. This functionality can be used in conjunction with the Ustore Write Strobe feature described in bit[30]. This bit is for Netronome use only.
0x0	Normal use								
0x1	When 1, the Data value written to the Ustore will be the logical inverse of the data in the Ustore DATA CSR's. This functionality can be used in conjunction with the Ustore Write Strobe feature described in bit[30]. This bit is for Netronome use only.								
28:13	16;13	Reserved	-	-	Reserved				

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
12:0	13:0	Uaddr	rw	-	Address of control store location to be accessed. Valid values are 0 to 8191.

#### 4.2.1.1.2 Offset 04h: UstorDataLwr - Control Store Data Lower Register

Assembler Register Name: USTORE\_DATA\_LOWER

Control Store Data. These registers (along with UstorAddr) are used to program the Microengine's Control Store while the Microengine is in the idle state.

Writing the Control Store address to UstorAddr CSR and following it with writes to these CSRs loads an instruction (the write to UstorDataUp actually causes the write, and causes UstorAddr to increment).

Reading these registers after writing UstorAddr reads the data stored in that Control Store address.

See [Section 4.2.1.1.3](#) for details on the UstorDataUp Register.

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

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
31:0	32:0	UdataLower	rw	-	Contains bits 31:00 of the instruction of the Control Store location specified by the UstorAddr CSR.

#### 4.2.1.1.3 Offset 08h: UstorDataUp - Control Store Data Upper Register

Assembler Register Name: USTORE\_DATA\_UPPER

Control Store Data. These registers (along with UstorAddr) are used to program the Microengine's Control Store while the Microengine is in the idle state. A write to this CSR causes an auto-increment of the Ustor\_Adr CSR; on the other hand, a read of this CSR does not cause an auto-increment of the Ustor\_Adr CSR.

Some of the bits in UstorDataUp hold ECC information, as shown in the table below. The Control Store logic does not compute the ECC bits. Therefore, if ECC checking is to be used, the correct ECC bits must be precomputed before programming the control store.

See [Section 4.2.1.1.2](#) for details on the on the UstorDataLwr Register.

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

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
31:20	12:20	Reserved	-	-	Reserved

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
19:13	7:13	ECC	rw	-	Contains the ECC Check bits for the instruction
12:0	13:0	UdataUpper	rw	-	Contains the data from bits [43:32] of the control store location specified by the UstorAddr.

#### 4.2.1.1.4 Offset 0Ch: UstorErrStat - Control Store Error Status Register

Assembler Register Name: USTORE\_ERROR\_STATUS

This status register captures information about ECC errors detected on Control Store reads. The contents of this CSR are valid and frozen while CtxEnables[CS\_ECCError] remains a '1'. Clearing CtxEnables[CS\_ECCError] re-enables this register so that information on the next CS ECC error may be captured.



#### Note

In this context, "frozen" means no updates to this register is possible; thus, information pertaining to a second correctable error (or uncorrectable error) would be lost while the "frozen" state remains in effect. However, other ECC error activities would occur as normal; that is, a correctable error would still be corrected, and an uncorrectable error would still halt the ME.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UncorrectableErr	reserved				Syndrome				reserved	Context				reserved	Uaddr																

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
31	1:31	UncorrectableErr	ro	0	Error Type. This bit indicates the type of error detected.  <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Correctable Error</td></tr> <tr> <td>0x1</td><td>Uncorrectable Error. ME will halt when this bit is set.</td></tr> </table>	0x0	Correctable Error	0x1	Uncorrectable Error. ME will halt when this bit is set.
0x0	Correctable Error								
0x1	Uncorrectable Error. ME will halt when this bit is set.								
30:27	4:27	Reserved	-	-	Reserved				
26:20	7:20	Syndrome	ro	-	This field records the syndrome that was found when the error occurred.				
19	1:19	Reserved	-	-	Reserved				
18:16	3:16	Context	ro	-	Context that was executing when the ECC error occurred				
15:14	2:14	Reserved	-	-	Reserved				

Bits	Width; Start	Field name	Access	Reset value	Description
13:0	14;0	Uaddr	ro	-	Contains the address that had the ECC error.

#### 4.2.1.1.5 Offset 10h: ALUOut - Arithmetic Logic Unit Out Register

Assembler Register Name: ALU\_OUT

This CSR can be used during debugging to read the contents of the Microengine bank A and B GPRs or Transfer In Registers. It can be read by an external Master (such as the ARM core or PCIe host) to view the current state of the ALU.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32;0	ALUOutput	ro	-	ALU output.

#### 4.2.1.1.6 Offset 14h: CtxArbCtrl - Context Arbiter Control Register

Assembler Register Name: CTX\_ARB\_CNTL

This register is used by the context arbiter and is also used for debugging.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:7	25;7	Reserved	-	-	Reserved
6:4	3;4	PreviousCtx	ro	-	Previous Context. This field contains the number of the last context that was running.
3	1;3	Reserved	-	-	Reserved
2:0	3;0	NextCtx	rw	-	Next Context. This field contains the number of the next context that will be run.

#### 4.2.1.1.7 Offset 18h: CtxEnables - Context Enables Register

Assembler Register Name: CTX\_ENABLES

This register is used primarily for enabling the ME contexts, but it also contains some debugging and status fields.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
InUseContexts	CSEccError	CSEccEnable	reserved	Breakpoint	reserved	RegisterParityErr	reserved	LMAdd3Global	LMAdd2Global	LMAdd1Global	NNsendConfig	EmptyAssert	NextNeighbor	LMAddr1Global	LMAddr0Global														NNreceiveConfig		

Bits	Width; Start	Field name	Access	Reset value	Description				
31	1:31	InUseContexts	rw	-	<p>Indicates the number of in-use contexts, which determines the GPR and Transfer Register allocation. Note that although this information could be inferred from bits C0 to C7, this field allows for contexts to be temporarily disabled due to error or debugging conditions. It is illegal to enable Contexts that are not currently in-use according to this field.</p> <table border="1"> <tr> <td>0x0</td><td>8 Context mode (Contexts 0, 1, 2, 3, 4, 5, 6, 7)</td></tr> <tr> <td>0x1</td><td>4 Context mode (Contexts 0, 2, 4, 6)</td></tr> </table>	0x0	8 Context mode (Contexts 0, 1, 2, 3, 4, 5, 6, 7)	0x1	4 Context mode (Contexts 0, 2, 4, 6)
0x0	8 Context mode (Contexts 0, 1, 2, 3, 4, 5, 6, 7)								
0x1	4 Context mode (Contexts 0, 2, 4, 6)								
30	1:30	PseudoRandNum	rw	-	<p>Controls when the Pseudo_Random_Number is generated.</p> <table border="1"> <tr> <td>0x0</td><td>PseudoRndNum is updated only when it is loaded or read using the local_CSR instruction (not when the IA core reads the local CSR)</td></tr> <tr> <td>0x1</td><td>PseudoRndNum is updated every cycle.</td></tr> </table>	0x0	PseudoRndNum is updated only when it is loaded or read using the local_CSR instruction (not when the IA core reads the local CSR)	0x1	PseudoRndNum is updated every cycle.
0x0	PseudoRndNum is updated only when it is loaded or read using the local_CSR instruction (not when the IA core reads the local CSR)								
0x1	PseudoRndNum is updated every cycle.								
29	1:29	CSEccError	rw1c	0	<p>Indicates that an ECC error was detected in the Control Store when an instruction was read. This bit will never be set if ECCErrorEnable bit is 0. When this bit is set the Microengine's attn output is asserted. More information about the error is available in UstorErrStat Register.</p>				
28	1:28	CSEccEnable	rw	0	<p>Enables ECC error detection on Control Store.</p> <table border="1"> <tr> <td>0x0</td><td>ECC Error checking is disabled, ECC errors are not detected and not reported.</td></tr> <tr> <td>0x1</td><td>Error checking is enabled.</td></tr> </table>	0x0	ECC Error checking is disabled, ECC errors are not detected and not reported.	0x1	Error checking is enabled.
0x0	ECC Error checking is disabled, ECC errors are not detected and not reported.								
0x1	Error checking is enabled.								
27	1:27	Breakpoint	rw1c	0	<p>The ctx_arb[bpt] instruction was executed. When this bit is set, the Microengine's attn output is asserted and all CtxEnables bits in this register are cleared.</p>				
26	1:26	Reserved	-	-	Reserved				
25	1:25	RegisterParityErr	rw1c	0	<p>Indicates that a parity error was detected when reading a datapath register. When this bit is set, the Microengine's</p>				

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
					<p>attn output is asserted and all CtxEnables bits in this register are cleared. RegErrStatus Local CSR has information about the error.</p> <table border="1"> <tr> <td>0x0</td><td>No Error detected since this bit was last cleared.</td></tr> <tr> <td>0x1</td><td>Parity error was detected.</td></tr> </table>	0x0	No Error detected since this bit was last cleared.	0x1	Parity error was detected.
0x0	No Error detected since this bit was last cleared.								
0x1	Parity error was detected.								
24	1;24	Reserved	-	-	Reserved				
23	1;23	LMEccEnable	rw	0	<p>Enables ECC error detection on Local Memory.</p> <table border="1"> <tr> <td>0x0</td><td>ECC Error checking is disabled; ECC errors are not detected and not reported.</td></tr> <tr> <td>0x1</td><td>Error checking is enabled.</td></tr> </table>	0x0	ECC Error checking is disabled; ECC errors are not detected and not reported.	0x1	Error checking is enabled.
0x0	ECC Error checking is disabled; ECC errors are not detected and not reported.								
0x1	Error checking is enabled.								
22	1;22	LMAddr3Global	rw	-	<p>Controls usage of ActLMAddr3.</p> <table border="1"> <tr> <td>0x0</td><td>ActLMAddr3 is Context Relative. Each Context uses a separate copy of ActLMAddr3, stored in IndLMAddr3.</td></tr> <tr> <td>0x1</td><td>ActLMAddr3 is Global. Only the working copy of ActLMAddr3 is used, independent of Active Context.</td></tr> </table>	0x0	ActLMAddr3 is Context Relative. Each Context uses a separate copy of ActLMAddr3, stored in IndLMAddr3.	0x1	ActLMAddr3 is Global. Only the working copy of ActLMAddr3 is used, independent of Active Context.
0x0	ActLMAddr3 is Context Relative. Each Context uses a separate copy of ActLMAddr3, stored in IndLMAddr3.								
0x1	ActLMAddr3 is Global. Only the working copy of ActLMAddr3 is used, independent of Active Context.								
21	1;21	LMAddr2Global	rw	-	<p>Controls usage of ActLMAddr2.</p> <table border="1"> <tr> <td>0x0</td><td>ActLMAddr2 is Context Relative. Each Context uses a separate copy of ActLMAddr2, stored in IndLMAddr2.</td></tr> <tr> <td>0x1</td><td>ActLMAddr2 is Global. Only the working copy of ActLMAddr2 is used, independent of Active Context.</td></tr> </table>	0x0	ActLMAddr2 is Context Relative. Each Context uses a separate copy of ActLMAddr2, stored in IndLMAddr2.	0x1	ActLMAddr2 is Global. Only the working copy of ActLMAddr2 is used, independent of Active Context.
0x0	ActLMAddr2 is Context Relative. Each Context uses a separate copy of ActLMAddr2, stored in IndLMAddr2.								
0x1	ActLMAddr2 is Global. Only the working copy of ActLMAddr2 is used, independent of Active Context.								
20	1;20	NNsendConfig	rw	-	<p>This bit controls the ME's Next-Neighbor "Send" Interface; it controls whether a Next-Neighbor destination from this ME is written to the Next Neighbor ME (general-usage) or to this same ME (mostly for debugging usage); "same ME" refers to the ME executing the instruction which carries the NN destination; "same ME" or "Self" refer to the same configuration.</p> <table border="1"> <tr> <td>0x0</td><td>Next Neighbor destinations will be written to the NN register file located in the next ME</td></tr> </table>	0x0	Next Neighbor destinations will be written to the NN register file located in the next ME		
0x0	Next Neighbor destinations will be written to the NN register file located in the next ME								

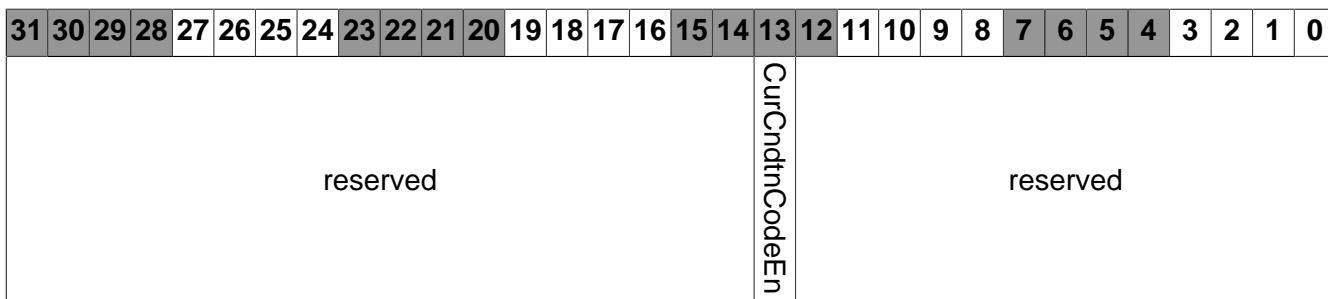
<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>									
					0x1	Next Neighbor destinations will be written to the NN register file located within this ME								
19:18	2;18	NextNeighborEmpty Assert	rw	-	Controls threshold when NN_Empty asserts. The number of entries valid is determined by comparing NNPut and NNGet Local CSRs. The use of indicating empty when there is really something on the Ring is if the cooperating processes transfer data in a block, and the consumer does not want to get a partial block.	<table border="1"> <tr><td>0x0</td><td>0 entries valid.</td></tr> <tr><td>0x1</td><td>&lt;=1 entries valid.</td></tr> <tr><td>0x2</td><td>&lt;=2 entries valid.</td></tr> <tr><td>0x3</td><td>&lt;=3 entries valid.</td></tr> </table>	0x0	0 entries valid.	0x1	<=1 entries valid.	0x2	<=2 entries valid.	0x3	<=3 entries valid.
0x0	0 entries valid.													
0x1	<=1 entries valid.													
0x2	<=2 entries valid.													
0x3	<=3 entries valid.													
17	1;17	LMAddr1Global	rw	-	Controls usage of ActLMAddr1.	<table border="1"> <tr><td>0x0</td><td>ActLMAddr1 is Context Relative. Each Context uses a separate copy of ActLMAddr1, stored in IndLMAddr1.</td></tr> <tr><td>0x1</td><td>ActLMAddr1 is Global. Only the working copy of ActLMAddr1 is used, independent of Active Context.</td></tr> </table>	0x0	ActLMAddr1 is Context Relative. Each Context uses a separate copy of ActLMAddr1, stored in IndLMAddr1.	0x1	ActLMAddr1 is Global. Only the working copy of ActLMAddr1 is used, independent of Active Context.				
0x0	ActLMAddr1 is Context Relative. Each Context uses a separate copy of ActLMAddr1, stored in IndLMAddr1.													
0x1	ActLMAddr1 is Global. Only the working copy of ActLMAddr1 is used, independent of Active Context.													
16	1;16	LMAddr0Global	rw	-	Controls usage of ActLMAddr0.	<table border="1"> <tr><td>0x0</td><td>ActLMAddr0 is Context Relative. Each Context uses a separate copy of ActLMAddr0, stored in IndLMAddr0.</td></tr> <tr><td>0x1</td><td>ActLMAddr0 is Global. Only the working copy of ActLMAddr0 is used, independent of Active Context.</td></tr> </table>	0x0	ActLMAddr0 is Context Relative. Each Context uses a separate copy of ActLMAddr0, stored in IndLMAddr0.	0x1	ActLMAddr0 is Global. Only the working copy of ActLMAddr0 is used, independent of Active Context.				
0x0	ActLMAddr0 is Context Relative. Each Context uses a separate copy of ActLMAddr0, stored in IndLMAddr0.													
0x1	ActLMAddr0 is Global. Only the working copy of ActLMAddr0 is used, independent of Active Context.													
15:8	8;8	CtxEnables	rw	0	Context Enables for Context 7 through Context 0.	<table border="1"> <tr><td>0x0</td><td>Context is in Inactive state.</td></tr> <tr><td>0x1</td><td>Context may be in Ready, Executing, or Sleep states.</td></tr> </table>	0x0	Context is in Inactive state.	0x1	Context may be in Ready, Executing, or Sleep states.				
0x0	Context is in Inactive state.													
0x1	Context may be in Ready, Executing, or Sleep states.													
7:3	5;3	Reserved	-	-	Reserved									
2:0	3;0	NNreceiveConfig	rw	-	This field controls the ME's Next-Neighbor "Receive" Interface; it selects which path has access to the NN Register File's write port; in MEv2.8, four different NN paths may be selected: NN from previous ME, Self, CTnn, and PushBus; please note that when the NNsendConfig bit is set to "Self", the NNreceiveConfig									

Bits	Width; Start	Field name	Access	Reset value	Description								
					field must also be set to "Self"; otherwise, the NN destination data generated from "this ME" is lost								
					<table border="1"> <tr> <td>0x0</td><td>Next Neighbor path from Previous ME is selected</td></tr> <tr> <td>0x1</td><td>Next Neighbor path from this ME is selected</td></tr> <tr> <td>0x2</td><td>Next Neighbor path from the Miscellaneous Engine in the CTM is selected</td></tr> <tr> <td>0x3</td><td>Next Neighbor path from the CPP Push Bus is selected</td></tr> </table>	0x0	Next Neighbor path from Previous ME is selected	0x1	Next Neighbor path from this ME is selected	0x2	Next Neighbor path from the Miscellaneous Engine in the CTM is selected	0x3	Next Neighbor path from the CPP Push Bus is selected
0x0	Next Neighbor path from Previous ME is selected												
0x1	Next Neighbor path from this ME is selected												
0x2	Next Neighbor path from the Miscellaneous Engine in the CTM is selected												
0x3	Next Neighbor path from the CPP Push Bus is selected												

#### 4.2.1.1.8 Offset 1Ch: CondCodeEn - Condition Code Enable Register

Assembler Register Name: CC\_ENABLE

The condition codes are always enabled for normal use and are disabled only during Microengine debugging where the GPRs are read by an external source (such as the ARM Core or another Microengine) and the condition codes need to be preserved.

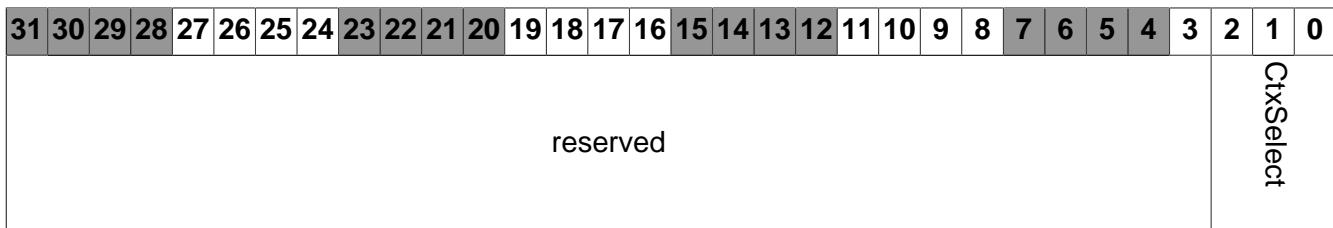


Bits	Width; Start	Field name	Access	Reset value	Description
31:14	18;14	Reserved	-	-	Reserved
13	1;13	CurCndtnCodeEn	rw	-	Current Condition Code Enable. Set to 1 to update the condition codes. When 0, condition codes will not be updated.
12:0	13;0	Reserved	-	-	Reserved

#### 4.2.1.1.9 Offset 20h: CSRCtxPtr - CSR Context Pointer Register

Assembler Register Name: CSR\_CTX\_POINTER

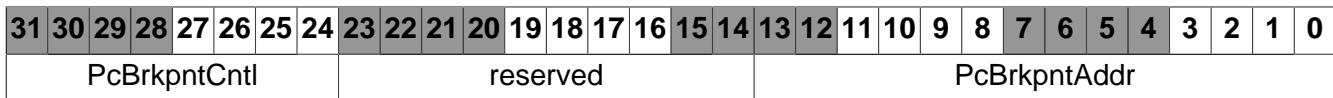
This register is used when reading or writing the Local CSRs that are unique per Context. An external Master (such as the ARM Core) or ME itself writes a context number into this register and then reads or writes any Local CSR with a name that starts with "Ind" to access the version of the register that is specific to the context.



Bits	Width; Start	Field name	Access	Reset value	Description
31:3	29:3	Reserved	-	-	Reserved
2:0	3:0	CtxSelect	rw	-	Selects which contexts Local CSR is accessed by local_csr_read, local_csr_write, and by the IA core.

#### 4.2.1.1.10 Offset 24h: PcBreakpoint0 - PC Breakpoint0

This control CSR is used to support the PCB mechanism described in the NFP-6xxx Databook, **PC Breakpoint - PCB mechanism section** ; please refer to this section for further details.



Bits	Width; Start	Field name	Access	Reset value	Description
31:24	8;24	PcBrkpntCntl	rw	0	PC Breakpoint Control: Controls the PC Breakpoint0 SDS-mechanism.
23:14	10;14	Reserved	-	-	Reserved
13:0	14;0	PcBrkpntAddr	rw	0	PC Breakpoint Address: PC Address corresponding to PcBreakpoint0; it gets compared to the ME Active PC value.

#### 4.2.1.1.11 Offset 28h: PcBreakpoint1 - PC Breakpoint1

This control CSR is used to support the PCB mechanism described in the NFP-6xxx Databook, **PC Breakpoint - PCB mechanism section** ; please refer to this section for further details.

Please see Offset 24h: PCBreakpoint0 for a description of the fields in this CSR; both PCBreakpoint0 and PCBreakpoint1 use the same CSR field layout and functionality.

#### 4.2.1.1.12 Offset 2Ch: PcBreakpointStatus - PC Breakpoint Status Register

This Status register is used (mostly) by external devices for obtaining feedback on the PC Breakpoint mechanism as described in the NFP-6xxx Databook, **PC Breakpoint - PCB mechanism section** . Status on both PcBreakpoint0 and PcBreakpoint1 is provided within this CSR.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved															PcBrkpntedCtx														PcBrkpnt0Status	PcBrkpnt1Status	

Bits	Width; Start	Field name	Access	Reset value	Description				
31:10	22:10	Reserved	-	-	Reserved				
9:2	8:2	PcBrkpntedCtx	rw1c	0	Indicates which context has triggered the PC breakpoint event; the least significant bit of this fields is associated with ctx 0; the most significant bit with ctx 7				
1	1:1	PcBrkpnt1Status	rw1c	0	Indicates the status of the PcBreakpoint1 SDS mechanism				
					<table border="1"> <tr> <td>0x0</td><td>PcBreakpoint0 has not triggered</td></tr> <tr> <td>0x1</td><td>PcBreakpoint0 has triggered</td></tr> </table>	0x0	PcBreakpoint0 has not triggered	0x1	PcBreakpoint0 has triggered
0x0	PcBreakpoint0 has not triggered								
0x1	PcBreakpoint0 has triggered								
0	1:0	PcBrkpnt0Status	rw1c	0	Indicates the status of the PcBreakpoint0 SDS mechanism				
					<table border="1"> <tr> <td>0x0</td><td>PcBreakpoint0 has not triggered</td></tr> <tr> <td>0x1</td><td>PcBreakpoint0 has triggered</td></tr> </table>	0x0	PcBreakpoint0 has not triggered	0x1	PcBreakpoint0 has triggered
0x0	PcBreakpoint0 has not triggered								
0x1	PcBreakpoint0 has triggered								

#### 4.2.1.1.13 Offset 30h: RegErrStatus - Register Error Status Register

Assembler Register Name: REGISTER\_ERROR\_STATUS

This status register records information about parity errors detected on Datapath Register Files, excluding LM (which is protected by ECC and has its own status CSR). The contents of this CSR are "frozen" when CtxEnables[RegisterParityErr] is a '1'; however, no other subsequent parity errors may occur since the ME simply halts on a first detection of a parity error.

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

Bits	Width; Start	Field name	Access	Reset value	Description		
31:18	14:18	Reserved	-	-	Reserved		
17:16	2:16	Type	ro	-	<p>Type of Register File where a Parity Error has been detected.</p> <table border="1"> <tr> <td>0x0</td><td>GPR</td></tr> </table>	0x0	GPR
0x0	GPR						

Bits	Width; Start	Field name	Access	Reset value	Description	
					0x1	Transfer Register
					0x2	Next Neighbor Register
15:0	16:0	RegisterNumber	ro	-	Entry Address of the Register File where a Parity Error has been detected. Unused upper bits will read 0.	

#### 4.2.1.1.14 Offset 34h: LMErrStatus - Local Memory Error Status Register

This status register captures information about ECC errors detected on Local Memory reads. The contents of this CSR are valid and frozen while CtxEnables[LM\_ECCError] remains a '1'. Clearing CtxEnables[LM\_ECCError] re-enables this register so that information on the next LM ECC error may be captured.



##### Note

In this context, "frozen" means no updates to this register is possible; thus, information pertaining to a second correctable error (or uncorrectable error) would be lost while the "frozen" state remains in effect. However, other ECC error activities would occur as normal; that is, a correctable error would still be corrected, and an uncorrectable error would still halt the ME.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
UncorrectableErr	reserved			Syndrome				reserved	Context		reserved		LMaddress																		

Bits	Width; Start	Field name	Access	Reset value	Description	
31	1;31	UncorrectableErr	ro	0	This bit indicates that the ECC error detected on Local Memory is uncorrectable	
					0x0	Correctable Error
					0x1	Uncorrectable Error. ME will enter the HALT state when this bit is set.
30:27	4;27	Reserved	-	-	Reserved	
26:20	7;20	Syndrome	ro	0	This field records the syndrome that was found when the ECC error occurred.	
19	1;19	Reserved	-	-	Reserved	
18:16	3;16	Context	ro	0	Context that was executing when the ECC error occurred	

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
15:10	6:10	Reserved	-	-	Reserved
9:0	10:0	LMaddress	ro	0	Contains the LM address where the ECC error has been detected.

#### 4.2.1.1.15 LM ECC Error Mask Register

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

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
31:0	32:0	ErrorInjectionMask	rw	0	This field is used for controlling Error Injection bits into an LM data-path word.

#### 4.2.1.1.16 Offset 40h: IndCtxStatus - Indirect Context Status Register

Assembler Register Name: INDIRECT\_CTX\_STS

There are eight IndCtxStatus registers, each contain the status of contexts 0 through 7. These registers are accessed indirectly using the CSRCtxPtr to select the specific context register and reading or writing this register. Note that the active context number can be read via the ActCtxStatus Local CSR.

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

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
31	1:31	ReadyToRun	ro	-	Ready to Run. Indicates that the context is in Ready state. (This bit will be 0 if the context is in any of the other three states.)				
					<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0x0</td><td>Not Ready</td></tr> <tr> <td>0x1</td><td>Ready</td></tr> </table>	0x0	Not Ready	0x1	Ready
0x0	Not Ready								
0x1	Ready								
30:14	17:14	Reserved	-	-	Reserved				
13:0	14:0	ContextPC	rw	-	The program counter at which the context begins executing when it is put into the executing state.				

#### 4.2.1.1.17 Offset 44h: ActCtxStatus - Active Context Status Register

Assembler Register Name: ACTIVE\_CTX\_STS

This register maintains the context number of the Context currently executing. Each Microengine supports eight contexts (0 through 7). This register also contains the ME number (ME\_ID) and the Island number (IL\_ID) where a given ME resides.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0																													
AB0	IslandId								ActiveContextPC												reserved	MENumber			ActiveContextNo				

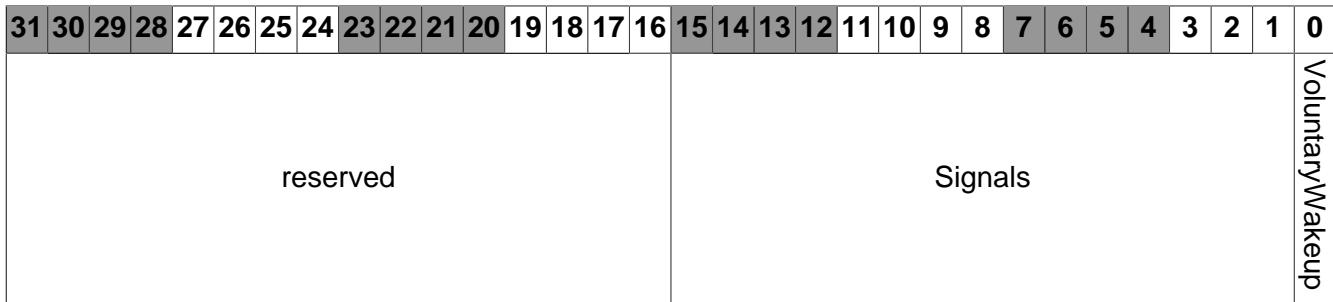
Bits	Width; Start	Field name	Access	Reset value	Description				
31	1;31	AB0	ro	0	If set, the microengine has a context in the Executing state. If clear, no context is in Executing state.  <table border="1"> <tr> <td>0x0</td><td>No context is running</td></tr> <tr> <td>0x1</td><td>A context is running</td></tr> </table>	0x0	No context is running	0x1	A context is running
0x0	No context is running								
0x1	A context is running								
30:25	6;25	IslandId	ro	-	Island number where the ME happens to be instantiated.				
24:8	17;8	ActiveContextPC	ro	0	PC of Executing Context. Only valid if AB0 is a 1. This field provides a snapshot value of the PC. This value is used for tracking/code profiling purposes. When issued as a local_csr_read from the Microengine, the PC value may not be the exact PC value of the local_csr_rd instruction.				
7	1;7	Reserved	-	-	Reserved				
6:3	4;3	MENumber	ro	-	A unique number which identifies the ME within the Island where it happens to be instantiated.				
2:0	3;0	ActiveContextNo	rw	0	The number of the Executing context. Only valid if AB0 bit is a 1.				

#### 4.2.1.1.18 Offset 48h: IndCtxSglEvt - Indirect Context Signal Events Register

Assembler Register Name: INDIRECT\_CTX\_SIG\_EVENTS

There are eight IndCtxSglEvt registers, and each contain status information that indicates which Event Signals have occurred for contexts 0 through 7. The registers are accessed indirectly using the **CSRCtxPtr** to select the specific context register, and reading or writing the IndCtxSglEvt register. The register for the context that is currently executing can be accessed by reading or writing the ActCtxStatus Local CSR.

This register is used in conjunction with the **IndCtxWkpEvt** register to move the Context from Sleep state to Ready state.



Bits	Width; Start	Field name	Access	Reset value	Description
31:16	16;16	Reserved	-	-	Reserved
15:1	15;1	Signals	rw	-	Each bit is set as described in the Event Signals section. Each is cleared by microengine hardware when: the signal is used to transition to Ready state if the CtxWkpEvt[AnyWakeEvnts] bit is clear, or a br_!signal on this signal is not taken, or a br_signal on this signal is taken.
0	1;0	VoluntaryWakeup	ro	1	Corresponds to Event for Voluntary arb wakeup event.

#### 4.2.1.1.19 Offset 4Ch: ActCtxSglEvt - Active Context Signal Events Register

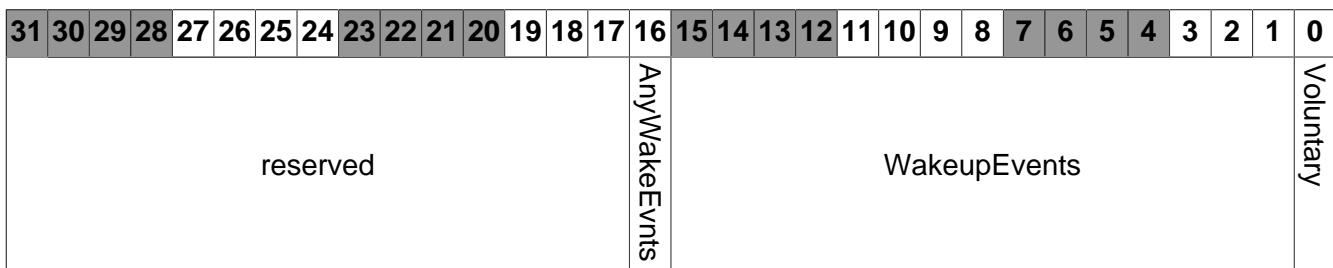
Assembler Register Name: ACTIVE\_CTX\_SIG\_EVENTS

This address is used to read the active copy of this register. Any of the eight context-specific copies can be read as described in [Offset 48h: IndCtxSglEvt - Indirect Context Signal Events Register](#).

#### 4.2.1.1.20 Offset 50h: IndCtxWkpEvt - Indirect Context Wakeup Events Register

Assembler Register Name: INDIRECT\_CTX\_WAKEUP\_EVENTS

There are eight IndCtxWkpEvt registers, and each contain status information that indicate which Event Signals are required to put the each of the Contexts into Ready state. The registers are accessed indirectly using the **CSRPtr** to select the specific context register and reading or writing the IndCtxWkpEvt register. The register for the context that is currently executing can be accessed by reading or writing the ActCtxWkpEvt Local CSR.



Bits	Width; Start	Field name	Access	Reset value	Description
31:17	15;17	Reserved	-	-	Reserved

Bits	Width; Start	Field name	Access	Reset value	Description				
16	1;16	AnyWakeEvnts	rw	-	<p>Any Wakeup Events. Set by the ANY token on a ctx_arb instruction. Note: this bit is undefined after a wakeup.</p> <table border="1"> <tr> <td>0x0</td><td>All mode (AND)</td></tr> <tr> <td>0x1</td><td>Any mode (OR)</td></tr> </table>	0x0	All mode (AND)	0x1	Any mode (OR)
0x0	All mode (AND)								
0x1	Any mode (OR)								
15:1	15;1	WakeupEvents	rw	-	Each wakeup event bit is set by either a ctx_swap_# token on an instruction, or by the Event Signal Mask of the ctx_arb instruction. All wakeup event bits are cleared by microengine hardware whether the context is put into execute state.				
0	1;0	Voluntary	rw	-	Set by ctx_arb[voluntary]. Cleared when the context is put into Execute state.				

#### 4.2.1.1.21 Offset 54h: ActCtxWkpEvt - Active Context Wakeup Events Register

Assembler Register Name: ACTIVE\_CTX\_WAKEUP\_EVENTS

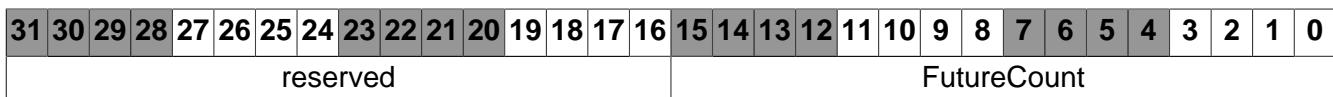
This address is used to read the active copy of this register. Any of the eight context-specific copies can be read as described in [Offset 50h: IndCtxWkpEvt - Indirect Context Wakeup Events Register](#).

#### 4.2.1.1.22 Offset 58h: IndCtxFtrCnt - Indirect Context Future Count Register

Assembler Register Name: INDIRECT\_CTX\_FUTURE\_COUNT

There are eight IndCtxFtrCnt registers for contexts 0 through 7. The value in each is compared to the value in the low 16-bits of Timestamp, and will set the signal specified in the IndFtrCntSgl register for this Context when it is armed and there is a match of FutureCount to Timestamp. Writing to the FutureCount bit arms the event. When the signal is set the event is disarmed (so that it will not set again when the lower half of Timestamp wraps around and matches FutureCount again).

The registers are accessed indirectly using the CSRCtxPtr to select the specific context register and reading or writing the IndCtxFtrCnt register. The register for the context that is currently executing can be accessed by reading or writing the ActCtxFtrCnt Local CSR.



Bits	Width; Start	Field name	Access	Reset value	Description
31:16	16;16	Reserved	-	-	Reserved
15:0	16;0	FutureCount	rw	-	Value to match against low 32-bits of Timestamp.

#### 4.2.1.1.23 Offset 5Ch: ActCtxFtrCnt - Active Context Future Count Register

Assembler Register Name: ACTIVE\_CTX\_FUTURE\_COUNT

This address is used to read the active copy of this register. Any of the eight context specific copies can be read as described in [Section 4.2.1.1.22](#).

#### **4.2.1.1.24 Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register**

Assembler Register Name: INDIRECT\_LM\_ADDR\_0

These registers hold the addresses that are used to read and write Local Memory (LM). Each Context has its own **IndLMAAddr0**, **IndLMAAddr1**, **IndLMAAddr2** and **IndLMAAddr3**. There is also a working copy of each. When a Context is put into Executing state, the value from its pair of LMAddr's are copied into the working pair.

Reads or writes of ActLMAAddr# select the working pair.

The context relative registers are accessed indirectly using the **CSRCtxPtr** to select the specific context register and reading or writing the IndLMAAddr0 and IndLMAAddr1 registers. The working registers can be accessed by reading or writing the ActLMAAddr0 and ActLMAAddr1 CSR. When the Context goes to Sleep state, the value in the working pair is moved to the Context Specific pair.

When a **LMAddr** is being used globally (as set in **CtxEnables[LMAddr#Global]** ), the working pair is used and never overwritten during Context swaps.

The working **LMAddr** can also be loaded with the result of a lookup\_cam instruction.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:12	20:12	Reserved	-	-	Reserved
11:2	10:2	LocalMemoryAddr	rw	-	Selects the specific 32-bit word in Local Memory. This field can be incremented or decremented by 1, or left unchanged after access, as specified in the instruction.
1:0	2:0	Reserved	-	-	Reserved

#### **4.2.1.1.25 Offset 64h: ActLMAAddr0 - Active Local Memory Address 0 Register**

Assembler Register Name: ACTIVE\_LM\_ADDR\_0

This address is used to write/read the active copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register”.

#### **4.2.1.1.26 Offset 68h: IndLMAAddr1 - Indirect Local Memory Register**

Assembler Register Name: INDIRECT\_LM\_ADDR\_1

This address is used to write/read the indirect copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register”.

#### 4.2.1.1.27 Offset 6Ch: ActLMAAddr1 - Active Local Memory Address 1 Register

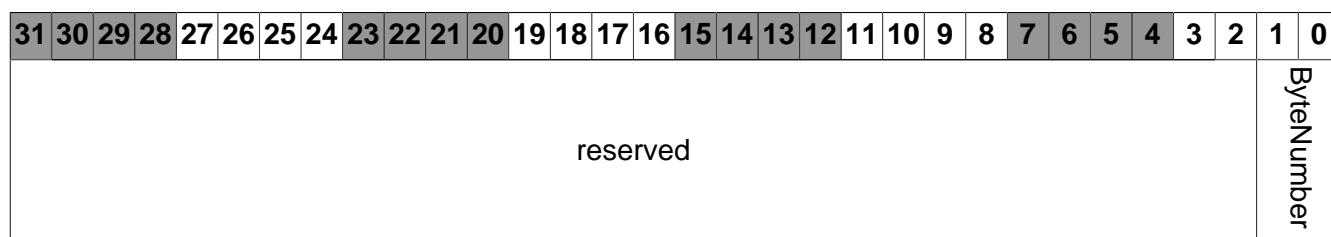
Assembler Register Name: ACTIVE\_LM\_ADDR\_1

This address is used to write/read the active copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register”.

#### 4.2.1.1.28 Offset 70h: ByteIndex - Byte Index Register

Assembler Register Name: BYTE\_INDEX

This register is used to control the byte shift amount during Byte\_Align instructions. This register can be written alone, or can be written along with **XferIndex** or **LMAAddr** (see [Offset F4h: XfrAndBytIdx - Transfer Index Byte Index Register](#)) and IndLMAAddr#BytIdx (see [Offset E0h: IndLMAAddr0BytIdx - Indirect Local Memory Address 0 Byte Index Register](#)).

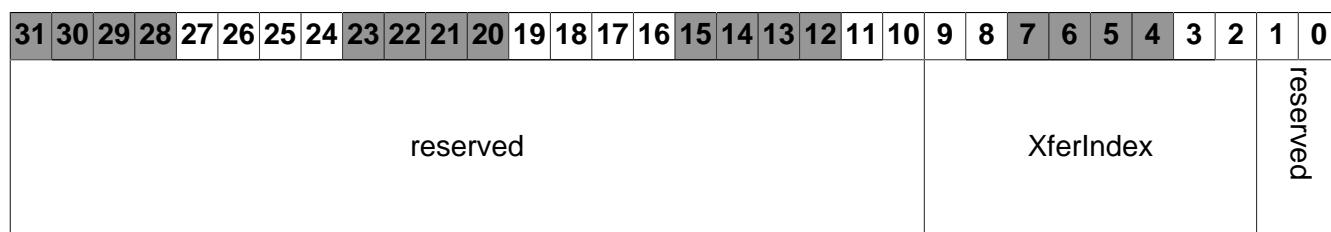


Bits	Width; Start	Field name	Access	Reset value	Description
31:2	30:2	Reserved	-	-	Reserved
1:0	2:0	ByteNumber	rw	-	Specifies a byte for use with the byte_align instruction.

#### 4.2.1.1.29 Offset 74h: XferIndex - Transfer Index Register

Assembler Register Name: T\_INDEX

This register is used when Transfer registers are accessed via indexed mode, which is specified in the source and destination fields of the instruction. This register can be incremented, decremented, or left unchanged after the access. This register can be written alone, or can be written along with **ByteIndex** (see [Offset F4h: XfrAndBytIdx - Transfer Index Byte Index Register](#)).



Bits	Width; Start	Field name	Access	Reset value	Description
31:10	22:10	Reserved	-	-	Reserved
9:2	8:2	XferIndex	rw	-	Transfer Register Index. Specifies one of 256 registers. The choice of TRANSFER_IN vs. TRANSFER_OUT is made based on the register use (either source or destination). This field can be incremented or decremented by 1, or left unchanged after the access, as specified in the instruction.
1:0	2:0	Reserved	-	-	Reserved

#### 4.2.1.1.30 Offset 78h: IndFtrCntSgl - Indirect Future Count Signal Register

Assembler Register Name: INDIRECT\_FUTURE\_COUNT\_SIGNAL

There are eight IndFtrCntSgl registers for contexts 0 through 7. The value in these registers indicate which signal to set when the FUTURE\_COUNT is used.

The registers are accessed indirectly using the CSRCtxPtr to select the specific context register and reading or writing the IndFtrCntSgl register. The register for the context that is currently executing can be accessed by reading or writing the ActFtrCntSgl Local CSR.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:4	28:4	Reserved	-	-	Reserved
3:0	4:0	SignalNo	rw	-	The signal number to set when FUTURE_COUNT == TIMESTAMP.

#### 4.2.1.1.31 Offset 7Ch: ActFtrCntSgl - Active Context Future Count Register

Assembler Register Name: ACTIVE\_FUTURE\_COUNT\_SIGNAL

See [Offset 78h: IndFtrCntSgl - Indirect Future Count Signal Register](#).

#### 4.2.1.1.32 Offset 80h: NNPut - Next Neighbor Put Register

Assembler Register Name: NN\_PUT

The NNPut register contains the “put” pointer used when the Next Neighbor registers are used as a Ring. When a *previous* Microengine executes an instruction that specifies the destination as \*n\$index, the Next Neighbor register in *this* Microengine is selected by the value in this register, and the value is then incremented by 1 (a value of 127 wraps back to 0). The value in this register is compared to the value in the **NNGet** register to determine when to assert NN\_FULL and NN\_EMPTY status signals.

The value in this register is compared to the value in the NNGet register to determine the number of valid entries in the NN ring. When receiving ME gets 96 valid entries, a FULL signal is sent to the NN\_receive\_fifo. At this point no more entries will be popped off this 8 deep NN\_receive\_fifo. Once this receive fifo fills up, the previous ME's 8 deep NN\_send\_fifo will become aware of this (due to credit trackers) and will stop sending data. Once this NN\_send\_fifo reaches 6 the FULL Input\_State will assert. Software should check for this INP\_STATE != FULL before sending. Upon detecting a NON-FULL condition it is always safe to send up to 3 entries (not full implies at least 3 available entries on NN\_send\_fifo).

One cannot immediately test for FULL after two or more consecutive PUTS. One needs at least two intervening instructions between last PUT of a sequence and the testing of the FULL state. It is OK to test for FULL immediately after a single PUT operation though.

#### **4.2.1.1.33 Offset 84h: NNGet - Next Neighbor Get Register**

Assembler Register Name: NN\_GET

This register contains the “get” pointer used when the Next Neighbor registers are used as a Ring. This register is used to specify the Next Neighbor register when a source operand is \*n\$index, and the value is then incremented (a value of 127 wraps back to 0). The value in this register is compared to the value in **NNPut** register to determine when to assert NN\_FULL and NN\_EMPTY status signals.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:7	25:7	Reserved	-	-	Reserved
6:0	7:0	NNRegIndex	rw	-	Specifies one of 128 NN registers to read.

#### **4.2.1.1.34 Offset 90h: IndLMAAddr2 - Indirect Local Memory Register**

This address is used to write/read the indirect copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register” (see Section 4.2.1.1.24 ).

#### **4.2.1.1.35 Offset 94h: ActLMAAddr2 - Active Local Memory Address 2 Register**

This address is used to write/read the active copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register” (see Section 4.2.1.1.24 ).

#### **4.2.1.1.36 Offset 98h: IndLMAAddr3 - Indirect Local Memory Register**

This address is used to write/read the indirect copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register” (see Section 4.2.1.1.24 ).

#### 4.2.1.1.37 Offset 9ch: ActLMAAddr3 - Active Local Memory Address 3 Register

This address is used to write/read the active copy of this register. Any of the eight context-specific copies can be read as described in “Offset 60h: IndLMAAddr0 - Indirect Local Memory Address 0 Register” (see Section 4.2.1.1.24 ).

#### 4.2.1.1.38 Offset B0h: IndPredCC - Indirect Predicate CC

Assembler Register Name: INDIRECT\_PREDICATE\_CC

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

Bits	Width; Start	Field name	Access	Reset value	Description																									
31:4	28:4	Reserved	-	-	Reserved																									
3:0	4:0	PredCCSel	rw	0	Encoded value for selecting the Predicate Condition Code; encoded values select Condition Codes as shown below	<table border="1"> <tr><td>0x0</td><td>Predicate based on the Z condition code.</td></tr> <tr><td>0x1</td><td>Predicate based on the ~Z condition code.</td></tr> <tr><td>0x2</td><td>Predicate based on the N condition code.</td></tr> <tr><td>0x3</td><td>Predicate based on the ~N condition code.</td></tr> <tr><td>0x4</td><td>Predicate based on the C condition code.</td></tr> <tr><td>0x5</td><td>Predicate based on the ~C condition code.</td></tr> <tr><td>0x6</td><td>Predicate based on the V condition code.</td></tr> <tr><td>0x7</td><td>Predicate based on the ~V condition code.</td></tr> <tr><td>0x8</td><td>Predicate based on the GE (greater or equal) condition code.</td></tr> <tr><td>0x9</td><td>Predicate based on the LT (less than) condition code.</td></tr> <tr><td>0xa</td><td>Predicate based on the LE (less than or equal) condition code.</td></tr> <tr><td>0xb</td><td>Predicate based on the GT (greater than) condition code.</td></tr> </table>	0x0	Predicate based on the Z condition code.	0x1	Predicate based on the ~Z condition code.	0x2	Predicate based on the N condition code.	0x3	Predicate based on the ~N condition code.	0x4	Predicate based on the C condition code.	0x5	Predicate based on the ~C condition code.	0x6	Predicate based on the V condition code.	0x7	Predicate based on the ~V condition code.	0x8	Predicate based on the GE (greater or equal) condition code.	0x9	Predicate based on the LT (less than) condition code.	0xa	Predicate based on the LE (less than or equal) condition code.	0xb	Predicate based on the GT (greater than) condition code.
0x0	Predicate based on the Z condition code.																													
0x1	Predicate based on the ~Z condition code.																													
0x2	Predicate based on the N condition code.																													
0x3	Predicate based on the ~N condition code.																													
0x4	Predicate based on the C condition code.																													
0x5	Predicate based on the ~C condition code.																													
0x6	Predicate based on the V condition code.																													
0x7	Predicate based on the ~V condition code.																													
0x8	Predicate based on the GE (greater or equal) condition code.																													
0x9	Predicate based on the LT (less than) condition code.																													
0xa	Predicate based on the LE (less than or equal) condition code.																													
0xb	Predicate based on the GT (greater than) condition code.																													

#### 4.2.1.1.39 Offset C0h: TimestampLow - Timestamp Low Register

Assembler Register Name: TIMESTAMP\_LOW

Please see Section 4.2.1.1.40 for details on the TimestampHgh CSR and for a general description of the Timestamp mechanism itself.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32;0	CountLower	rw	-	Current 64-bit count value (lower 32 bits).

#### 4.2.1.1.40 Offset C4h: TimestampHigh - Timestamp High Register

Assembler Register Name: TIMESTAMP\_HIGH

The Timestamp CSR is a 64-bit wide register; it comprises 2 32-bit registers (TimestampHigh and TimestampLow - upper and lower 32-bit segments respectively), each with its own CSR address. It is a general purpose counter that increments by one every sixteen T cycles. When the maximum count is reached, the Timestamp CSR wraps around to 64h0 on the next increment. Island-level CSRs (please see the for details) provide control over 2 Timestamp functions: island-level enabling and island-level resetting. With the first functionality, the Timestamp CSRs in all ME units (within an Island) are enabled at the same time (same cycle) and thus start counting at the same time. With the second functionality, the Timestamp CSRs in all ME units (within an Island) are cleared (64h0) at the same time (same cycle); the clearing functionality has priority over the enabling functionality (in other words, there is no need to disable the Timestamp CSRs prior to clearing them - if enabled, these CSRs will simply start counting from 64h0 as soon as the clearing signal deasserts).

Regarding write transactions, both TimestampHigh and TimestampLow CSRs have internal and external write access; they may be independently written with specific values for the starting counting point. A write transaction to these CSRs must be done while they are disabled; once enabled (that is, once they are counting), all writes are ignored.

Regarding internal reads, both TimestampHigh and TimestampLow CSRs may be independently read, but a synchronized 64 bit snapshot is only provided when a TimestampLow read instruction is followed by a TimestampHigh read instruction (not necessarily back-to-back). Whenever a TimestampLow read instruction occurs, the hardware automatically makes a copy of the TimestampHigh CSR into a TimestampHigh "shadow" register; a subsequent read instruction of the TimestampHigh CSR is in effect a read of the "shadow" register. The net result is that the reading sequence (first Low, then High) captures a 64 bit value which belongs to the same instant of time (that is, to the same T cycle).

Regarding external reads, both TimestampHigh and TimestampLow CSRs may be read as a single 64 bit transaction. When the external read transaction targets the TimestampLow CSR, the ME automatically supplies both CSRs as the returned data as follows: lower 32 bits carry TimestampLow value; upper 32 bits carry TimestampHigh value. When the external read transaction targets the TimestampHigh CSR, the ME simply treats it as a normal 32-bit CSR read (that is, the value of TimestampHigh is carried in the lower 32 bits of the returned data). All this implies that external reads provide data directly from the TimestampHigh CSR itself, bypassing the "shadow" register.

Please see [Section 4.2.1.1.39](#) for further details on the TimestampLow CSR.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32;0	CountUpper	rw	-	Current 64-bit count value (upper 32 bits).

#### 4.2.1.1.41 Offset E0h: IndLMAAddr0BytIdx - Indirect Local Memory Address 0 Byte Index Register

Assembler Register Name: INDIRECT\_LM\_ADDR\_0\_BYTE\_INDEX

This is an alias of the IndLMAAddr0 (see [Section 4.2.1.1.24](#)) and ByteIndex (see [Section 4.2.1.1.28](#)) registers. Reading and writing this register reads and writes both the IndLMAAddr0 and ByteIndex registers.

#### 4.2.1.1.42 Offset E4h: ActLMAAddr0BytIdx - Active Local Memory Address 0 Byte Index Register

Assembler Register Name: ACTIVE\_LM\_ADDR\_0\_BYTE\_INDEX

This address is used to read the active copy of this register. Any of the eight context-specific copies can be read as described in [Offset E0h: IndLMAAddr0BytIdx - Indirect Local Memory Address 0 Byte Index Register](#).

#### 4.2.1.1.43 Offset E8h: IndLMAAddr1BytIdx - Indirect Local Memory Address 1 Byte Index Register

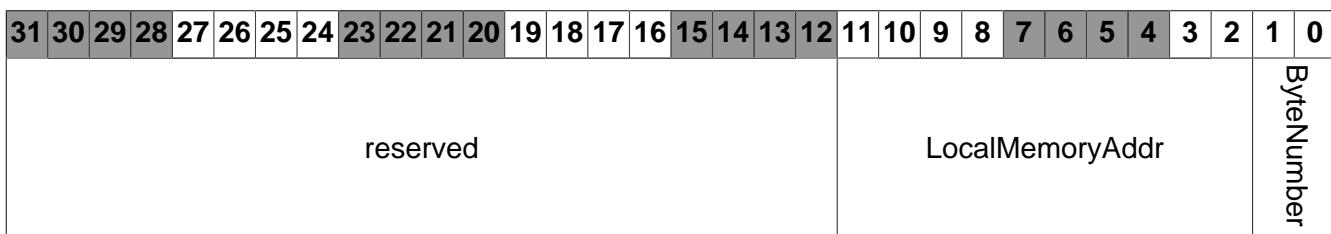
Assembler Register Name: INDIRECT\_LM\_ADDR\_1\_BYTE\_INDEX

This is an alias of the IndLMAAddr1 (see [Section 4.2.1.1.26](#)) and ByteIndex (see [Section 4.2.1.1.28](#)) registers. Reading and writing this register reads and writes both the IndLMAAddr1 and ByteIndex registers.

#### 4.2.1.1.44 Offset ECh: ActLMAAddr1BytIdx - Active Local Memory Address 1 Byte Index Register

Assembler Register Name: ACTIVE\_LM\_ADDR\_1\_BYTE\_INDEX

This address is used to read the active copy of this register. Any of the eight context-specific copies can be read as described in [Offset E8h: IndLMAAddr1BytIdx - Indirect Local Memory Address 1 Byte Index Register](#).



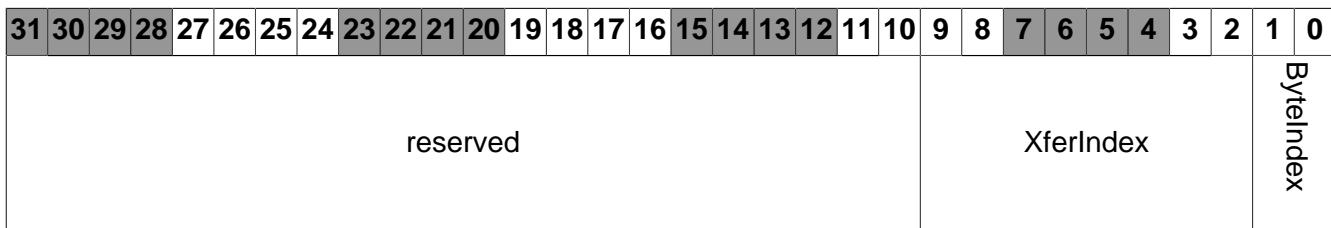
Bits	Width; Start	Field name	Access	Reset value	Description
31:12	20;12	Reserved	-	-	Reserved
11:2	10;2	LocalMemoryAddr	rw	-	Indirect Local Memory Address Register.

Bits	Width; Start	Field name	Access	Reset value	Description
1:0	2;0	ByteNumber	rw	-	Byte Index Register.

#### 4.2.1.1.45 Offset F4h: XfrAndBytIdx - Transfer Index Byte Index Register

Assembler Register Name: T\_INDEX\_BYTE\_INDEX

This is an alias of the XferIndex (see [Section 4.2.1.1.29](#)) and the ByteIndex (see [Section 4.2.1.1.28](#)) registers. Reading and writing this register reads and writes both the XferIndex and ByteIndex registers.



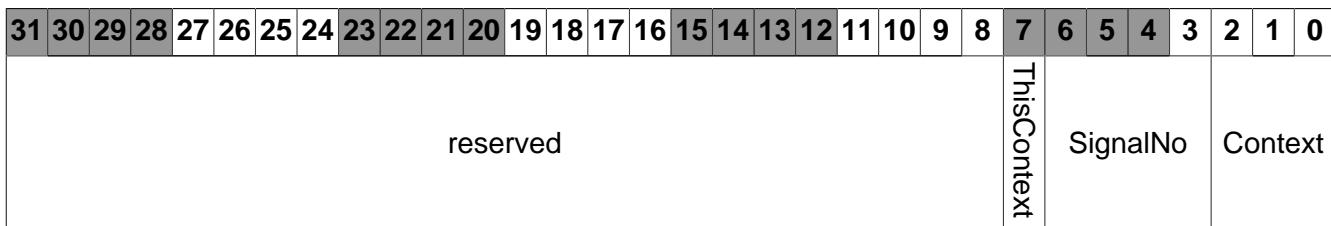
Bits	Width; Start	Field name	Access	Reset value	Description
31:10	22;10	Reserved	-	-	Reserved
9:2	8;2	XferIndex	rw	-	XferIndex: Transfer Index Register
1:0	2;0	ByteIndex	rw	-	ByteIndex - Byte Index Register

#### 4.2.1.1.46 Offset 100h: NxtNghbrSgl - Next Neighbor Signal Register

Assembler Register Name: NEXT\_NEIGHBOR\_SIGNAL

The **NxtNghbrSgl** CSR allows a program to signal a Context in the Next Neighbor Microengine. The data is used to select which Context and Event Signal number is set.

This register must not be written to on two consecutive instructions. The limitation is due to the fact that the I/O signals of the Microengine run at one-half the internal rate.



Bits	Width; Start	Field name	Access	Reset value	Description
31:8	24;8	Reserved	-	-	Reserved

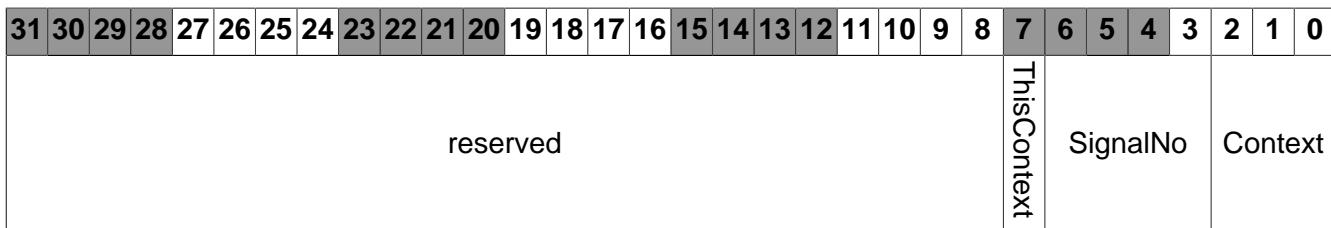
<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
7	1;7	ThisContext	wo	0	<p>Controls whether or not the context field of this register is used in selecting the context that is signaled in the next neighbor microengine.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Use the context field of this register to select the context in the next neighbor micorengine.</td></tr> <tr> <td>0x1</td><td>Signal the same numbered context as the one that does the write to this register. This is useful if running common code on multiple contexts that need to signal the same numbered context.</td></tr> </table>	0x0	Use the context field of this register to select the context in the next neighbor micorengine.	0x1	Signal the same numbered context as the one that does the write to this register. This is useful if running common code on multiple contexts that need to signal the same numbered context.
0x0	Use the context field of this register to select the context in the next neighbor micorengine.								
0x1	Signal the same numbered context as the one that does the write to this register. This is useful if running common code on multiple contexts that need to signal the same numbered context.								
6:3	4;3	SignalNo	wo	0	Signal to set in the next neighbor microengine.				
2:0	3;0	Context	wo	0	Context to signal in the next neighbor microengine. This field is only used if ThisContext is not asserted.				

#### 4.2.1.1.47 Offset 104h: PrvNghbrSgl - Previous Neighbor Signal Register

Assembler Register Name: PREV\_NEIGHBOR\_SIGNAL

The PrvNghbrSgl CSR allows a program to signal a Context in the Previous Neighbor Microengine (the next lower numbered Microengine). The data is used to select which context and signal number is set.

This register must not be written to on two consecutive instructions. The limitation is due to the fact that the I/O signals of the Microengine run at one-half the internal rate.



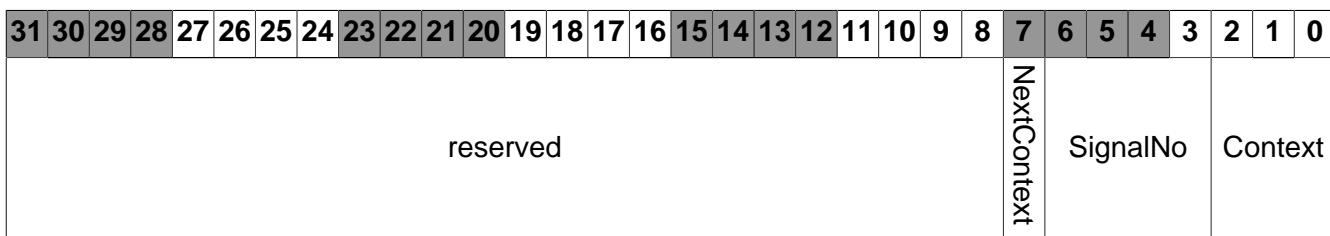
<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
31:8	24;8	Reserved	-	-	Reserved				
7	1;7	ThisContext	wo	0	<p>Controls whether or not the context field of this register is used in selecting the context that is signaled in the previous neighbor microengine.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Use the context field of this register to select the context in the previous neighbor micorengine.</td></tr> <tr> <td>0x1</td><td>Signal the same numbered context as the one that does the write to this register. This is</td></tr> </table>	0x0	Use the context field of this register to select the context in the previous neighbor micorengine.	0x1	Signal the same numbered context as the one that does the write to this register. This is
0x0	Use the context field of this register to select the context in the previous neighbor micorengine.								
0x1	Signal the same numbered context as the one that does the write to this register. This is								

Bits	Width; Start	Field name	Access	Reset value	Description		
					useful if running common code on multiple contexts that need to signal the same numbered context.		
6:3	4;3	SignalNo	wo	0	Signal to set in the previous neighbor microengine.		
2:0	3;0	Context	wo	0	Context to signal in the previous neighbor microengine. This field is only used if ThisContext is not asserted.		

#### 4.2.1.1.48 Offset 108h: SameMESignal - Same Microengine Signal Register

Assembler Register Name: SAME\_ME\_SIGNAL

The **SameMESignal** CSR allows a thread to signal another Context in the same Microengine. The data is used to select which Context and signal number is set.



Bits	Width; Start	Field name	Access	Reset value	Description				
31:8	24;8	Reserved	-	-	Reserved				
7	1;7	NextContext	wo	0	Controls whether or not the Context field of this register is used in selecting the context that is signaled in the microengine.				
6:3	4;3				0x0	Use the Context field of this register to select the context in the microengine.			
2:0	3;0	Context	wo	0	0x1	Signal the context this is one greater (modulo the number of active contexts in CtxEnables[InUseContexts]) as the one that does the write to this register. This is useful if running common code on multiple contexts that need to signal the next sequential context.			
6:3	4;3	SignalNo	wo	0	Signal to set in microengine.				
2:0	3;0	Context	wo	0	Context to signal in the microengine. This field is only used if NextContext field is not asserted.				

#### 4.2.1.1.49 Offset 140h: CRC\_REMAINDER - CRC Remainder Register

Assembler Register Name: CRC\_REMAINDER

**CRCRemainder** provides one operand to the CRC Unit, and gets written with the result of the CRC operation after a CRC instruction.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32;0	CRCRemainder	rw	-	Input operand and result of the CRC instruction.

#### 4.2.1.1.50 Offset 144h: ProfileCnt - Profile Count Register

Assembler Register Name: PROFILE\_COUNT

**ProfileCnt** is used for code profiling and tuning. It counts once per cycle, and when it reaches maximum value it wraps to 0.

A typical use of **ProfileCnt** is to read and store the value in a register, start an external event (for example an external memory read), and then, when the external event completes, read the value in **PROFILE\_COUNT** again. Subtracting the two values will give the elapsed number of cycles.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:16	16;16	Reserved	-	-	Reserved
15:0	16;0	Count	rw	-	Advances by one every cycle.

#### 4.2.1.1.51 Offset 148h: PseudoRndNum - Pseudo Random Number Register

Assembler Register Name: PSEUDO\_RANDOM\_NUMBER

**PseudoRndNum** (PRN) uses a Linear Feedback Shift Register (LFSR) to generate a pseudo random number which can be used by Microengine software. It can be initialized by a local\_csr\_wr, and the number changes after each read by a local\_csr\_rd, according to the polynomial:

$$x^{32} + x^{31} + x^{28} + x^{27} + x^{24} + x^{17} + x^{16} + x^{14} + x^{13} + x^{12} + x^{11} + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + 1.$$

The period of the output sequence is  $2^{32} - 1$ .



#### Note

When a PRN is generated, it is determined by the PRN Mode bit in the “Offset 18h: CtxEnables - Context Enables Register” CSR.



### Note

The PRN will be updated if the local\_csr\_rd occurs in the shadow of a taken branch. This is OK as long as one just wants a random number. However, if a guaranteed repeatable random sequence is desired, then do not place local\_csr\_rd of PRN in branch shadow.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32:0	Number	rw	-	Pseudo Random Number. This field must be non-zero to generate a pseudo random number.

### 4.2.1.1.52 Offset 160h: MiscControl - Miscellaneous Control Register

This control CSR is used for enabling a number of architectural features introduced over several generations of MEs. Some fields control fundamental ME modes of operation, such as whether ME will operate in SCS mode; other fields are simply reserved for Netronome use. Programmers should be aware that the MiscControl CSR contains fields with significantly different levels of ME behavior control; to better understand this issue, the following "field classes" are defined below.

The phrase "initial ME set up" used in the subsequent paragraphs below means: the ME CSRs set-up that system software performs before any of ME contexts gets enabled.

**Class 1:** Fields are defined for initial ME setup exclusively - updating these fields while ME is active (a context is running) will lead to undefined ME behavior. Contexts must be idled and disabled in order to update this class of field. The CSR fields under class 1 are:

1. ShareUStore
2. LegacyIndRef

**Class 2:** Fields are defined for initial ME set up, but they can also be updated by running code as software needs may require it. The CSR fields under class 2 are:

1. ThrdPrtyAdrMode
2. CsECCCrcEn
3. ParityEnable

**Class 3:** Fields are defined for Netronome use exclusively; these fields are used for silicon testing/debugging; it is recommended to keep these fields in their reset values. The CSR fields under class 3 are:

1. Interrupt
2. ForceBadParity
3. ForceBadECConLM

#### 4. CsECCCorrGate

**Class 4:** These fields can be dynamically updated by running u-code according to the particular requirements of an ME application. The CSR fields under class 4 are:

1. LMregionOnPush
2. LMregionOnPull

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

Bits	Width; Start	Field name	Access	Reset value	Description				
31	1;31	Reserved	-	-	Reserved				
30	1;30	ReqOnCsEccError	rw	0	<p>This bit enables the ME to generate an external Event request (ATTN asserts) on CS ECC correctable error.</p> <table border="1"> <tr> <td>0x0</td><td>Do not generate an external request on CS ECC correctable error.</td></tr> <tr> <td>0x1</td><td>Generate an external request on CS ECC correctable error.</td></tr> </table>	0x0	Do not generate an external request on CS ECC correctable error.	0x1	Generate an external request on CS ECC correctable error.
0x0	Do not generate an external request on CS ECC correctable error.								
0x1	Generate an external request on CS ECC correctable error.								
28:27	2;27	Reserved	-	-	Reserved				
26	1;26	Interrupt	rw	0	<p>This bit supports the External Interrupt functionality.</p> <table border="1"> <tr> <td>0x0</td><td>Not interrupted - ME is any of its normal mode of operation.</td></tr> <tr> <td>0x1</td><td>An external interrupt request has been detected, and ME is in the interrupt state.</td></tr> </table>	0x0	Not interrupted - ME is any of its normal mode of operation.	0x1	An external interrupt request has been detected, and ME is in the interrupt state.
0x0	Not interrupted - ME is any of its normal mode of operation.								
0x1	An external interrupt request has been detected, and ME is in the interrupt state.								
25	1;25	ECCCorrGate	rw	0	<p>Correction on Control Store ECC errors. This bit is used to enable testing that the microengine responds correctly to hard ECC errors.</p> <table border="1"> <tr> <td>0x0</td><td>No effect on correcting Control Store ECC errors.</td></tr> <tr> <td>0x1</td><td>Gate the writeback of corrected instruction upon detecting a correctable ECC error in Control Store.</td></tr> </table>	0x0	No effect on correcting Control Store ECC errors.	0x1	Gate the writeback of corrected instruction upon detecting a correctable ECC error in Control Store.
0x0	No effect on correcting Control Store ECC errors.								
0x1	Gate the writeback of corrected instruction upon detecting a correctable ECC error in Control Store.								
24	1;24	ParityEnable	rw	0	Parity checking on register files.				

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>	
					0x0	No parity checking is done on Register Files. Note that parity is still written when registers are written.
					0x1	Parity checking is done on Register Files.
23	1;23	ForceBadParity	rw	0	Force incorrect parity value on register file when registers are written. This bit is used for manufacturing test and should be left as 0 for normal operation.	
					0x0	Correct parity value is written to Register Files when register is written.
					0x1	Incorrect parity value (eg. the inverse of the correct parity) is written to Register Files, when register is written.
22:13	10;13	Reserved	-	-	Reserved	
12	1;12	UstorECCCrcEn	rw	0	UstoreECCCorrectEnable: Enables the microengine to correct correctable control store ECC errors. Note that ECC checking must be enabled in CtxEnables for this to have any meaning. Also note that when ECC error correcting is enabled, local CSR shadow rule is in effect. This means that any time a local CSR is changed by program control, it must not be read until the write-to-use latency is met. Specifically, software must not count on being able to user the value prior to the write in the write-to-use latency shadow. This applies to all local CSRs except LMAddr, XferIndex, NNPut, and NNGet. For those registers the HW will guarantee that the old value can be used in the write-to-use latency shadow.	
					0x0	Disabled. No correcting will be done. If ECC checking is enabled, all ECC errors will halt the microengine.
					0x1	Enabled. The microengine will automatically correct all correctable errors.
11:5	7;5	Reserved	-	-	Reserved	
4	1;4	ThrdPrtyAdrMode	rw	0	Selects between 32-bit and 40-bit addressing for the 3rd Party command instruction.	
					0x0	Calculate command bus address using 40-bit addressing.
					0x1	Calculate command bus address using 32-bit addressing.

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>				
3	1;3	LegacyIndRef	rw	0	<p>Selects between Normal and Legacy Indirect Reference Modes.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Selects Normal Indirect Reference Mode.</td></tr> <tr> <td>0x1</td><td>Selects Legacy Indirect Reference Mode.</td></tr> </table>	0x0	Selects Normal Indirect Reference Mode.	0x1	Selects Legacy Indirect Reference Mode.
0x0	Selects Normal Indirect Reference Mode.								
0x1	Selects Legacy Indirect Reference Mode.								
2	1;2	ShareUstore	rw	0	<p>Enables Shared Control Store between the two MEs in an ME group.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Not shared. Each microengine uses only its own control store. Maximum of 8K instructions in a program.</td></tr> <tr> <td>0x1</td><td>Shared. Each microengine uses only its own control store. Maximum of 16K instructions in a program.</td></tr> </table>	0x0	Not shared. Each microengine uses only its own control store. Maximum of 8K instructions in a program.	0x1	Shared. Each microengine uses only its own control store. Maximum of 16K instructions in a program.
0x0	Not shared. Each microengine uses only its own control store. Maximum of 8K instructions in a program.								
0x1	Shared. Each microengine uses only its own control store. Maximum of 16K instructions in a program.								
1	1;1	LMRegionPull	rw	0	<p>Selects between the lower and upper halves of the 1K LM address space for CPP Pull transactions.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Lower Half of Local Memory is selected.</td></tr> <tr> <td>0x1</td><td>Upper Half of Local Memory is selected.</td></tr> </table>	0x0	Lower Half of Local Memory is selected.	0x1	Upper Half of Local Memory is selected.
0x0	Lower Half of Local Memory is selected.								
0x1	Upper Half of Local Memory is selected.								
0	1;0	LMRegionPush	rw	0	<p>Selects between the lower and upper halves of the 1K LM address space for CPP Push transactions.</p> <table border="1" style="margin-left: 20px;"> <tr> <td>0x0</td><td>Lower Half of Local Memory is selected.</td></tr> <tr> <td>0x1</td><td>Upper Half of Local Memory is selected.</td></tr> </table>	0x0	Lower Half of Local Memory is selected.	0x1	Upper Half of Local Memory is selected.
0x0	Lower Half of Local Memory is selected.								
0x1	Upper Half of Local Memory is selected.								

#### 4.2.1.1.53 Offset 164h-168h: PcBreakpointMask Registers - PC Breakpoint0/1 Mask - Control Register

These control registers are used for masking the matching of the PC address in PcBreakpoint0/1 as described in the NFP-6xxx Databook, **PC Breakpoint - PCB mechanism section** .

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

<b>Bits</b>	<b>Width; Start</b>	<b>Field name</b>	<b>Access</b>	<b>Reset value</b>	<b>Description</b>
31:14	18;14	Reserved	-	-	Reserved
13:0	14;0	PcMask	rw	-	Masks the PC (program counter) bits which should not be considered for comparison against the value in the PcBrkpt field of the PcBreakpoint CSR.

#### 4.2.1.1.54 Offset 0170h-017C: Mailbox0, Mailbox1, Mailbox2, Mailbox3 - Mailbox Registers 0-3

Assembler Register Name: MAILBOX\_0, MAILBOX\_1, MAILBOX\_2, MAILBOX\_3

The Mailbox registers are for general software usage. They can be used for passing messages among contexts within an ME, or between contexts in an ME and a remote processor; this remote processor may be a system management processor, a PCIe bus based host processor, or another ME within the chip. The actual usage and meaning of these messages are entirely defined by the application software.

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

Bits	Width; Start	Field name	Access	Reset value	Description
31:0	32:0	Mailbox	rw	0	Mailbox Data value defined by the application.

#### 4.2.1.1.55 Offset 0190h: CmdIndirectRef0 - Control Register

Assembler Register Name: CMD\_INDIRECT\_REF\_0

It supports the command instruction's indirect reference in the

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved	Island			Master			SignalMaster			SignalCtx			SignalNum			reserved	ByteMask														

Bits	Width; Start	Field name	Access	Reset value	Description
31:30	2;30	Reserved	-	-	Reserved
29:24	6;24	Island	rw	0	Override value for the CPP command's island field.
23:20	4;20	Master	rw	0	Various overriding definitions depending on the value of the OVE_MASTER field in the previous ALU.
19:16	4;16	SignalMaster	rw	0	Override value for the CPP command's signal master field.
15:13	3;13	SignalCtx	rw	0	Override value for the CPP command's signal context field.
12:9	4;9	SignalNum	rw	0	Override value for the CPP command's signal number field.
8	1;8	Reserved	-	-	Reserved
7:0	8;0	ByteMask	rw	0	Override value for the CPP command's byte mask field.

## 5. Programming Examples

---

This chapter presents programming examples using NFP-6xxx micro-code. The examples include processing IPv4 header, calculating TCP checksum, using byte\_align instruction to align targeted data for packet header processing.

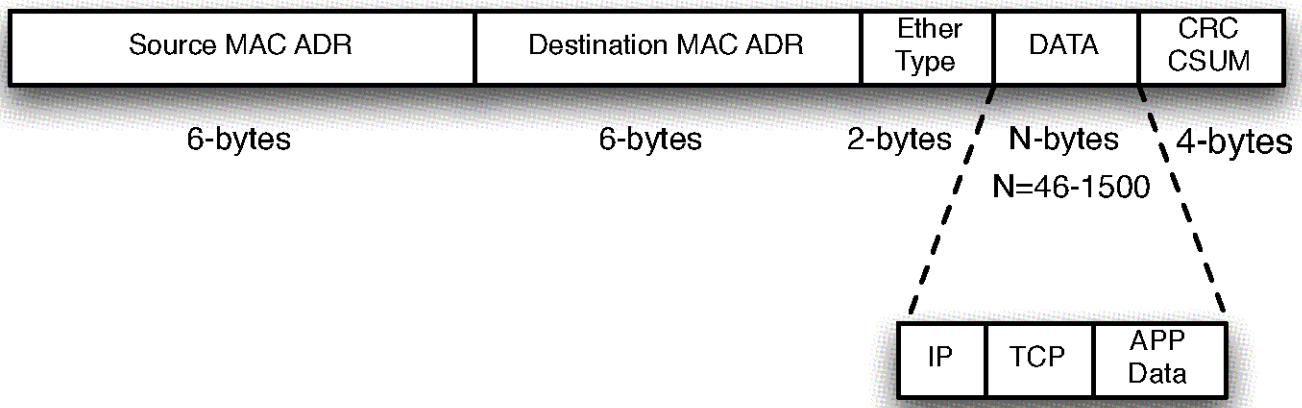
### 5.1 IPv4 Packet processing

An IPv4 Packet is received by the external Ethernet Interface and is processed by the NBI and the 20 byte IPv4 header is pushed to the Flow Processing Core (FPC) XFER-In registers. The FPC thread 0 is the receive thread that sole responsibility is to process IPv4 packets on the Ethernet Interface. As thread 0 is wakened by a signal from the CTM Packet Engine completed push operation of the packet's IPv4 header to the FPC core thread 0 XFER-In registers.

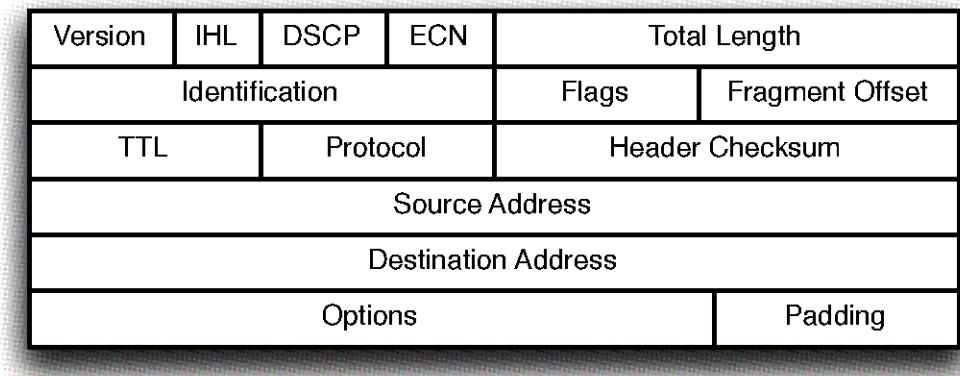
Here the FPC thread's job is to check the IPv4 packet headers like the version, IHL, Length, protocol and checksum. Once the IPv4 header fields are successfully checked the IPv4 TTL field is decremented.

Figure below shows the IPv4 Header fields.

## Ethernet Type II Frame



## IPv4 Header



**Figure 5.1. IPv4 Header Fields**

The metadata in FPC Read Transfer registers (\$in\_pkt[]) is shown as follows:

- \$in\_pkt[0] : Ethernet destination address bytes 1-4
- \$in\_pkt[1] : Ethernet destination address bytes 5-6, Ethernet source address bytes 3-6
- \$in\_pkt[2] : Ethernet source address bytes 3-6
- \$in\_pkt[3] : Ether Type and IP Header Version, IHL, DSC and ECN fields
- \$in\_pkt[4] : IP Header Total Length and ID
- \$in\_pkt[5] : IP Header Flags, Fragment Offset, TTL, and Protocol fields

- \$in\_pkt[6] : IP Header Checksum, Source IP Address 1 and 2
- \$in\_pkt[7] : IP Header Source IP Address bytes 3 and 4, Destination IP Address 1 and 2
- \$in\_pkt[8] : IP Header Destination bytes 3 and 4, data 2 bytes

The example code below show the IPv4 Header metadata is processed by FPC core thread 0. Note that the Ethernet MAC header followed by IPv4 Header is stored in the FPC read transfer registers \$in\_pkt[]. To remove the Ethernet MAC header while storing the IPv4 Header in FPC GPR registers byte\_align\_be[] is used and is shown in the example code below. This command will select a byte offset into the read transfer register and allow moving read transfer registers data into specified FPC GPRs.

```

; -----
; FPC 4
; -----
.reg ipHdr_0 ipHdr_1 ipHdr_2 ipHdr_3 ipHdr_4
.reg ipTtl ipIhl ipLen ipProtocol
.reg temp
.reg check
.reg $in_pkt[10] // must be even numbers since CTM is 64-bit accesses
.reg $out_pkt[4] // must be even numbers since CTM is 64-bit accesses
.xfer_order $in_pkt
.xfer_order $out_pkt

.reg eth_da1_4
.reg eth_da5_6_sa1_2
.reg eth_sa3_6
.reg ethType_ipHdri_2
.reg ipTlen_Id
.reg ipFlg_TtlProt
.reg ipCsum_SAdrl_2
.reg ipSAdr3_4_DAdrl_2
.reg ipDAdr3_4_Data
.reg r_mem_addr_ulw
.reg r_mem_addr_llw
.reg data1 r_wait
.reg temp2

; Signal wakes up FPC Core thread 0 when the packet is available
; Move read transfer register to GPRS
; Setup for the byte_align_be[] to to select 4-5 byte in 32-bit
; read transfer register $in_pkt[3]. Three nop instr are required
; for read to use latency.
local_csr_wr[BYTE_INDEX, 2]
nop
nop
nop

; Load FPC GPRs with IP Header fields ONLY.
byte_align_be[--, $in_pkt[3]] ; skip first two bytes (Ethr type)

; Load $in_pkt[3] last two bytes with first two bytes in $in_pkt[4].
; Continue same adjustments for $in_pkt[5] - $in_pkt[8].
; Throw away $in_pkt[8] last two bytes.
byte_align_be[ipHdr_0, $in_pkt[4]]
byte_align_be[ipHdr_1, $in_pkt[5]]
byte_align_be[ipHdr_2, $in_pkt[6]]
byte_align_be[ipHdr_3, $in_pkt[7]]

```

```
byte_align_be[ipHdr_4, $in_pkt[8]]  
  
; check IPv4 version is equal to 4  
alu_shf[temp, 0xf, AND, ipHdr_0, >>28]  
alu[--, 0x4, -, temp]  
bne[error_NOT_IPV4#]  
  
; check IHL[4:7] is greater than 4 and less than 16 long words.  
alu_shf[ipIhl, 0xf, AND, ipHdr_0, >>24]  
alu[--, ipIhl, -, 0x4]  
blt[error_INVALID_IHL#]  
alu[--, ipIhl, -, 0x10]  
bgt[error_INVALID_IHL#]  
  
; check total len value is greater than 19 bytes or less than or equal  
; to 65,535 bytes. ipLen local register will be used later for length check.  
temp = 0xfffff  
alu_shf[ipLen, temp, AND, ipHdr_0]  
alu[--, ipLen, -, 0x14]  
blt[error_INVALID_LEN#]  
  
; Verify Protocol value is TCP protocol (0x6).  
alu_shf[ipProtocol, 0xff, AND, ipHdr_2, >>16]  
alu[--, ipProtocol, -, 6]  
bne[error_INVALID_PROTOCOL#]  
  
; Perform IP header checksum and compare to header CSum of packet.  
alu[temp, ipHdr_0, +, ipHdr_1]  
alu[temp, temp, +carry, ipHdr_2]  
alu[temp, temp, +carry, ipHdr_3]  
alu[temp, temp, +carry, ipHdr_4]  
alu[temp, temp, +carry, 0]  
  
; Initialize check to 0x0  
immed[check, 0x0]  
; Load upper 16 bits of the result in temp into the upper 16 bits  
; of check. 1100 is used as a byte mask.  
ld_field[check, 1100, temp]  
  
; shift upper 16 bits to lower 16 bits and lower 16 bits to upper 16 bits.  
; Use the shift operation. Rotate will rotate the value in temp 16 bits  
; to the left. Then add rotated value to check value.  
; The combined effect of these two instructions is to add the upper and lower  
; 16 bits of temp together and store the result in check. The result will be  
; all 1's if the IP header has been received without error.  
alu_shf[temp, --, B, temp, <<rot16]  
alu[check, check, +, temp]  
  
; Left shift the contents of check 16 bits and add 1 to it.  
; A valid check would contain all 1's, and adding a 1 to it  
; would produce a carryout.  
alu[temp2, --, B, 1, <<16]  
alu[check, check, +, temp2]  
bcc[error_INVALID_CSUM#]  
  
; If IP CSUM is correct then decrement TTL by one.  
alu_shf[ipTtl, 0xff, AND, ipHdr_2, >>24]  
alu[ipTtl, 1, +, ipTtl]  
temp2 = 0xffffffff
```

```
alu[ipHdr_2, ipHdr_2, AND, temp2]
alu_shf[$out_pkt[0], ipHdr_2, OR, ipTtl, <<24]

// Error processing stubs are shown with no code.
error_NOT_IPV4#:

error_INVALID_IHL#:

error_INVALID_LEN#:

error_INVALID_PROTOCOL#:

error_INVALID_CSUM#:
```

## 5.2 TCP Header Checksum

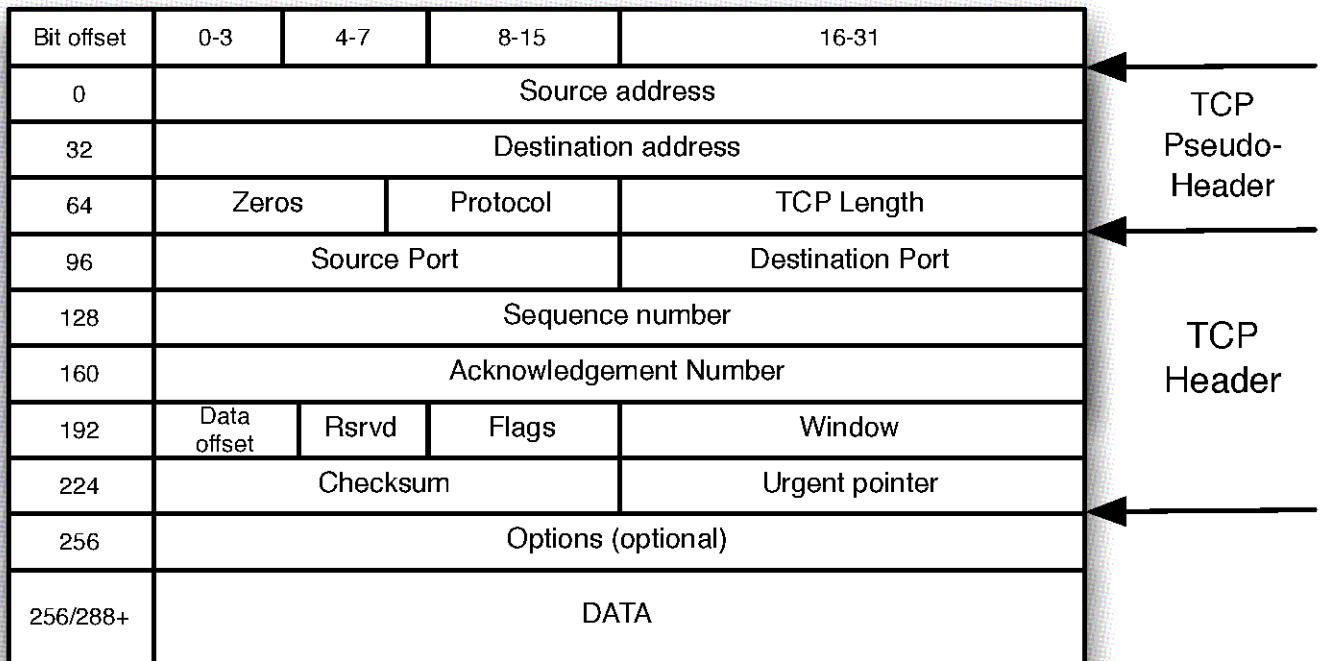
In this example the TCP checksum is calculated and demonstrates using a macro, use of t\_index to select transfer registers and byte\_align\_be[] to retrieve the TCP header aligned for check sum calculation.

The read transfer registers will contain the Ethernet packet (SA, DA, Len/Type fields) followed by IP header (Version, IHL, DSCP, ECN, Total Length, ID, Flags, Frag Offset, TTL, Protocol, Header Checksum, Src IP, Dst IP) followed by the TCP header (Src Port, Dst Port, Sequence No, Ack No, Data offset, Flags, Window, Header Checksum, Urgent Pointer). The Ethernet header contains 14 bytes, IP header contains 20 bytes and TCP header contains 20 bytes (minimum). As one can recognize the Ethernet header does not end on a 32 bit word boundary and when the TCP checksum macro is called the byte\_align\_be[] instruction is used to copy the read transfer registers into the NFP GPR registers aligning the TCP header on a 32 bit boundary. Then the TCP header is in the GPR registers aligned to 32 bit boundary for computing the TCP checksum. The TCP checksum computation continues for the TCP data field.

Byte align operation allows one to load data from read transfer registers into FPC GPR registers starting from any byte not on a 32-bit word boundary. A common function is to read a L2 and L3 header into transfer registers, determine the offset of the L3 header based on the L2 protocol field, and then build a TCP ‘Pseudo Header’ for checksum calculation. The TCP pseudo header contains selected data from both the TCP header and the IP datagram into which the TCP segment will be encapsulated.

Ethernet packet format is shown below where the header is  $6+6+2 = 14$  bytes (not aligned on 32-bit boundary). When FPC thread reads entire packet into the read transfer registers the IP header will start at byte 15, not on a 32-bit boundary. This is where byte\_align\_be[] is used to setup for computing the TCP checksum.

## TCP Pseudo-header for checksum computation (IPv4)



**Figure 5.2. TCP Pseudo-Header Fields**

```

; -----
; FPC 4 thread 0
; -----
; Macro to compute the TCP checksum using Pseudo-Header
; Inputs:
; in_tid - thread ID
; in_pkt - read transfer register holding the packet headers
; in_pkt_l4_offset - offset to where the TCP header begins
; in_protocol - IP protocol ID
; in_tcp_pkt_length - length of TCP packet
; in_sip - IP Source IP (4 bytes)
; in_dip - IP Destination IP (4 bytes)
; Outputs:
; result - Result of TCP checksum generation (0 : pass)
;
#macro _tcp_small_pkt_checksum(in_tid, in_pkt, in_pkt_l4_offset, \
                                in_protocol, in_tcp_pkt_length, in_sip, \
                                in_dip, result)
.begin

    ; Determine the type of transfer register for
    ; byte_align_be[] below.
    #if (strstr(in_pkt, $$))
        #define_eval _XFER_TYPE $$
    #elif (strstr(in_pkt, $))

```

```

#define_eval _XFER_TYPE $
#else
    #error "Transfers register must be prefixed by either $ or $$"
#endif

; Declare local registers
.reg _tc_tmp
.reg _tc_xfer_addr
.reg _tc_cksum
.reg _tc_remaining_bytes
.reg _tc_indirect_shift_reg
.reg tcp_hdr[5]

; Set byte index for byte_align_be[] where nops are not
; required since three alu instructions follow that
; provide the delay to access requirement.
local_csr_wr[BYTE_INDEX, 2]

; Setup transfer register pointer for indirect accesses
; using T_INDEX. This value will be loaded in T_INDEX CSR.
alu[_tc_xfer_addr, --, b, in_tid, <<6]
alu[_tc_xfer_addr, _tc_xfer_addr, or, &$in_pkt[0]]
alu[_tc_xfer_addr, _tc_xfer_addr, +, in_pkt_l4_offset]

; Transfer Index register, used when Transfer registers are accessed
; via indexed mode, which is specified in the src and dest fields of
; the instruction. Used with byte_align_be[] below to specify src
; Transfer register.
local_csr_wr[T_INDEX, _tc_xfer_addr]

; start tcp checksum calculation with IP pseudo-header portion
alu[_tc_cksum, in_sip, +, in_dip]           ; IP SA and DA
alu[_tc_cksum, _tc_cksum, +carry, in_protocol] ; IP pid for TCP
alu[_tc_cksum, _tc_cksum, +carry, in_tcp_pkt_length] ; TCP length
alu[_tc_cksum, _tc_cksum, +carry, 0]
alu[_tc_remaining_bytes, in_tcp_pkt_length, -, 20] ; subtract tcp header size

; Get tcp header - 20 bytes using indirect index method.
; When index++ is executed the T_INDEX increments to next
; transfer register.
byte_align_be[--, *_XFER_TYPE/**/index++]
byte_align_be[tcp_hdr[0], *_XFER_TYPE/**/index++]
byte_align_be[tcp_hdr[1], *_XFER_TYPE/**/index++]
byte_align_be[tcp_hdr[2], *_XFER_TYPE/**/index++]
byte_align_be[tcp_hdr[3], *_XFER_TYPE/**/index++]
byte_align_be[tcp_hdr[4], *_XFER_TYPE/**/index++]

; Add in TCP Header
alu[_tc_cksum, _tc_cksum, +, tcp_hdr[0]]
alu[_tc_cksum, _tc_cksum, +carry, tcp_hdr[1]]
alu[_tc_cksum, _tc_cksum, +carry, tcp_hdr[2]]
br=byte[_tc_remaining_bytes, 0, 0, tspc_finish#], defer[3]
alu[_tc_cksum, _tc_cksum, +carry, tcp_hdr[3]]
alu[_tc_cksum, _tc_cksum, +carry, tcp_hdr[4]]
alu[_tc_cksum, _tc_cksum, +carry, 0]

; Once header is processed, maximum possible bytes
; is 64-14(eth)-20(IP)-20(TCP)=10 bytes

```

```

alu[_tc_remaining_bytes, _tc_remaining_bytes, -, 4]
ble[tspc_le_4_bytes_remaining#], defer[1]
    byte_align_be[_tc_tmp, *_XFER_TYPE/**/index++]; 4 bytes more
alu[_tc_cksum, _tc_cksum, +, _tc_tmp]
alu[_tc_cksum, _tc_cksum, +carry, 0]

alu[_tc_remaining_bytes, _tc_remaining_bytes, -, 4]
ble[tspc_le_4_bytes_remaining#], defer[1]
    byte_align_be[_tc_tmp, *_XFER_TYPE/**/index++]; 4 bytes more
alu[_tc_cksum, _tc_cksum, +, _tc_tmp]
alu[_tc_cksum, _tc_cksum, +carry, 0]

alu[_tc_remaining_bytes, _tc_remaining_bytes, -, 4]
byte_align_be[_tc_tmp, *_XFER_TYPE/**/index++]

tspc_le_4_bytes_remaining#:
; 4 bytes read into _tc_tmp; may not need all four
; need to shift by 4-remaining_bytes
; A operand contains indirect shift value in lowest 5 bits

; _tc_remaining_bytes should be negative
alu[_tc_indirect_shift_reg, 0, -, _tc_remaining_bytes]
; convert to bits for shifting
alu[_tc_indirect_shift_reg, --, b, _tc_indirect_shift_reg, <<3]
alu[--, _tc_indirect_shift_reg, or, 0]
; use shift to clear bytes to 0
alu[_tc_tmp, _tc_indirect_shift_reg, b, _tc_tmp, >>indirect]
; shift back to proper position
alu[_tc_tmp, --, b, _tc_tmp, <<indirect]
alu[_tc_cksum, _tc_cksum, +, _tc_tmp] ; final add
alu[_tc_cksum, _tc_cksum, +carry, 0]

tspc_finish#:
alu[_tc_tmp, --, b, _tc_cksum, >>16]
alu[_tc_cksum, _tc_tmp, +16, _tc_cksum]
alu[_tc_tmp, --, b, _tc_cksum, >>16]
alu[_tc_cksum, _tc_tmp, +16, _tc_cksum]
alu[_tc_tmp, _tc_cksum, +, 1]
alu[_tc_tmp, --, b, _tc_tmp, <<16]

; Check if alu operation results in ALU CC Z bit set.
bne[TCP_CHECKSUM_ERROR#]
result = 0
br[DONE#]

TCP_CHECKSUM_ERROR#:
result = 1

DONE#:
.end
#endifm

// Main code
begin#:
.begin
.if(ctx() == 0)

.sig s1

```

```
.reg in_tid
.reg $in_pkt[18]
.xfer_order $in_pkt
.reg in_pkt_l4_offset
.reg in_tcp_pkt_length
.reg in_sip
.reg in_dip
.reg data
.reg r_mem_addr_ulw r_mem_addr_llw
.reg in_protocol

; Read memory back into read transfer registers.
; Use NFP-6xxx indirect_ref to override length.
; Set prev_alu data[7] to override Length and load length
; value in data[12:8]. The alu[] instruction will put 0x780 in the
; prev_alu register for mem[read, ...], indirect_ref instruction.
; prev_alu.Length => CPP LENGTH field, read 8 64-bit words.
data = 0x780
alu[--, --, B, data]
mem[read, $in_pkt[0], r_mem_addr_ulw, <<8, r_mem_addr_llw, max_8], \
    indirect_ref, ctx_swap[s1]

; Set the active context number to select group of transfer registers.
; For this example set it to 0x0.
; See NFP-6xxx Databook "Registers Used By Contexts in Context-Relative
; Addressing Mode" Table for Context Number and Transfer Register Number.
in_tid = 0x0

; Prepare pseudo header. The IP data was retrieved earlier.
; L4 Header offset from Ethernet DA in read transfer register is 8.
; TCP packet length is 30 bytes. IP Protocol ID is 6 for TCP.
in_pkt_l4_offset = 8
in_tcp_pkt_length = 30
in_protocol = 6
; Offset to 4 bytes boundary
alu[in_pkt_l4_offset, --, B, in_pkt_l4_offset, <<2]

; Setup to get Src and Dst IP addresses from read transfer register
; using byte_align_be[]. Write to CSR with byte index of 2
; because the first two bytes in read transfer register $in_pkt[6]
; is IP Hdr checksum. The Src and Dst IP is made up of 4 + 4 bytes
; spread over $in_pkt[6], $in_pkt[7] and $in_pkt[8].
local_csr_wr[BYTE_INDEX, 2]
nop // must place 3 nops for read-to-use latency
nop
nop

; Get Src IP and Dst IP from read transfer registers
byte_align_be[--, $in_pkt[6]]
byte_align_be[in_sip, $in_pkt[7]]
byte_align_be[in_dip, $in_pkt[8]]

; Call macro to compute pkt TCP checksum.
; Arguments are retrieved from the receive packet and
; provided to the macro where TCP checksum is performed.
; in_sip, in_dip and in_tcp_pkt_length and in_protocol are used
; as TCP Pseudo-header.
_tcp_small_pkt_checksum(in_tid, $in_pkt, in_pkt_l4_offset, \
    in_protocol, in_tcp_pkt_length, in_sip, \
```

```
        in_dip, result)

; If result value is 0 then TCP checksum is OK.
; otherwise TCP Header has error.
alu[--, result, -, 1]
br_bset[result, 0, TCP_CSUM_ERROR#]
br[TCP_CSUM_PASS#]

TCP_CSUM_ERROR#:
; Handle errored Packet

TCP_CSUM_PASS#:
; Process the Packet

.endif
ctx_arb[kill]

.end
```

## ***6. NFP-32xx to NFP-6xxx Coding Recommendations***

---

Moving from NFP-32xx code base to NFP-6xxx please consider the following recommendations.

Global Scratch is removed in NFP-6xxx, and below is a list of recommendations to help you with moving your NFP-32xx code base to NFP-6xxx architecture.

- Use CLS rings instead of global scratch rings wherever possible.
- Use MU rings for global rings, only for global communication.
- Do not poll a ring instead use the Work Queues in the MU. Polling rings in the Scratch was deprecated in the NFP-32xx and is not supported in the NFP-6xxx.

## Glossary

---

_IMM	Use immediate data as the source for the current Flow Processor Core instruction.
_SAT	Saturate, or hold to the limit of the storage (32 or 64 bits). For example: adding 0x10 to 0xFFFF3 will not wrap to 0x0003 but will saturate to 0xFFFF and subtract 0x10 from 0x0005 will saturate to 0x0000.
--	Annotation to a Flow Processor Core instruction that a field is not used.
[width;LSb]	Specification indicating number of bits (width) starting at bit location (LSb), where LSb is least significant bit.
[MSb:LSb]	Specification indicating a field's makeup in bits, starting MSb position to ending LSb position.
0x	Specifies that the number is in hexadecimal format.
action	Subtype of a command issued over the CPP bus (e.g. mem[read e.g. MEM is the command and read is action"] the 40-bit CPP address on the target where the current command is to be performed.
ALU	ALU Arithmetic Logic Unit, the computational part of a microprocessor.
Atomic	Atomic Operation that completes before any other operation.
byte_mask	An array of bits representing what bytes are to be operated on.
BYTE_SWAP	Assertion to enable byte swapping forwarded to a memory operation.
CAM	CAM Content Addressable Memory. A associative memory that returns the address of stored data.
CC	See: Condition Code
CLS	Cluster Local Scratch
Cluster	A grouping of components generally on a silicon island
Condition Code	Results of an ALU expression: Negative (N), Zero (Z), Carry Out (C), and Overflow(V) that can be used in subsequent instructions (e.g. BGT)
context	State within a Flow Processor Core, that represents a single logical thread of control.
control store	The Flow Processor Core instruction memory
CPP	Command/Push/Pull bus
CRC	Cyclic Redundancy Check

CSR	Control and Status Registers
CT	Cluster Target
CTM	Cluster Target Memory is an intelligent memory, accessed through a CPP operation. CTM exists on any island that also has FPCs
CTM_ADDRESS	Cluster Target Memory address, typically the object of a CPP or DMA operation
CTX	Context- logical thread of control such as in a Flow Processor Core
ctx_swap	An optional token in a Flow Processor Core instruction to voluntarily yield the current thread control to the scheduler.
ctx_swap[sig]	Assembler instruction optional token that schedules the thread to sleep until [sig] is asserted
data_master	Initiator of a I/O (CPP) command
data_ref	Portion of an I/O address that may describe data elements.
defer[1-3]	A modifier token to a Flow Processor Core ALU to not flush instructions in the pipeline during a branch operation.
deprecate	Obsolete, not tested and not supported - may still work
Done	Operation completed
DRAM	Dynamic Random Access Memory, used for bulk storage and controlled by an EMEM controller.
DSF	Distributed Switch Fabric
ECC	Error Correction Code data and techniques that can detect and correct errors in data.
EMEM	External memory - DDR3
EOP	End of Packet
EMA	External memory address, including island and locality, for the DMA. SIZE - Size of the DMA transfer (in units of 64-bytes).
FIFO	First In First Out
GPIO	General Purpose I/O
GPR	General Purpose Register
Hash	To reduce data to a reproducible pseudo-random value.
HWM	High Water Mark

I/O	Input / Output
ignore_data_error	An optional token to a Flow Processor Core instruction. (Deprecated, not used any more)
ILA	Interlaken Look-Aside
IMEM	NFP-6xxx Internal Memory
Immediate	A constant value encoded into an assembler instruction.
IND_CSR	The indirect CSR register (CMD_INDIRECT_REF_0) is used for override fields that do not fit in PREV_ALU. The CSR only needs to be written if the fields are enabled by override bits in PREV_ALU.
indirect address	Offset to a register that is used to point at an address
indirect reference mode	A modifier to a Flow Processor Core I/O command that allows for the presentation of more options by previous instruction
indirect_ref	An optional token to a Flow Processor Core, see indirect reference mode
indirect_target	Optional token that describes which external Flow Processor Core to signal
int	Internal, usually used to specify the address is to be interpreted by the internal address map of the target.
Island	A grouping of components that are largely isolated on the silicon (see also cluster)
label	A logical notion often represented with data, signal or address
LM	See: local memory
local memory	Memory exclusively available to a particular Flow Processor Core
LRU	Least recently used
LSB/LSb	Least Significant Byte/bit
master_island	CPP master that issued the first command that may generate one or more bus operations (see data_master)
ME	Micro Engine or Flow Processor Core - a small, yet powerful microprocessor
Mesh	Set of routed buses across a chip, with an 8-way route node in each island with 6 connections to other islands.
MEv2.7	MicroEngine used on NFP-32xx
MEv2.8	FPC used on NFP-6xxx
MRU	Most recently used

MSB/MSb	Most Significant Byte/bit
multi-port memory	Multi-ported memory memory that can simultaneously serve multiple memory cycles. e.g. Simultaneous read and write over multiple buses, such as the Cluster Target Memory
NBI	Network Block Interface, also called Packet Processing Subsystem
Neighbor bus	A direct and unidirectional path between FPCs, often used to pass state information.
NFAS	Network Flow Assembler
NFLD	Network Flow Linker
NFP	Netronome Network Flow Processor
no_pull	Replaces pull transfer register contents in memory writes with results from previous ALU operation.
octet	An 8-bit word, also known as a byte.
override	A description of how default information can be modified with exceptional data. Commonly see override used with instructions specified with indirect_ref opt_tok.
PCC	Predicate Condition Code - state that is passed from one ALU operation to modify another instruction. Controls write-back to the destination register based on the condition code generated by the instruction.
PCIe	PCI Express
PPID	Push Pull bus Identifier
Predicate Condition Code	Compare CC with PredCondCodes CSR and if they match, complete the I/O operation
predicate_cc	Predicate Condition Code token
PREV_ALU	The ALU result from a previous Flow Processor Core instruction; always used with indirect_ref token
Q_link	Pointer to next element in a linked list
RC	RC is used for register reset specifications.
RC	RC is used to indicate “Root Complex” on the PCIe bus.
ref_cnt	Flow Processor Core I/O instruction, the number of additional consecutive transfer registers to be used (encoded 0=1 and 15=16)
Requestor ID	The external PCIe bridge identifier for the source of the transaction request.

Reset	Return to known state
rid	see Requestor ID
Ring	A buffer that recurses on itself, that is, created to look logically like a circular buffer on a standard linear-addressed memory space.
RO	Read only. Write may result in unpredictable behavior.
RW	Read and Write.
RW1C	Read and Write 1 to Clear.
SA	Source Address
Scratch	SCRATCH (in an FPC)
SCS	Shared control Store allowing two Flow Processor Core to share the same instruction memory, providing double the available number of instructions for each core.
sig_done	an optional token indicating to the Flow Processor Core scheduler that the current thread should be suspended until the specified signal is asserted.
sig_remote	Optional token that indicates that a signal should be sent to another thread when the operation is completed. May be obsolete as it was only used in CAP instructions in the prior chip. (obsolete - DEPRICATE)
signal_master	Is the master ID of the master to which the signal is to be sent. Reference within the signal_master as to which signal should be indicated with the commands push or pull. This consists of a 3-bit context followed by a 4-bit signal number.
signal_name	Signal label tied to one of 15 eligible signals that are passed over the I/O DSF (formerly CPP bus)
SOP	Start of Packet
SRAM	Static Random Access Memory
State	State a representation of logical information; such as the status of a packet
swap 1	Swap to atomically exchange transfer register with value in memory (e.g. mem[swap, reg_A, reg_B, 1])
swap context	Mark the Flow Processor Core context inactive (e.g. mem[read, ..], ctx_swap[sig] ; wait until 'sig' is asserted), allowing the next context to become active.
target 1	remote storage or component that is the object of an operation.
target 2	Within the DSF the taget addess

Target	Component connected to the CPP which accepts commands; in response to commands data is pulled or pushed from/to a master.
TAT_CSR	The Target Address Token CSR used for testing. It contains a set of override bits and fields for the target section and token command parameters.
TCAM	Ternary CAM - Search Function that returns match, no match or undefined.
token 0	A logical notion often represented with data or a signal; see also label
token 1	Token optional operand indicating a change to be performed by the assembler (e.g. defer[])
token 2	Subtype of an action on the CPP bus (e.g. sig_done) or modifier applies to a Flow Processor Core instruction
Transfer Register	Transfer Registers are Flow Processor Core registers that are exposed to the rest of the system and can be a source or target for I/O commands. Transfer registers have an absolute address and relative address, based on number of Flow Processor Core contexts that are configured.
UART	Universal Asynchronous Receiver/Transmitter
UDR	Unlinked Descriptor Read
undef	Undefined for read
W1C	Write 1 to clear
W1S	Write 1 to set
WO	Write only. Read may result in unpredictable behavior.
WRC	Write and Clear on Read

## 7. Technical Support

---

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