
ST40 core and instruction set architecture

Introduction

The ST40 is a 32-bit RISC (reduced instruction set computer) microprocessor. It includes separate instruction and operand caches. The operand caches support both copy-back and write-through modes.

The 16-bit fixed-length instruction set gives the architecture extremely good code density.

The ST40 core products are grouped into series. These series can be distinguished by the following features.

- The associativity of the cache architecture.
- The presence of index and RAM modes in the cache architecture.
- The presence of a floating point unit (FPU). This provides hardware support for scalar and vector floating point computation.
- The presence of a memory management unit (MMU). This provides hardware support for translation and protection of parts of the address space. It incorporates a 64 entry unified translation lookaside buffer (TLB) and a 4 entry instruction TLB which caches entries from the unified TLB without software intervention.
- The width of physical addressing used on the STBus. Earlier ST40 cores support 29-bit physical addressing. Newer ST40 cores can also support 32-bit physical addressing.

Naming conventions

| | |
|---------------------|---|
| ST40 | for information common to all products. |
| ST40-100/200 series | for series specific features. |
| ST40-103/202 | for variant specific features. |

Contents

| | |
|--|-----------|
| Preface | 10 |
| Document identification and control | 10 |
| ST40 documentation suite | 10 |
| Conventions used in this guide | 11 |
| 1 Overview | 12 |
| 1.1 ST40 core features | 12 |
| 1.2 Block diagram | 17 |
| 2 Programming model | 18 |
| 2.1 General registers | 19 |
| 2.2 System registers | 20 |
| 2.3 Control registers | 24 |
| 2.4 Floating-point registers | 26 |
| 2.5 Memory-mapped control registers | 28 |
| 2.6 Data format in registers | 28 |
| 2.7 Data formats in memory | 28 |
| 2.8 Processor states | 29 |
| 2.8.1 Reset state | 29 |
| 2.8.2 Reset address | 29 |
| 2.8.3 Exception-handling state | 29 |
| 2.8.4 Program execution state | 30 |
| 2.8.5 Power-down state | 31 |
| 2.9 Processor modes | 31 |
| 3 Memory management unit (MMU) | 32 |
| 3.1 Terminology | 32 |
| 3.2 ST40 MMU variants | 33 |
| 3.3 Role of the MMU | 34 |
| 3.4 Physical address space | 37 |
| 3.4.1 29-bit physical address space | 37 |
| 3.4.2 32-bit physical address space (space-enhancement mode) | 38 |

| | | |
|--------|--|----|
| 3.4.3 | Overview of mode switching | 39 |
| 3.4.4 | Switching into space-enhancement mode | 40 |
| 3.4.5 | Switching out of space-enhancement mode | 40 |
| 3.4.6 | Limitations on some ST40-200 series parts | 40 |
| 3.4.7 | Effect of mode switching on the UTLB and ITLB | 40 |
| 3.4.8 | Effect of mode switching on the instruction cache | 40 |
| 3.4.9 | Effect of mode switching on the operand cache | 41 |
| 3.4.10 | Effect of mode switching on bus operations | 41 |
| 3.4.11 | Mode switching when the physical address of the PC changes | 42 |
| 3.4.12 | Examples of switching into space-enhancement mode | 43 |
| 3.4.13 | Example of switching out of space-enhancement mode | 44 |
| 3.5 | Virtual addresses | 45 |
| 3.5.1 | MMU operating modes | 45 |
| 3.5.2 | Virtual address regions | 46 |
| 3.5.3 | Virtual to physical address mapping | 46 |
| 3.5.4 | Multiple virtual address spaces | 52 |
| 3.6 | Hardware and software functions | 53 |
| 3.6.1 | Hardware functions | 53 |
| 3.6.2 | Software functions | 53 |
| 3.7 | P4 address region | 54 |
| 3.7.1 | Resources accessible only through P4 | 54 |
| 3.7.2 | Resources accessible through P4 and through translations | 54 |
| 3.7.3 | P4 virtual addressing structure | 55 |
| 3.8 | Register descriptions | 57 |
| 3.8.1 | Page table entry high register (PTEH) | 57 |
| 3.8.2 | Page table entry low register (PTL) | 59 |
| 3.8.3 | Translation table base register (TTB) | 62 |
| 3.8.4 | TLB exception address register (TEA) | 63 |
| 3.8.5 | MMU control register (MMUCR) | 63 |
| 3.8.6 | Physical address space control register (PASCR) | 66 |
| 3.8.7 | Instruction refetch inhibit control (IRMCR) | 68 |
| 3.9 | Unified TLB (UTLB) | 69 |
| 3.9.1 | UTLB operations | 70 |
| 3.9.2 | UTLB address array | 72 |
| 3.9.3 | UTLB data array | 74 |

| | | |
|----------|--|------------|
| 3.10 | Instruction TLB (ITLB) | 79 |
| 3.10.1 | ITLB operations | 79 |
| 3.10.2 | ITLB LRU state | 81 |
| 3.10.3 | ITLB address array | 82 |
| 3.10.4 | ITLB data array | 84 |
| 3.11 | Privileged mapping buffer (PMB) | 88 |
| 3.11.1 | PMB operations | 89 |
| 3.11.2 | PMB address array | 89 |
| 3.11.3 | PMB data array | 91 |
| 3.12 | Handling MMU exceptions | 94 |
| 3.12.1 | ITLBMULTIHIT | 95 |
| 3.12.2 | ITLBMISS | 95 |
| 3.12.3 | EXECPROT | 96 |
| 3.12.4 | OTLBMULTIHIT | 96 |
| 3.12.5 | WTLBMISS/RTLBMISS | 96 |
| 3.12.6 | READPROT/WRITEPROT | 97 |
| 3.12.7 | FIRSTWRITE | 97 |
| 3.13 | Interaction of MMU and cache | 98 |
| 3.14 | Memory coherency | 99 |
| 3.14.1 | Coherency mechanisms on ST40-100, ST40-200, ST40-400 and ST40-500 series cores | 99 |
| 3.14.2 | Coherency mechanisms on ST40-300 series cores | 103 |
| 3.15 | MMU state coherency | 105 |
| 3.15.1 | Coherency requirements | 105 |
| 3.15.2 | Achieving coherency (ST40-100, ST40-200, ST40-400 and ST40-500 series cores) | 106 |
| 3.15.3 | Achieving coherency (ST40-300 series cores) | 106 |
| 3.16 | Side effects of instruction pre-fetch | 107 |
| 3.16.1 | Inline fetching near ends of memory regions | 107 |
| 3.16.2 | Branches following exceptions | 108 |
| 4 | Caches | 114 |
| 4.1 | Overview | 114 |
| 4.1.1 | Features | 114 |
| 4.1.2 | Cache modes | 114 |
| 4.1.3 | Store queues | 115 |

| | | |
|-------|--|-----|
| 4.2 | Register descriptions | 116 |
| 4.2.1 | Cache control register (CCR) | 116 |
| 4.2.2 | Queue address control register 0 (QACR0) | 119 |
| 4.2.3 | Queue address control register 1 (QACR1) | 120 |
| 4.2.4 | On-chip memory control register (RAMCR) | 120 |
| 4.3 | Operand cache (OC) | 121 |
| 4.3.1 | Configuration | 121 |
| 4.3.2 | Read operation | 123 |
| 4.3.3 | Write operation | 124 |
| 4.3.4 | Write-back buffer | 125 |
| 4.3.5 | Write-through buffer | 125 |
| 4.3.6 | RAM mode | 125 |
| 4.3.7 | OC index mode | 127 |
| 4.3.8 | Explicit cache controls | 127 |
| 4.4 | Instruction cache (IC) | 128 |
| 4.4.1 | Configuration | 128 |
| 4.4.2 | Read operation | 129 |
| 4.4.3 | IC index mode | 130 |
| 4.4.4 | Explicit cache controls | 130 |
| 4.5 | Memory-mapped cache configuration | 131 |
| 4.5.1 | IC address array | 133 |
| 4.5.2 | IC data array | 135 |
| 4.5.3 | OC address array | 136 |
| 4.5.4 | OC data array | 138 |
| 4.6 | Store queues (SQs) | 140 |
| 4.6.1 | Power-save functions | 141 |
| 4.6.2 | SQ configuration | 141 |
| 4.6.3 | SQ writes | 141 |
| 4.6.4 | SQ reads | 142 |
| 4.6.5 | Transfer to external memory | 142 |
| 4.6.6 | Physical address for external transfer | 143 |
| 4.6.7 | Access rights for store queue operations | 144 |
| 4.7 | Cache state coherency | 145 |

| | | |
|----------|---|------------|
| 5 | Exceptions | 146 |
| 5.1 | Register descriptions | 146 |
| 5.1.1 | Exception event register (EXPEVT) | 146 |
| 5.1.2 | Interrupt event register (INTEVT) | 147 |
| 5.1.3 | TRAPA exception register (TRA) | 147 |
| 5.2 | Exception handling functions | 148 |
| 5.2.1 | Exception handling flow | 148 |
| 5.2.2 | Exception handling vector addresses | 148 |
| 5.3 | Exception types and priorities | 149 |
| 5.4 | Exception flow | 150 |
| 5.4.1 | Exception source acceptance | 151 |
| 5.4.2 | Exception requests and BL bit | 152 |
| 5.4.3 | Return from exception handling | 153 |
| 5.5 | Description of exceptions | 153 |
| 5.5.1 | Resets | 153 |
| 5.5.2 | General exceptions | 156 |
| 5.5.3 | Interrupts | 159 |
| 5.5.4 | Priority order with multiple exceptions | 160 |
| 5.6 | Usage notes | 161 |
| 6 | Floating-point unit | 163 |
| 6.1 | Floating-point format | 163 |
| 6.1.1 | Non-numbers (NaN) | 165 |
| 6.1.2 | Denormalized numbers | 165 |
| 6.2 | Rounding | 166 |
| 6.3 | Floating-point exceptions | 166 |
| 6.4 | Graphics support functions | 168 |
| 6.4.1 | Geometric operation instructions | 168 |
| 6.5 | 64-bit data transfer | 169 |
| 6.5.1 | Register-to-register transfers | 169 |
| 6.5.2 | Memory transfers | 170 |

| | | |
|----------|--------------------------------------|------------|
| 7 | Instruction set | 172 |
| 7.1 | Execution environment | 172 |
| 7.2 | Addressing modes | 173 |
| 7.3 | Instruction set summary | 177 |
| 8 | Instruction specification | 189 |
| 8.1 | Variables and types | 189 |
| 8.1.1 | Integer | 189 |
| 8.1.2 | Boolean | 190 |
| 8.1.3 | Bit-fields | 190 |
| 8.1.4 | Arrays | 190 |
| 8.1.5 | Floating point values | 190 |
| 8.2 | Expressions | 191 |
| 8.2.1 | Integer arithmetic operators | 191 |
| 8.2.2 | Integer shift operators | 192 |
| 8.2.3 | Integer bitwise operators | 192 |
| 8.2.4 | Relational operators | 193 |
| 8.2.5 | Boolean operators | 193 |
| 8.2.6 | Single-value functions | 194 |
| 8.3 | Statements | 196 |
| 8.3.1 | Undefined behavior | 196 |
| 8.3.2 | Assignment | 196 |
| 8.3.3 | Conditional | 197 |
| 8.3.4 | Repetition | 197 |
| 8.3.5 | Exceptions | 198 |
| 8.3.6 | Procedures | 198 |
| 8.4 | Architectural state | 199 |
| 8.5 | Memory model | 200 |
| 8.5.1 | Support functions | 201 |
| 8.5.2 | Instruction fetch | 202 |
| 8.5.3 | Reading memory | 202 |
| 8.5.4 | Prefetching memory | 203 |
| 8.5.5 | Writing memory | 204 |
| 8.6 | Sleep and synchronization operations | 205 |
| 8.7 | Cache model | 205 |

| | | |
|-----------|--|------------|
| 8.8 | Floating-point model | 206 |
| 8.8.1 | Functions to access SR and FPSCR | 206 |
| 8.8.2 | Functions to model floating-point behavior | 207 |
| 8.8.3 | Floating-point special cases and exceptions | 209 |
| 8.9 | Abstract sequential model | 209 |
| 8.10 | Example instructions | 210 |
| 8.10.1 | ADD #imm, Rn | 210 |
| 8.10.2 | FADD FRm, FRn | 211 |
| 9 | Instruction descriptions | 213 |
| 9.1 | Alphabetical list of instructions | 213 |
| 10 | ST40-100, 200, 400, 500 performance characteristics | 474 |
| 10.1 | Pipelines | 474 |
| 10.2 | Parallel executables | 479 |
| 10.3 | Execution cycles and pipeline stalling | 482 |
| 11 | ST40-300 performance characteristics | 497 |
| 11.1 | Basic pipeline structure | 497 |
| 11.2 | Issue constraints | 498 |
| 11.3 | Instruction performance characteristics | 499 |
| 11.3.1 | Integer instructions | 500 |
| 11.3.2 | Floating point instructions | 505 |
| 11.3.3 | Branching scenarios | 508 |
| 12 | User break controller (UBC) | 510 |
| 12.1 | Overview | 510 |
| 12.1.1 | Features | 510 |
| 12.1.2 | Block diagram | 511 |
| 12.2 | Register overview | 512 |
| 12.3 | Register descriptions | 513 |
| 12.3.1 | Access to UBC control registers | 513 |
| 12.3.2 | Break address register A (UBC.BARA) | 513 |
| 12.3.3 | Break ASID register A (UBC.BASRA) | 514 |
| 12.3.4 | Break address mask register A (UBC.BAMRA) | 514 |
| 12.3.5 | Break bus cycle register A (UBC.BBRA) | 516 |

| | | |
|--------------------------------------|--|------------|
| 12.3.6 | Break address register B (UBC.BARB) | 517 |
| 12.3.7 | Break ASID register B (UBC.BASRB) | 517 |
| 12.3.8 | Break address mask register B (UBC.BAMRB) | 517 |
| 12.3.9 | Break data register B (UBC.BDRB) | 517 |
| 12.3.10 | Break data mask register B (UBC.BDMRB) | 518 |
| 12.3.11 | Break bus cycle register B (UBC.BBRB) | 518 |
| 12.3.12 | Break control register (UBC.BRCR) | 518 |
| 12.4 | Operation | 520 |
| 12.4.1 | Explanation of terms relating to accesses | 520 |
| 12.4.2 | Explanation of terms relating to instruction intervals | 521 |
| 12.4.3 | User break operation sequence | 521 |
| 12.4.4 | Instruction access cycle break | 522 |
| 12.4.5 | Operand access cycle break | 523 |
| 12.4.6 | Condition match flag setting | 524 |
| 12.4.7 | Program counter (PC) value saved | 525 |
| 12.4.8 | Contiguous A and B settings for sequential conditions | 526 |
| 12.5 | Usage notes | 527 |
| 12.6 | User break debug function | 529 |
| 12.7 | Examples of use | 530 |
| 12.7.1 | Instruction access cycle break condition settings | 530 |
| 12.7.2 | Operand access cycle break condition settings | 533 |
| 12.7.3 | User break controller stop function | 534 |
| Appendix A Address list | | 535 |
| Revision history | | 536 |

Preface

Comments on this manual should be made by contacting your local STMicroelectronics sales office or distributor.

Document identification and control

Each book carries a unique identifier of the form:

nnnnnnnn Rev x

where *nnnnnnnn* is the document number, and *x* is the revision.

Whenever making comments on this document, quote the complete identification *nnnnnnnn Rev x*.

ST40 documentation suite

The ST40 documentation suite comprises the following volumes:

32-Bit RISC Series, ST40 Core Architecture Manual (7182230)

This manual describes the architecture and instruction set of the ST40 (previously known as ST40-C200) core as used by STMicroelectronics.

ST40 Core Support Peripherals Manual (7988763)

This manual provides details of the architecture of the ST40 core support peripherals which include: *Clock, power and reset control (CPRC)*, *Real-time clock (RTC)*, *Timer unit (TMU)*, *Serial comms interface with FIFO (SCIF)*, and the *Interrupt controller (INTC)*.

The manual also describes the peripheral bridge that acts as the interface between the peripherals and the STBus. This is achieved by grouping the modules into a single port on the STBus, and handling any data size, data rate and endianness issues transparently.

ST40-300 Core Support Peripherals Manual (8011247)

This manual provides details on the architecture of the ST40-300 core support peripherals which includes: *Clock, power and reset control (CPRC)*, *Timer unit (TMU)*, *User debug interface (UDI)* and the *Interrupt controller (INTC)*.

This manual also describes the peripheral bridge that acts as the interface between the peripherals and the STBus.

ST40 Micro Toolset User Guide (7379953)

This manual describes the ST40 Micro Toolset and provides an introduction to OS21. It covers the various code and cross development tools that are provided in the toolset, how to boot OS21 applications from ROM and how to port applications which use STMicroelectronics' OS20 operating systems. Information is also given on how to build the open source packages that provide the compiler tools, base run-time libraries and debug tools and how to set up an ST Micro Connect.

OS21 User Manual (7358306)

This manual describes the generic use of OS21 across the supported platforms. It describes all the core features of OS21 and their use and details the OS21 function definitions. It also explains how OS21 differs from OS20, the API targeted at ST20 platforms.

OS21 for ST40 User Manual (7358673)

This manual describes the use of OS21 on ST40 platforms. It describes how specific ST40 facilities are exploited by the OS21 API. It also describes the OS21 board support packages for ST40 platforms.

Conventions used in this guide

General notation

The notation in this document uses the following conventions:

- *sample code*, *keyboard input* and *file names*,
- *variables*, *code variables* and *code comments*,
- *equations* and *math*,
- **screens**, **windows**, **dialog boxes** and **tool names**,
- **instructions**.

Hardware notation

The following conventions are used for hardware notation:

- REGISTER NAMES and FIELD NAMES,
- PIN NAMES and SIGNAL NAMES.

1 Overview

This chapter provides an overview of the ST40.

1.1 ST40 core features

This manual describes the architecture of the ST40 core. The core is a highly encapsulated design component that can be integrated into any product. There are therefore no references to clock speeds, system facilities, pin-outs or similar data in this manual. This information is contained in the product's *Datasheet*, *System Architecture Manual* or *Core Support Package Architecture Manual* (where applicable).

[Table 1](#) summarizes the differences between each series.

Table 1. Summary of differences between ST40 product series

| Series | Cache associativity | FPU | MMU | 32-bit physical addressing | Examples |
|--------|------------------------|----------|-----|----------------------------|----------|
| 100 | Direct-mapped | Yes | Yes | No | ST40-103 |
| 200 | Two-way ⁽¹⁾ | Yes | Yes | Some ⁽²⁾ | ST40-202 |
| 300 | Two-way ⁽³⁾ | Optional | Yes | Yes | ST40-300 |
| 400 | Two-way | No | No | No | ST40-400 |
| 500 | Two-way | No | Yes | Yes | |

1. The ST40-202 includes a compatibility mode which halves the sizes of the caches and causes them to operate with direct-mapped associativity.
2. Depends on how the ST40-202 has been integrated into the specific product.
3. Extra option to use four-way on later implementations.

Most soft core products support configurable cache sizes (in the range 4 Kbytes to 64 Kbytes, with it being possible to set the sizes of the instruction and operand independently). Refer to the product datasheet for further details.

Some differences are specific to particular products. For example, the ST40-202 has a fixed cache configuration (a 16 Kbyte 2-way instruction cache and a 32 Kbyte 2-way operand cache) and has a compatibility mode that causes it to behave as a 16 Kbyte direct mapped operand cache and an 8 Kbyte direct mapped instruction cache (as found in the ST40-103). All such product-specific architectural features are described in this document, and the different variants to which they apply are clearly indicated.

The features of the ST40 CPU core are summarized in the following list.

CPU

- Original Renesas architecture
- 32-bit internal data bus
- General register file:
 - sixteen 32-bit general registers (and eight 32-bit shadow registers)
 - seven 32-bit control registers
 - four 32-bit system registers
- RISC-type instruction set
 - fixed 16-bit instruction length for improved code efficiency
 - load-store architecture
 - delayed branch instructions
 - conditional execution
- Superscalar architecture: parallel execution of two instructions
- Instruction execution time: maximum 2 instructions/cycle
- On-chip multiplier
- Five-stage pipeline (100, 200, 400 and 500 series)
- Seven-stage pipeline (300 series)
- Branch target cache for bubble-free branching along predicted path (300 series)

FPU (if present)

- On-chip floating-point co-processor
- Supports single-precision (32 bits) and double-precision (64 bits)
- Supports IEEE754-compliant data types and exceptions
- Two rounding modes: round to nearest and round to zero
- Handling of denormalized numbers: truncation to zero or interrupt generation for compliance with IEEE754
- Floating-point registers:
 - 2 banks of sixteen 32-bit single precision registers or
 - 2 banks of eight 64-bit double precision registers or
 - 2 banks of four 128-bit vector registers (each vector is 4 single precision elements)
- 32-bit CPU-FPU floating-point communication register (FPUL)
- Supports FMAC (multiply-and-accumulate) instruction
- Supports FDIV (divide) and FSQRT (square root) instructions
- Supports FLDI0/FLDI1 (load constant 0/1) instructions
- Instruction execution times (100 and 200 series):
 - latency (FMAC/FADD/FSUB/FMUL): 3 cycles (single-precision), 8 cycles (double-precision)
 - pitch (FMAC/FADD/FSUB/FMUL): 1 cycle (single-precision), 6 cycles (double-precision)
- Instruction execution times (300 series):
 - FMAC/FADD/FSUB: 1 cycle pitch, 6 cycles latency (single and double precision)
 - FMUL (single precision): 1 cycle pitch, 6 cycles latency
 - FMUL (double precision): 4 cycles pitch, 10 cycles latency

Note: FMAC is supported for single-precision only.

- 3-D graphics instructions (single-precision only):
 - 4-dimensional vector conversion and matrix operations (FTRV):
4 cycles (pitch), 7 cycles (latency) (100 and 200 series)
4 cycles (pitch), 10 cycles (latency) (300 series)
 - 4-dimensional vector (FIPR) inner product:
1 cycle (pitch), 4 cycles (latency) (100 and 200 series)
1 cycle (pitch), 6 cycles (latency) (300 series)
- Five-stage pipeline (100 and 200 series)
- Ten-stage pipeline (300 series)
- Single-precision polynomial approximations to sine, cosine and reciprocal square root

Power-down

- Power-down modes:
 - sleep mode
 - standby mode
 - module standby function

MMU (if present)

- 4-Gbyte virtual address space, 256 address space identifiers (8-bit ASIDs)
- Single virtual mode and multiple virtual memory mode
- Supports multiple page sizes: 1 Kbyte, 4 Kbytes, 64 Kbytes, 1 Mbyte
- 4-entry fully-associative TLB for instructions
- 64-entry fully-associative TLB for instructions and operands
- Supports software-controlled replacement and random-counter replacement algorithm
- TLB contents can be accessed directly by address mapping
- 29-bit physical addressing (all series)
- 32-bit physical addressing (200, 300 and 500 series)

Cache memory**ST40-103**

- Instruction cache (IC) features:
 - 8 Kbytes, direct mapping
 - 256 entries, 32-byte block length
 - normal mode (8 Kbyte cache)
 - index mode
- Operand cache (OC) features:
 - 16 Kbytes, direct mapping
 - 512 entries, 32-byte block length
 - normal mode (16 Kbyte cache)
 - index mode
 - RAM mode (8 Kbyte cache + 8 Kbyte RAM)
 - choice of write method (copy-back or write-through)
- Single-stage copy-back buffer, single-stage write-through buffer
- Cache memory contents can be accessed directly by address mapping (usable as on-chip memory)
- Store queue (32 bytes x 2 entries)

ST40-202

- Instruction cache (IC) features:
 - 16 Kbyte, 2-way set associative
 - 512 entries, 32-bytes block length
 - compatibility mode (8 Kbyte direct mapped)
 - index mode
- Operand cache (OC) features:
 - 32 Kbyte, 2-way set associative
 - 1024 entries, 32 bytes block length
 - compatibility mode (16 Kbyte direct mapped)
 - index mode
 - RAM mode (16 Kbyte cache + 16 Kbyte RAM)
- Single-stage copy-back buffer, single-stage write-through buffer
- Cache memory contents can be accessed directly by address. mapping (usable as on-chip memory)
- Store queue (32 bytes x 2 entries).

Note: On the ST40-202 the state of the cache becomes undefined if the OC index mode and OC RAM mode are used together.

ST40-200 series (other than the ST40-202), ST40-400 and ST40-500 series

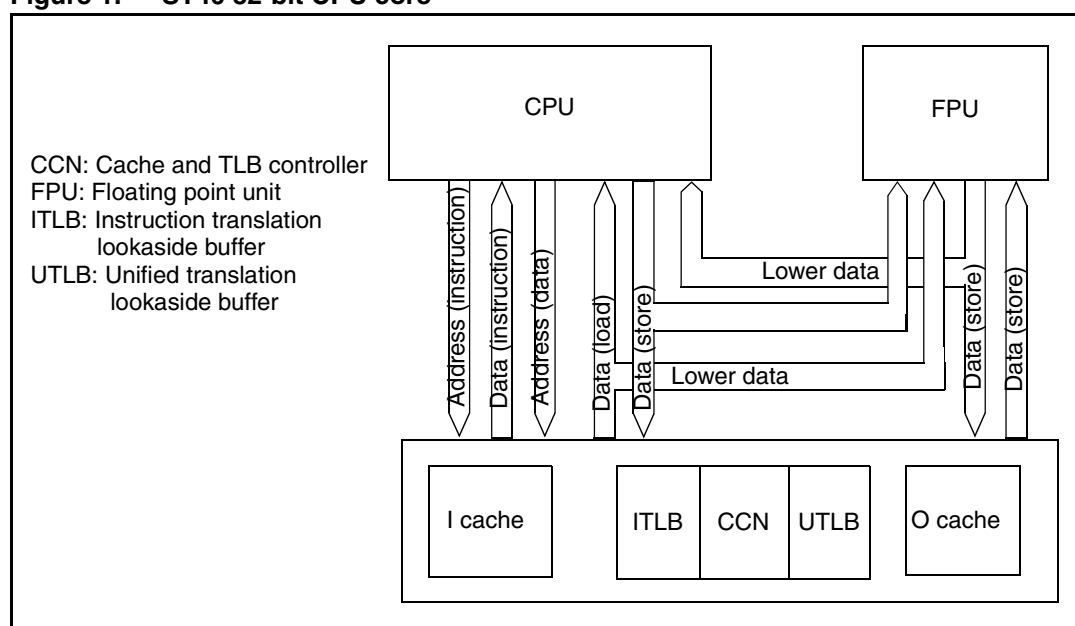
- Instruction cache (IC) features:
 - 4, 8, 16, 32 or 64 Kbyte, 2-way set associative cache
 - 32 bytes block length
 - no compatibility or index mode support
- Operand cache (OC) features:
 - 4, 8, 16, 32 or 64 Kbyte, 2-way set associative cache
 - 32 bytes block length
 - no compatibility mode or index mode support
 - RAM mode (half of each way can be reassigned as RAM)
- Single-stage copy-back buffer, single-stage write-through buffer
- Cache memory contents can be accessed directly by address mapping (usable as on-chip memory)
- Store queue (32 bytes x 2 entries)

- ST40-300 series**
- Instruction cache (IC) features:
 - 4, 8, 16, 32, 64 or 128 Kbyte, 2-way set associative cache
 - option to use 4-way associativity on a future implementation
 - 32 bytes block length
 - no compatibility or index mode support
 - Operand cache (OC) features:
 - 4, 8, 16, 32, 64 or 128 Kbyte, 2-way set associative cache
 - option to use 4-way associativity on a future implementation
 - 32 bytes block length
 - no compatibility mode, index mode or RAM mode support
 - 8-stage memory store/writeback buffer
 - Up to 6 outstanding operand prefetches
 - Store queue (32bytes x 2 entries)

1.2 Block diagram

Figure 1 shows an internal block diagram of the ST40 32-bit CPU core.

Figure 1. ST40 32-bit CPU core



2 Programming model

The ST40 CPU core has two processor modes, user mode and privileged mode. The ST40 normally operates in user mode, and switches to privileged mode when an exception occurs, or an interrupt is accepted.

There are four kinds of registers.

- General registers
There are 16 general registers, R0 to R15. General registers R0 to R7 are banked registers which are switched by a processor mode change.
- System registers
Access to these registers does not depend on the processor mode
- Control registers
- Floating-point registers (only present in products with an FPU)
There are thirty-two floating-point registers, FR0–FR15 and XF0–XF15. FR0–FR15 and XF0–XF15 can be assigned to either of two banks (FPR0_BANK0–FPR15_BANK0 or FPR0_BANK1–FPR15_BANK1).

The registers that can be accessed differ in the two processor modes.

Register values after a reset are shown in [Table 2](#).

Table 2. Initial register values

| Type | Registers | Reset value ⁽¹⁾ |
|---|--|--|
| General registers | R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, R8–R15 | Undefined |
| Control registers | SR | MD=1, RB=1, BL=1, I3-I0=0xF FD=0 - ST40 FPU family), 1 - otherwise reserved bits=0, others undefined |
| | GBR, SSR, SPC, SGR, DBR | Undefined |
| | VBR | 0x0000 0000 |
| System registers | MACH, MACL, PR, FPUL | Undefined |
| | PC | See Section 2.8.2: Reset address on page 29 |
| Floating-point registers (only if FPU present) | FR0–FR15, XF0–XF15 | Undefined |
| | FPSCR | 0x0004 0001 |

1. Initial value after power-on or manual reset.

2.1 General registers

Figure 2 shows the relationship between the processor modes and the general registers. The ST40 CPU core has twenty-four 32-bit general registers (R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, and R8–R15). However, only 16 of these can be accessed as general registers, R0–R15, in either processor mode. The assignment of R0–R7, in both modes, is shown below.

- R0_BANK0–R7_BANK0:
 - in user mode (SR.MD = 0), R0–R7 are always assigned to R0_BANK0–R7_BANK0
 - in privileged mode (SR.MD = 1), R0–R7 are assigned to R0_BANK0–R7_BANK0 only when SR.RB = 0
- R0_BANK1–R7_BANK1:
 - in user mode, R0_BANK1–R7_BANK1 cannot be accessed
 - in privileged mode, R0–R7 are assigned to R0_BANK1–R7_BANK1 only when SR.RB = 1

Figure 2. General registers

| SR.MD = 0 or (SR.MD = 1, SR.RB = 0) | | | (SR.MD = 1, SR.RB = 1) | | |
|--|----------|-----|------------------------|--|--|
| R0 | R0_BANK0 | | R0_BANK | | |
| R1 | R1_BANK0 | | R1_BANK | | |
| R2 | R2_BANK0 | | R2_BANK | | |
| R3 | R3_BANK0 | | R3_BANK | | |
| R4 | R4_BANK0 | | R4_BANK | | |
| R5 | R5_BANK0 | | R5_BANK | | |
| R6 | R6_BANK0 | | R6_BANK | | |
| R7 | R7_BANK0 | | R7_BANK | | |
| R0_BANK | R0_BANK1 | R0 | | | |
| R1_BANK | R1_BANK1 | R1 | | | |
| R2_BANK | R2_BANK1 | R2 | | | |
| R3_BANK | R3_BANK1 | R3 | | | |
| R4_BANK | R4_BANK1 | R4 | | | |
| R5_BANK | R5_BANK1 | R5 | | | |
| R6_BANK | R6_BANK1 | R6 | | | |
| R7_BANK | R7_BANK1 | R7 | | | |
| R8 | R8 | R8 | | | |
| R9 | R9 | R9 | | | |
| R10 | R10 | R10 | | | |
| R11 | R11 | R11 | | | |
| R12 | R12 | R12 | | | |
| R13 | R13 | R13 | | | |
| R14 | R14 | R14 | | | |
| R15 | R15 | R15 | | | |

Programming note

As the user's R0–R7 are assigned to R0_BANK0–R7_BANK0, and after an exception or interrupt R0–R7 are assigned to R0_BANK1–R7_BANK1, it is not necessary for the interrupt handler to save and restore the user's R0–R7 (R0_BANK0–R7_BANK0).

After a reset, the values of R0_BANK0–R7_BANK0, R0_BANK1–R7_BANK1, and R8–R15 are undefined.

2.2 System registers

Table 3. System registers

| Name | Size | Reset value ⁽¹⁾ | Synopsis |
|-------|-----------|-----------------------------------|---|
| MACH | 32 | Undefined | Multiply-and-accumulate register high. |
| | Operation | | MACH is used for the added value in a MAC instruction, and to store a MAC instruction or MUL instruction operation result. |
| MACL | 32 | Undefined | Multiply-and-accumulate register low. |
| | Operation | | MACL is used for the added value in a MAC instruction, and to store a MAC instruction or MUL instruction operation result. |
| PR | 32 | Undefined | Procedure register. |
| | Operation | | The return address is stored when a subroutine call using a BSR, BSRF or JSR instruction. PR is referenced by the subroutine return instruction (RTS). |
| PC | 32 | See Section 2.8.2 | Program counter. |
| | Operation | | PC indicates the executing instruction address. |
| FPSCR | 32 | 0x0004 0001 | Floating-point status/control register. |
| | Operation | | Refer to Table 4: FPSCR register description . This register is only present in products with an FPU. |
| FPUL | 32 | Undefined | Floating-point communication register. |
| | Operation | | Data transfer between FPU registers and CPU registers is carried out via the FPUL register. The FPUL register is a system register, and is accessed from the CPU side by means of LDS and STS instructions. For example, to convert the integer stored in general register R1 to a single-precision floating-point number, the processing flow is as follows: R1 →(LDS instruction) →FPUL →(single-precision FLOAT instruction) →FR1 This register is only present in products with an FPU. |

1. Initial value after power-on or manual reset.

Table 4. FPSCR register description⁽¹⁾

| FPSCR | | | | |
|------------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| RM | 0 | 1 | Rounding mode. | RW |
| | Operation | | RM = 0: round to nearest, RM = 1: round to zero. For details see Section 6.2: Rounding on page 166 . | |
| | Reset | | 1 | |
| Flag inexact | 2 | 1 | FPU inexact exception flag. | RW |
| | Operation | | Set to 1 if Inexact exception occurs. | |
| | Reset | | 0 | |
| Flag underflow | 3 | 1 | FPU underflow exception flag. | RW |
| | Operation | | Set to 1 if Underflow exception occurs. | |
| | Reset | | 0 | |
| Flag overflow | 4 | 1 | FPU overflow exception flag. | RW |
| | Operation | | Set to 1 if overflow exception occurs. | |
| | Reset | | 0 | |
| Flag division by zero | 5 | 1 | FPU division by zero exception flag. | RW |
| | Operation | | Set to 1 if division by zero exception occurs. | |
| | Reset | | 0 | |
| Flag invalid operation | 6 | 1 | FPU invalid operation exception flag. | RW |
| | Operation | | Set to 1 if Invalid operation exception occurs. | |
| | Reset | | 0 | |
| Enable inexact | 7 | 1 | FPU invalid exception enable field. | RW |
| | Operation | | Set to 1 to cause a trap when an inexact exception occurs. | |
| | Reset | | 0 | |
| Enable underflow | 8 | 1 | FPU underflow exception enable field. | RW |
| | Operation | | Set to 1 to cause a trap when an underflow exception occurs. | |
| | Reset | | 0 | |
| Enable overflow | 9 | 1 | FPU overflow exception enable field. | RW |
| | Operation | | Set to 1 to cause a trap when an overflow exception occurs. | |
| | Reset | | 0 | |

Table 4. FPSCR register description⁽¹⁾ (continued)

| FPSCR | | | | |
|-------------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Enable division by zero | 10 | 1 | FPU division by zero exception enable field. | RW |
| | Operation | | Set to 1 to cause a trap when a division by zero exception occurs. | |
| | Reset | | 0 | |
| Enable invalid | 11 | 1 | FPU invalid exception enable field. | RW |
| | Operation | | Set to 1 to cause a trap when an Invalid exception occurs. | |
| | Reset | | 0 | |
| Cause inexact | 12 | 1 | FPU inexact exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if an Inexact exception occurs. | |
| | Reset | | 0 | |
| Cause underflow | 13 | 1 | FPU underflow exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if an underflow exception occurs. | |
| | Reset | | 0 | |
| Cause overflow | 14 | 1 | FPU overflow exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if an overflow exception occurs. | |
| | Reset | | 0 | |
| Cause division by zero | 15 | 1 | FPU division by zero exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if a division by zero exception occurs. | |
| | Reset | | 0 | |
| Cause invalid | 16 | 1 | FPU invalid exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if an invalid exception occurs. | |
| | Reset | | 0 | |
| Cause FPU error | 17 | 1 | FPU error exception cause field. | RW |
| | Operation | | Set to 0 before an FPU instruction is executed. Set to 1 if an FPU error exception occurs. | |
| | Reset | | 0 | |

Table 4. FPSCR register description⁽¹⁾ (continued)

| FPSCR | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| DN | 18 | 1 | Denormalization mode. | RW |
| | Operation | | DN = 0: A denormalized number is treated as such. DN = 1: A denormalized number is treated as zero. See Section 6.1.2: Denormalized numbers on page 165 | |
| | Reset | | 1 | |
| PR | 19 | 1 | Precision mode. | RW |
| | Operation | | PR = 0: floating point instructions are executed as single precision operations. PR = 1: floating point instructions are executed as double-precision operations (the result of instructions for which double-precision is not supported is undefined). For 100- and 200- series cores, mode setting [SZ = 1, PR = 1] is reserved. FPU operation results are undefined in this mode. For the 300- series core, mode setting [SZ=1, PR=1] is well defined. | |
| | Reset | | 0 | |
| SZ | 20 | 1 | Transfer size mode. | RW |
| | Operation | | SZ = 0: The data size of the FMOV instruction is 32 bits. SZ = 1: The data size of the FMOV instruction is a 32-bit register pair (64 bits). Programming note: 100- and 200- series cores: when SZ = 1 and big endian mode is selected, FMOV can be used for double-precision floating-point data load or store operations. In little endian mode, two 32-bit data size moves must be executed, with SZ = 0, to load or store a double-precision floating-point number. For the 300-series core, mode setting [SZ=1,PR=1] allows double precision load and store operations to be performed. | |
| | Reset | | 0 | |
| FR | 21 | 1 | Floating-point register bank. | RW |
| | Operation | | FR = 0: FPR0_BANK0-FPR15_BANK0 are assigned to FR0-FR15; FPR0_BANK1-FPR15_BANK1 are assigned to XF0-XF15. FR = 1: FPR0_BANK0-FPR15_BANK1 are assigned to FR0-FR15. | |
| | Reset | | 0 | |

Table 4. FPSCR register description⁽¹⁾ (continued)

| FPSCR | | | | |
|-------|------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| RES | [31:22], 1 | 11 | Bits reserved | RW |
| | Operation | | Must be written as 0. Reads return an undefined value. | |
| | Reset | | Undefined | |

1. Only present on products with an FPU.

2.3 Control registers

Table 5. Control registers

| Name | Size | Reset value ⁽¹⁾ | Privilege protection | Synopsis |
|------|-----------|--|---|------------------------|
| SR | 32 | See Table 6 for individual bits. | Yes | Status register |
| | Operation | | Refer to Table 6 . | |
| SSR | 32 | Undefined | Yes | Saved status register |
| | Operation | | The current contents of SR are saved to SSR in the event of an exception or interrupt. | |
| SPC | 32 | Undefined | Yes | Saved program counter |
| | Operation | | The address of an instruction at which an interrupt or exception occurs is saved to SPC. | |
| GBR | 32 | Undefined | No | Global base register |
| | Operation | | GBR is referenced as the base address in a GBR-referencing MOV instruction. | |
| VBR | 32 | 0x0000 0000 | Yes | Vector base register |
| | Operation | | VBR is referenced as the branch destination base address in the event of an exception or interrupt. For details, see Chapter 5: Exceptions on page 146 . | |
| SGR | 32 | Undefined | Yes | Saved general register |
| | Operation | | The contents of R15 are saved to SGR in the event of an exception or interrupt. | |
| DBR | 32 | Undefined | Yes | Debug base register |
| | Operation | | When the user break debug function is enabled (BRCR.UBDE = 1), DBR is referenced as the user break handler branch destination address instead of VBR. | |

1. Initial value at power-on or manual reset.

Table 6. SR register description

| SR | | | | |
|-------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| T | 0 | 1 | True/false condition or carry/borrow bit. | RW |
| | Operation | | Refer to individual instruction descriptions, which affect the T bit. | |
| | Reset | | Undefined | |
| S | 1 | 1 | Specifies a saturation operation for a MAC instruction. | RW |
| | Operation | | Refer to individual instruction descriptions, which affect the S bit. | |
| | Reset | | Undefined | |
| IMASK | [7:4] | 4 | Interrupt mask level. | RW |
| | Operation | | External interrupts lower than or equal to IMASK are masked. | |
| | Reset | | 0xF | |
| Q | 8 | 1 | State for divide step. | RW |
| | Operation | | Used by the DIV0S, DIV0U and DIV1 instructions. | |
| | Reset | | Undefined | |
| M | 9 | 1 | State for divide step. | RW |
| | Operation | | Used by the DIV0S, DIV0U and DIV1 instructions. | |
| | Reset | | Undefined | |
| FD | 15 | 1 | FPU disable bit. | RW |
| | Operation | | <p>FD = 1: An FPU instruction causes a general FPU disable exception, and if the FPU instruction is in a delay slot, a slot FPU disable exception is generated.</p> <p>For further details see FPUDIS description in Section 6.3: Floating-point exceptions on page 166.</p> <p>An FPU instruction is defined as any instruction with an opcode in which the first four bits are 0xF (apart from 0xFFFFD which is the opcode of the reserved instruction), plus the LDC(.L)/STS(.L) instructions for accessing the FPUL and FPSCR registers.</p> <p>On products without an FPU the bit always reads as 1.</p> | |
| | Reset | | 0 (1 on products without an FPU) | |
| BL | 28 | 1 | Exception/interrupt block bit (set to 1 by a reset, exception, or interrupt). | RW |
| | Operation | | BL = 1: Interrupt requests are masked. If a general exception, other than a user break occurs while BL = 1, the processor switches to the reset state. | |
| | Reset | | 1 | |

Table 6. SR register description (continued)

| SR | | | | |
|-------|-------------------------------------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| RB | 29 | 1 | General register bank specifier in privileged mode (set to 1 by a reset, exception or interrupt). | RW |
| | Operation | | RB = 0: R0_BANK0-R7_BANK0 are accessed as general registers R0-R7. (R0_BANK1-R7_BANK1 can be accessed using LDC/STC R0_BANK-R7_BANK instructions.). RB = 1: R0_BANK1-R7_BANK1 are accessed as general registers R0-R7. (R0_BANK0-R7_BANK0 can be accessed using LDC/STC R0_BANK-R7_BANK instructions.). | |
| | Reset | | 1 | |
| MD | 30 | 1 | Processor mode. | RW |
| | Operation | | MD = 0: user mode (Some instructions cannot be executed, and some resources cannot be accessed). MD = 1: privileged mode. | |
| | Reset | | 1 | |
| RES | [3:2] [14:10] [27:16] [31] | 20 | Bits reserved | RW |
| | Reset | | Undefined | |

2.4 Floating-point registers

Note: These registers are only available if the FPU is present.

[Figure 3](#) shows the floating-point registers, there are thirty-two 32-bit floating-point registers, divided into two banks (FPR0_BANK0–FPR15_BANK0 and FPR0_BANK1–FPR15_BANK1). These 32 registers are referenced as FR0–FR15, DR0-14 in steps of 2, FV0-12 in steps of 4, XF0–XF15, XD0-14 in steps if 2, or XMTRX. The correspondence between FPRn_BANKi and the reference name is determined by the FR bit in FPSCR.

- Floating-point registers: FPRn_BANKi (32 registers) where n=[0..15], i=[0..31].
- Single-precision floating-point registers, FRi where i=[0..15]. When FPSCR.FR is clear, FR0–FR15 are assigned to FPR0_BANK0–FPR15_BANK0, when set, FR0–FR15 are assigned to FPR0_BANK1–FPR15_BANK1.
- Double-precision floating-point registers or single-precision floating-point register pairs, DR0-14 in steps of 2. A DR register comprises two FR registers, for example DR2 is composed from FR2 and FR3.
- Single-precision floating-point four element vector registers, FV0-12 (in steps of 4). An FV register comprises four FR registers, for example, FV8 is composed from FR8, FR9, FR10 and FR11.
- Single-precision floating-point extended registers, XF_i (16 registers), which give access to the opposite bank of registers. When FPSCR.FR is clear, XF0-XF15 are assigned to

FPR0_BANK1-FPR15_BANK1, when FPSCR.FR is set XF0-XF15 are assigned to FPR0_BANK0-FPR15_BANK0.

- Single-precision floating-point extended register pairs, XD0-14 in steps of 2, an XD register comprises two XF registers, for example XB12 is composed from XF12 and XF13.
- Single-precision floating-point extended register matrix, XMTRX, comprising of all 16 XF registers.

Figure 3. Floating-point registers

| FPSCR.FR = 0 | | | | FPSCR.FR = 1 | | | |
|--------------|------|------|-------------|--------------|------|-------|--|
| FV0 | DR0 | FR0 | FPR0_BANK0 | XF0 | XD0 | XMTRX | |
| | | FR1 | FPR1_BANK0 | XF1 | | | |
| | DR2 | FR2 | FPR2_BANK0 | XF2 | XD2 | | |
| | | FR3 | FPR3_BANK0 | XF3 | | | |
| FV4 | DR4 | FR4 | FPR4_BANK0 | XF4 | XD4 | | |
| | | FR5 | FPR5_BANK0 | XF5 | | | |
| | DR6 | FR6 | FPR6_BANK0 | XF6 | XD6 | | |
| | | FR7 | FPR7_BANK0 | XF7 | | | |
| FV8 | DR8 | FR8 | FPR8_BANK0 | XF8 | XD8 | | |
| | | FR9 | FPR9_BANK0 | XF9 | | | |
| | DR10 | FR10 | FPR10_BANK0 | XF10 | XD10 | | |
| | | FR11 | FPR11_BANK0 | XF11 | | | |
| FV12 | DR12 | FR12 | FPR12_BANK0 | XF12 | XD12 | | |
| | | FR13 | FPR13_BANK0 | XF13 | | | |
| | DR14 | FR14 | FPR14_BANK0 | XF14 | XD14 | | |
| | | FR15 | FPR15_BANK0 | XF15 | | | |
| XMTRX | XD0 | XF0 | FPR0_BANK1 | FR0 | DR0 | FV0 | |
| | | XF1 | FPR1_BANK1 | FR1 | | | |
| | XD2 | XF2 | FPR2_BANK1 | FR2 | DR2 | | |
| | | XF3 | FPR3_BANK1 | FR3 | | | |
| | XD4 | XF4 | FPR4_BANK1 | FR4 | DR4 | FV4 | |
| | | XF5 | FPR5_BANK1 | FR5 | | | |
| | XD6 | XF6 | FPR6_BANK1 | FR6 | DR6 | | |
| | | XF7 | FPR7_BANK1 | FR7 | | | |
| | XD8 | XF8 | FPR8_BANK1 | FR8 | DR8 | FV8 | |
| | | XF9 | FPR9_BANK1 | FR9 | | | |
| | XD10 | XF10 | FPR10_BANK1 | FR10 | DR10 | | |
| | | XF11 | FPR11_BANK1 | FR11 | | | |
| | XD12 | XF12 | FPR12_BANK1 | FR12 | DR12 | FV12 | |
| | | XF13 | FPR13_BANK1 | FR13 | | | |
| | XD14 | XF14 | FPR14_BANK1 | FR14 | DR14 | | |
| | | XF15 | FPR15_BANK1 | FR15 | | | |

Programming note

After a reset, the values of all 32 registers are undefined.

2.5 Memory-mapped control registers

[Appendix A: Address list on page 535](#) defines the available memory-mapped control registers for the ST40 core. A particular product may have additional control registers (for additional peripherals, for example); these are specific to that product and product documentation should be consulted.

The address list defines the P4 virtual address at which each register may be accessed. The address range 0xFD00 0000 to 0xFFFF FFFF is called the control and peripherals area.

The control and peripherals area may also be accessed from other parts of the virtual address space using an address translation. This is described in [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#).

Note: Software must only access defined locations in the control and peripherals area. The operation of an access to other locations is undefined. Memory-mapped registers must be accessed using a load/store instruction of an equal size to that of the register. The operation of an access using an invalid data size is undefined.

2.6 Data format in registers

Register operands are always long-words (32 bits). When a memory operand is only a byte (8 bits) or a word (16 bits), it is sign-extended into a long-word when loaded into a register.

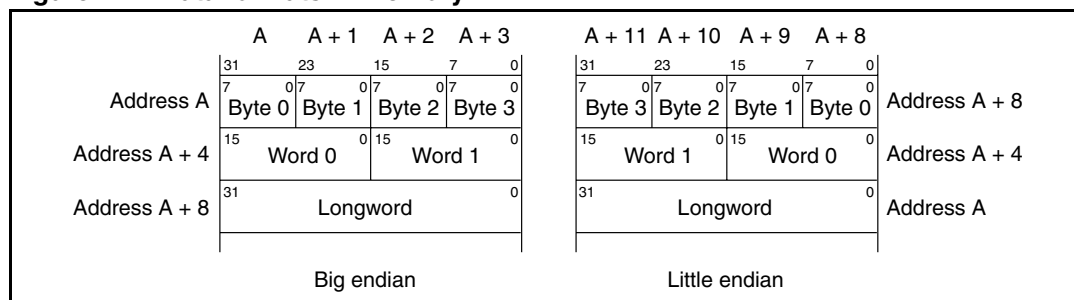
2.7 Data formats in memory

Memory can be accessed in 8-bit byte, 16-bit word, or 32-bit long-word form. A memory operand less than 32 bits in length is sign-extended before being loaded into a register.

A word operand must be accessed starting from a word boundary (even address of a 2-byte unit: address 2n), and a long-word operand starting from a long-word boundary (even address of a 4-byte unit: address 4n). An address error will result if this rule is not observed. A byte operand can be accessed from any address.

Big endian or little endian byte order can be selected for the data format. This endian selection cannot be changed dynamically and is selected by the system during reset. Refer to the datasheet of the product for details of how to perform endian selection. Bit positions are numbered left to right from most-significant to least-significant. Thus, in a 32-bit long-word, the left-most bit, bit 31, is the most significant bit and the right-most bit, bit 0, is the least significant bit.

The data format in memory is shown in [Figure 4](#).

Figure 4. Data formats in memory

Note: The ST40 CPU core only supports endian conversion for the 64-bit data format in the ST40-300 series. Therefore, if double-precision floating-point format (64-bit) access is performed in little endian mode on a 100 or 200 series core, the upper and lower 32 bits will be reversed.

2.8 Processor states

The ST40 CPU core has four processor states. Transitions between the states are shown in [Figure 5](#).

2.8.1 Reset state

The CPU can be placed in one of two reset states, either power-on reset or manual reset. External reset events can place the processor in either of these two reset states, refer to the product's datasheet, *Core Support Peripherals Architecture Manual* and *Emulation Support Peripheral Architecture Manual* for details. Software events which cause a reset, such as an exception while the BL bit in the SR is set, always result in a manual reset. For more information on resets, see [Chapter 5: Exceptions on page 146](#).

The purpose of having two reset modes is to allow some flexibility over which system components are reset. Typically:

- power-on reset will cause all system components to be reset.
- manual reset may, for example, avoid resetting DRAM controllers so that memory contents are preserved.

2.8.2 Reset address

For the ST40-100, 200, 400 and 500 series cores, the reset address is fixed at 0xA000 0000. This applies to both manual and power-on reset.

For the ST40-300 core, the default reset address is 0xA000 0000. Product-specific documentation may define an alternative address.

2.8.3 Exception-handling state

This is a transient state during which the CPU's processor state flow is altered by a reset, general exception, or interrupt exception source.

In the case of a reset, the CPU branches to the reset address (see [Section 2.8.2](#)) and starts executing the user-coded exception handling program.

In the case of a general exception or interrupt, the program counter (PC) contents are saved in the saved program counter (SPC), the status register (SR) contents are saved in the saved status register (SSR), and the R15 contents are saved in saved general register (SGR). The CPU branches to the start address of the user-coded exception service routine, found from the sum of the contents of the vector base address and the vector offset.

See [Chapter 5: Exceptions on page 146](#), for more information on resets, general exceptions, and interrupts.

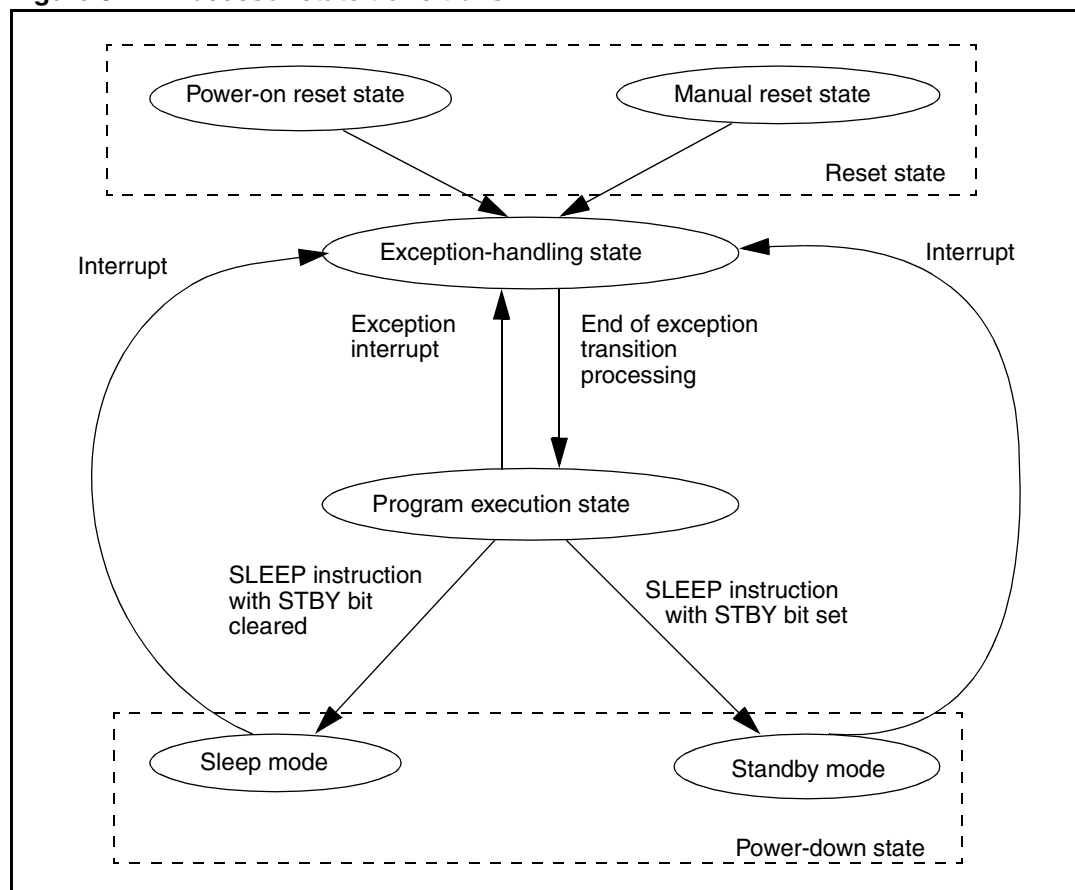
2.8.4 Program execution state

In this state the CPU executes program instructions in sequence.

2.8.5 Power-down state

The power-down state is entered by executing a SLEEP instruction. In this state the CPU stops executing instructions and signals to the system that the CPU has been put to sleep. The system response to receiving this signal is described in the *Core Support Peripherals Architecture Manual* of the appropriate product. The CPU is restarted by raising an interrupt.

Figure 5. Processor state transitions



2.9 Processor modes

There are two processor modes: user mode and privileged mode. The processor mode is determined by the processor mode bit (MD) in the status register (SR). User mode is selected when the MD bit is cleared to 0, and privileged mode when the MD bit is set to 1. When the reset state or exception-handling state is entered, the MD bit is set to 1. When exception handling ends, the MD bit returns to the value held before the exception occurred.

3 Memory management unit (MMU)

The MMU converts virtual addresses emitted by the CPU to physical addresses used by the on-chip bus.

Virtual addresses in the ST40 are 32-bits wide, giving a 4-Gbyte virtual address space. When address translation is enabled in the MMU, 256 address space identifiers (ASIDs) are available. These ASIDs provides the means for several tasks to have translations for the same virtual address set in the MMU at the same time.

Physical addresses in the ST40 may be 29-bits or 32-bits wide, depending on the ST40 variant in use and the operating mode. ST40 cores that support 32-bit physical addressing also support 29-bit addressing, using a 29-bit subset of the 32-bit physical address space. A particular product may alter its physical address map based on whether the ST40 is using 29-bit or 32-bit addressing at the time. Such details are product-specific and will be described in the product's own documentation.

The MMU is an optional feature of the ST40. The ST40-400 series cores do not have an MMU.

3.1 Terminology

This section summarizes the terminology used in describing the ST40 MMU behavior.

| | |
|-------------------------------|---|
| ITLB | Instruction TLB. A micro-TLB that holds a cache of recently accessed instruction mappings. It is automatically refilled from the UTLB and PMB by the ST40 hardware as required. |
| MMU | Memory management unit. The part of the ST40 that converts virtual addresses to physical addresses and manages access to memory resources. |
| P0 | The region of virtual addresses between 0x0000 0000 and 0x7FFF FFFF accessed by the CPU in privileged mode. |
| P1 | The region of virtual addresses between 0x8000 0000 and 0x9FFF FFFF accessed by the CPU in privileged mode. |
| P2 | The region of virtual addresses between 0xA000 0000 and 0xBFFF FFFF accessed by the CPU in privileged mode. |
| P3 | The region of virtual addresses between 0xC000 0000 and 0xDFFF FFFF accessed by the CPU in privileged mode. |
| P4 | The region of virtual addresses between 0xE000 0000 and 0xFFFF FFFF accessed by the CPU in privileged mode. |
| Physical address | Address used by the ST40 at its interface with the STBus |
| PMB | Privileged mapping buffer. A TLB used to map virtual addresses in P1 and P2 when the ST40 is in space-enhancement mode. |
| Space-enhancement mode | The mode of operation where the ST40 uses 32-bit physical addressing at its STBus interface. |

| | |
|------------------------|--|
| TLB | Translation lookaside buffer. A hardware lookup table used for converting virtual addresses to physical addresses. |
| U0 | The region of virtual addresses between 0x0000 0000 and 0x7FFF FFFF accessed by the CPU in user mode. |
| UTLB | Unified TLB. A TLB used to map virtual addresses in U0, P0 and P3. |
| Virtual address | Address used by the ST40 CPU registers when performing instruction and data accesses. |

3.2 ST40 MMU variants

This chapter describes all MMU features that are present across the whole ST40 core family. For a particular type of ST40, only a subset of the chapter applies.

[Table 7](#) shows which features are available on which ST40 core variants.

Table 7. ST40 MMU variants

| Feature | ST40 core family | | | | |
|---|------------------|---------------------------|-----|-----|-----|
| | 101 103 | 202 210 ⁽¹⁾ | 500 | 400 | 300 |
| Presence of memory management unit (MMU) | ✓ | ✓ | ✓ | | ✓ |
| UTLB and ITLB (including memory-mapped arrays) | ✓ | ✓ | ✓ | | ✓ |
| PMB (including memory-mapped array) | | ✓ | ✓ | | ✓ |
| Control registers | | | | | |
| TEA and TTB | ✓ | ✓ | ✓ | ✓ | ✓ |
| PTEH and PTEL | ✓ | ✓ | ✓ | | ✓ |
| MMUCR.{LRUI,URB,URC,SV,TI,AT} | ✓ | ✓ | ✓ | | ✓ |
| MMUCR.SQMD | ✓ | ✓ | ✓ | ✓ | ✓ |
| MMUCR.SE | | ✓ | ✓ | | |
| PASCR | | | | | ✓ |
| IRMCR | | | | | ✓ |
| RAM mode | ✓ | ✓ | | | |
| 29-bit physical addressing | ✓ | ✓ | ✓ | ✓ | ✓ |
| 32-bit physical addressing (space-enhancement mode) (classic scheme) | | ✓ | ✓ | | |
| 32-bit physical addressing (space-enhancement mode) (SH-4A compatible scheme) | | | | | ✓ |

Table 7. ST40 MMU variants (continued)

| Feature | ST40 core family | | | | |
|---|------------------|---------------------------|-----|-----|-----|
| | 101 103 | 202 210 ⁽¹⁾ | 500 | 400 | 300 |
| MMU operating mode availability | | | | | |
| Real mode, 29-bit physical addressing | ✓ | ✓ | ✓ | ✓ | ✓ |
| Translated mode, 29-bit physical addressing | ✓ | ✓ | ✓ | | ✓ |
| Real mode, 32-bit physical addressing | | ✓ | ✓ | | ✓ |
| Translated mode, 32-bit physical addressing | | ✓ | ✓ | | ✓ |
| ICBI and SYNCO instructions | | | | | ✓ |

1. In some ST products containing the ST40-202 core, only the lower 29 physical address bits are connected to the STBus. The 32-bit physical addressing mode is ineffective on such devices. Refer to product-specific documentation as to whether 32-bit physical addressing is available on a particular device.

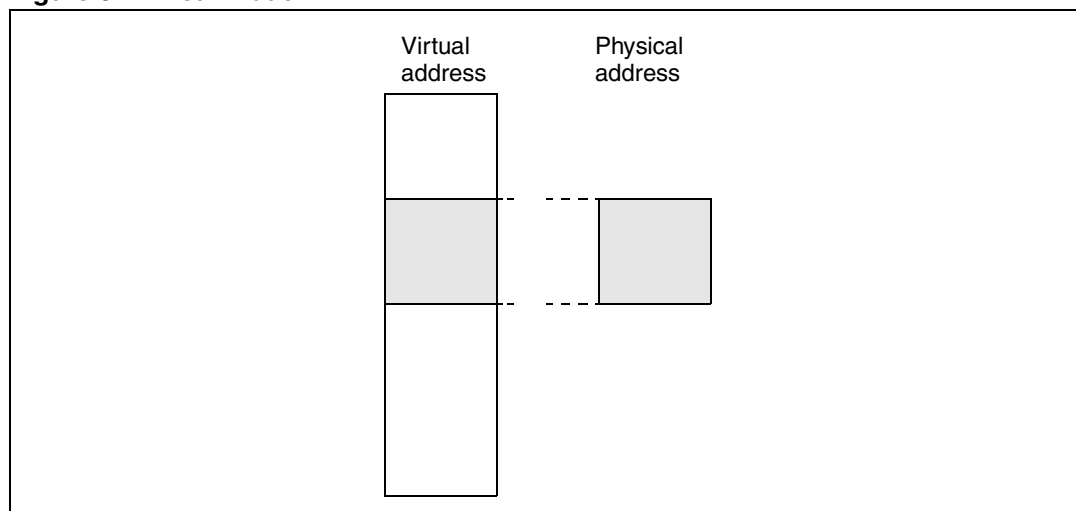
Software may detect whether an MMU is present by attempting to set the MMUCR.AT bit to 1. If the bit remains at zero, there is no MMU: the software is running on a 400-series core. In this case, the physical address space is 29-bits wide and virtual addresses are converted to physical addresses by masking out bits [31:29]. The behavior is identical to that in real mode (MMUCR.AT=0) on cores that have an MMU.

3.3 Role of the MMU

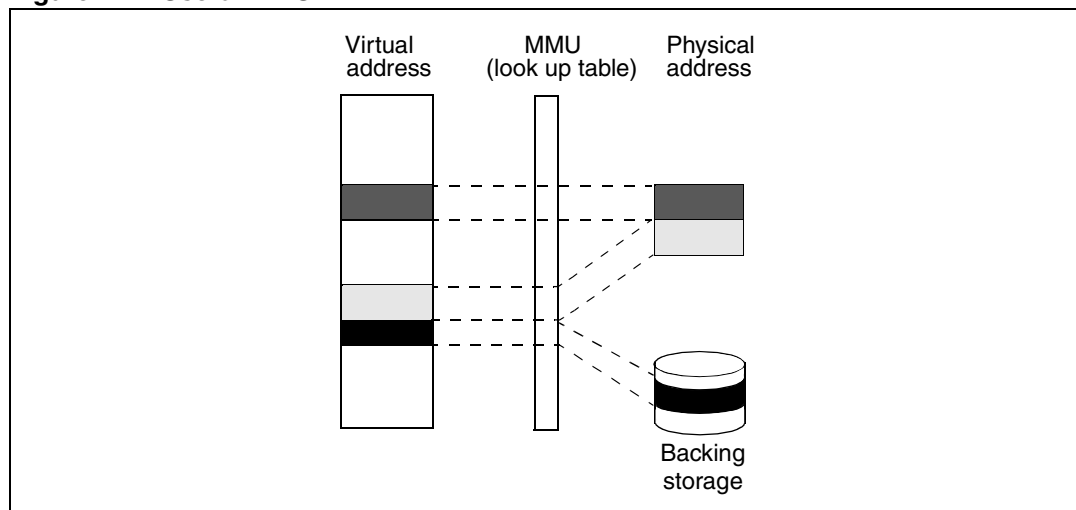
A memory management unit (MMU) has two primary functions.

- It allows the address space used by application software (the virtual addresses) to be independent of the physical addresses at which memory is available.
In particular, it allows an operating system to manage limited physical memory resources efficiently without imposing constraints on the addressing used within the application software.
- It provides isolation between the memory allocated to different applications, and between application memory and an operating system's private memory.

With no MMU, the CPU operates in real mode. This mode uses a fixed translation scheme where the virtual address is converted to the physical address by masking out the upper 3 bits. This is illustrated in [Figure 6](#).

Figure 6. Real mode

With an MMU, the virtual addresses are mapped to the physical addresses by a lookup table. This allows the virtual addressing scheme to be made more convenient for the application to use, regardless of the layout of the physical memory. It also allows an application to use more virtual address space than there is memory available; the operating system exchanges blocks of data between physical memory and backing storage as required. This is illustrated in [Figure 7](#).

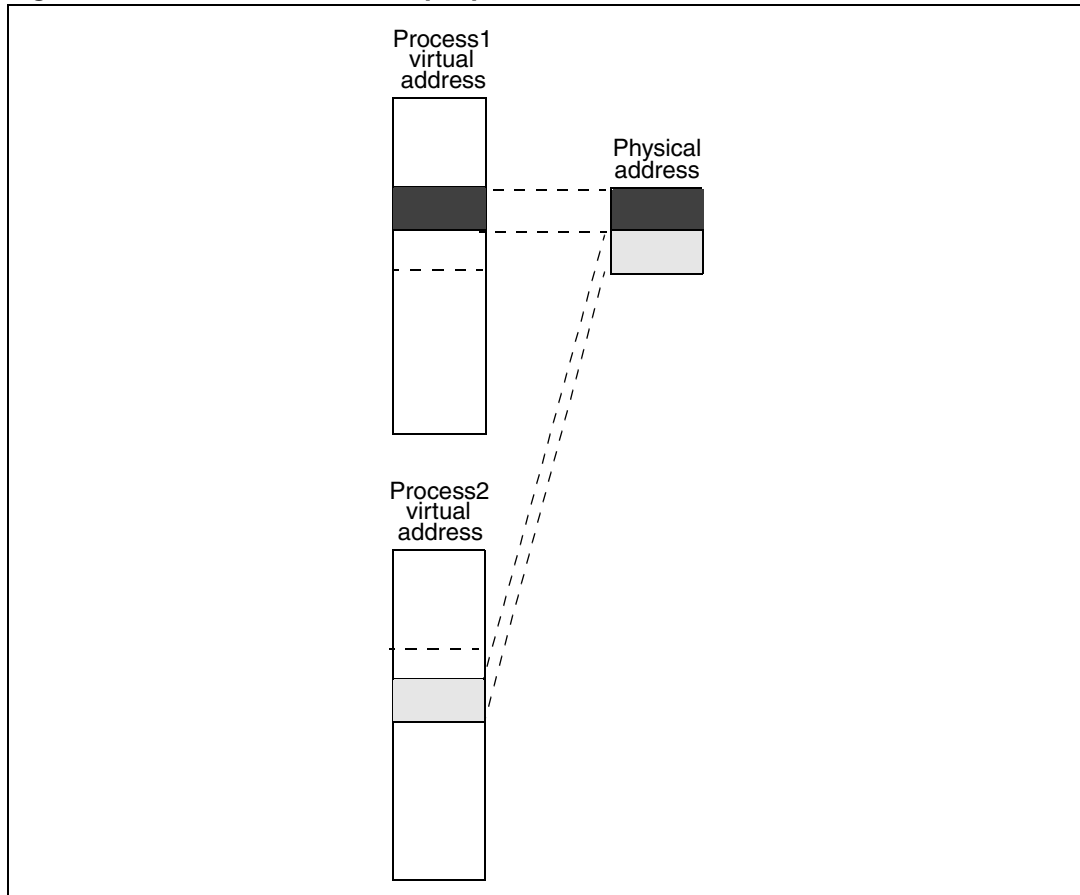
Figure 7. Use of MMU

Although the functions of the MMU could be performed by software alone, this would be extremely inefficient. For this reason, a lookup table for address translation (the translation lookaside buffer (TLB)) is provided by the hardware. The translation information for frequently accessed blocks of virtual addresses is stored there. Software only has to intervene if a virtual address is accessed for which there is currently no translation available in the TLB. This partitioning of responsibilities allows for flexibility in how the software manages the contents of the TLB.

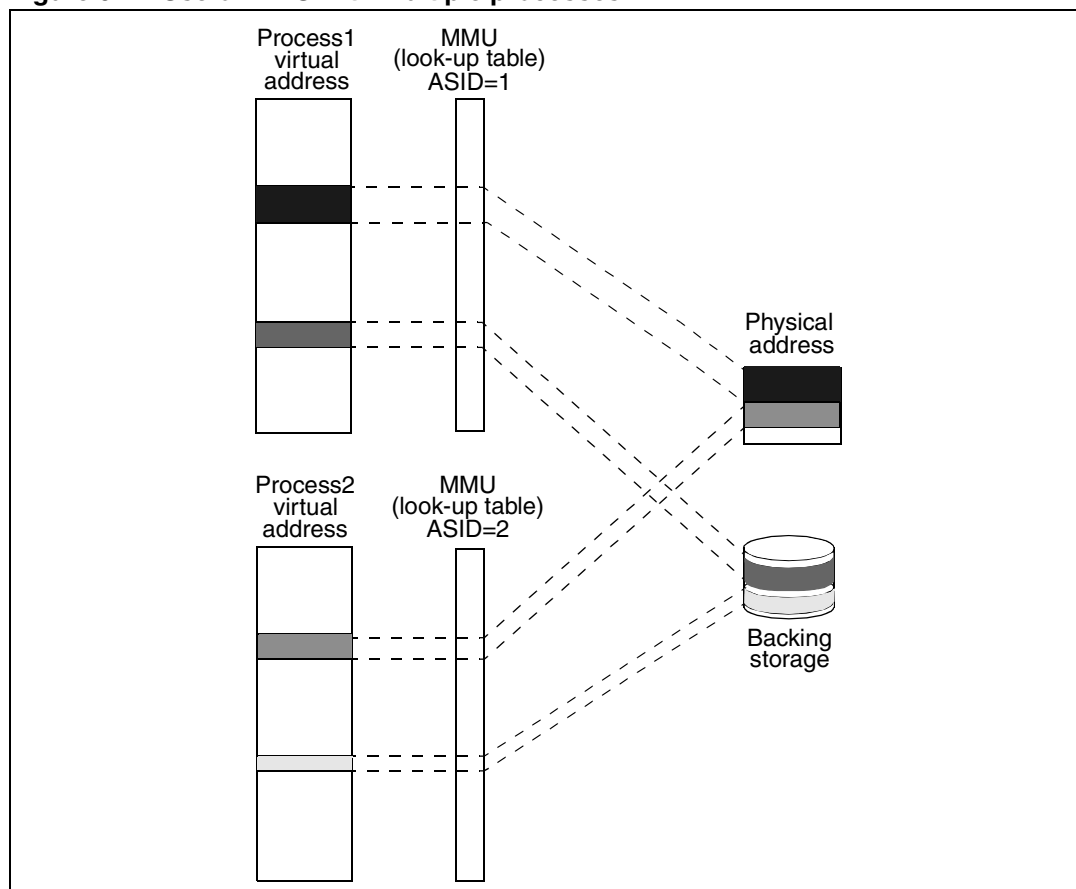
The MMU is particularly useful when there are multiple applications which are time-sharing the ST40. The MMU allows the virtual address space of each application to be independent

of the other applications. Without the MMU, the applications' address spaces would have to be fully co-operative to allow them to share the physical memory, as illustrated in [Figure 8](#).

Figure 8. Real mode with multiple processes



The MMU also provides isolation between the applications; it is impossible for them to access each other's memory maliciously or by accident. Each application has a unique address space identifier (ASID); each translation in the TLB is associated with a particular ASID. This allows the TLB to contain translations for more than one application at the same time, avoiding the need to clear the TLB on each context switch and thereby improving efficiency. This is illustrated in [Figure 9](#).

Figure 9. Use of MMU with multiple processes

3.4 Physical address space

This section describes the ST40's view of the addressing used on STBus.

3.4.1 29-bit physical address space

ST40-100 and ST40-400 series cores only support a 29-bit physical address space. ST40-200, ST40-300 and ST40-500 operate by default with a 29-bit physical address space.

Although the ST40 drives all 32 address lines at its STBus interface, only a 512 Mbyte subset is accessible in 29-bit mode. The resources available in the 29-bit mode are product-specific and product documentation should be consulted. However, all products should normally have been designed so that the ST40 core support peripherals are visible at their documented 29-bit physical addresses.

Software must only access physical addresses which are known to have a resource connected to them. Accesses to unconnected physical addresses will result in undefined behavior. Consult the product documentation for a description of which physical addresses are available.

All current ST40 variants enter 29-bit mode at power-on reset.

3.4.2 32-bit physical address space (space-enhancement mode)

The mode that allows 32-bit physical addressing is termed space-enhancement mode.

Note: Space-enhancement mode is only available on some ST40 core variants. Refer to [Table 7: ST40 MMU variants on page 33](#) for details.

Overview of space-enhancement mode

Where a core supports space-enhancement mode, it may be enabled by setting the SE bit to 1. For the ST40-200 and ST40-500 series cores, the SE bit is in the MMUCR (see [Section 3.8.5: MMU control register \(MMUCR\) on page 63](#)). For the ST40-300 series cores, the SE bit is in the PASCER (see [Section 3.8.6: Physical address space control register \(PASCER\) on page 66](#)).

In space-enhancement mode, a full 32-bit physical address space is available to software. There is normally a transparent relationship between the ST40 physical address and the address of a resource within the device containing the ST40, except in the following cases:

- when the device contains some address mapping outside the ST40 (this would be described in the device datasheet)
- when the ST40 core support peripherals are being accessed (the peripherals may lie at a device-specific address on the bus; the conversion between the ST40 defined addresses and the device address space is managed automatically by hardware)

Feature set

- When SE mode is enabled, the ST40 uses 32-bit physical addressing instead of 29-bit physical addressing.
All current ST40 variants have SE mode turned off at power-on reset, for backwards-compatibility.
- Extension of the PPN field in the UTLB and ITLB, and the tags in the instruction and operand caches, to allow them to represent 32-bit physical addresses.
When SE mode is off, bits [31:29] of these resources operate in a manner that is backwards-compatible with pure 29-bit mode operation.
- A new translation lookaside buffer (the privileged mapping buffer (PMB)) to translate virtual addresses in the P1 and P2 regions when SE mode is enabled. The PMB is fully associative and has 16 entries.
Operating systems have traditionally relied on having a view of the entire 29-bit physical address space through the P1 and P2 regions. By suitable management of the translations in the PMB, operating systems can continue to have access to all physical memory through those regions.

This approach keeps the U0/P0 and P3 region behaviors unchanged as seen by application software, thereby localizing the software changes to support 32-bit physical addressing to certain parts of the operating system.

The space-enhancement mode architecture has been extended for the ST40-300 series. [Table 8](#) shows the differences between the ST40-200 and ST40-500-series architecture and the ST40-300 series architecture.

Table 8. Space-enhancement mode differences

| Feature | ST40-200 and ST40-500 series implementation | ST40-300 series implementation |
|---|--|---|
| Location of SE bit to enable the space-enhancement mode | Bit 4 of MMUCR (Section 3.8.5 on page 63) | Bit 31 of PASC.R (Section 3.8.6 on page 66) |
| Return from SE=1 (32-bit) mode to SE=0 (29-bit) mode | Power-on reset | Power-on reset Manual reset Clearing PASC.R.SE |
| Explicit control of re-ordering and combining for writes to the STBus | None. (The ST40 bus interface and the STBus are required to preserve all critical write-order properties.) | UB bit in each UTLB & PMB entry for per-page control. PASC.R.UB[7:0] for real mode. See Write buffering, re-ordering and combining: UB bits on page 103 . |

3.4.3 Overview of mode switching

When the SE bit is changed, the resulting change of physical address model affects:

- the fetching of subsequent instructions (due to instruction pre-fetching, some subsequent instructions may already have been fetched before the mode switch took effect)
- the handling of any load or store operations that are in progress at the time of the mode switch (load and store operations may be processed in parallel with nearby instructions, including a mode switching instruction)
- the interpretation of the physical addresses information in the UTLB, ITLB and the tags in the instruction and operand caches.

The potential hazards are minimized if the following constraints are adhered to by software when switching mode:

- the PC is in uncached instruction space, in the P2 or P4 region
- the instruction cache is disabled and contains no valid entries
- the operand cache is disabled and contains no valid entries
- the UTLB contains no valid entries
- the ITLB contains no valid entries relating to the U0/P0 or P3 regions
- MMUCR.AT=0
- no data load or store operations to the STBus are in progress

With great care, software may relax these constraints when switching modes. The following sections explain the hazards that must be avoided if these constraints are to be relaxed.

3.4.4 Switching into space-enhancement mode

A switch from 29-bit to space-enhancement (32-bit) mode is achieved by changing the SE bit from 0 to 1. For the ST40-200 and ST40-500 series, the SE bit is bit [4] of MMUCR. For the ST40-300 series, it is bit [31] of PASC.R.

After space-enhancement mode is enabled, the P2 region immediately becomes subject to translation by the PMB^(a). Therefore, when the PC is in the P2 region, before setting SE to 1, software must configure at least one PMB entry so that the virtual addresses of the instructions following the mode switch are mapped. If this is not done, a PMB miss event will occur. A PMB miss results in a reset of the ST40 (see [Section 3.12.4: OTLBMULTIHIT on page 96](#)). If the PC is in P2 during mode switching, behavior is only guaranteed if the PMB page used to map it after the mode switch is uncached.

3.4.5 Switching out of space-enhancement mode

To leave space enhancement mode (other than by reset), software must clear the PASC.R.SE bit to 0.

Note: For the ST40-200 and ST40-500 series, the only guaranteed means to leave space-enhancement mode is through a power-on reset.

3.4.6 Limitations on some ST40-200 series parts

Although the ST40-202 core supports space-enhancement mode internally, several ST products containing a ST40-202 do not have all the core's address lines connected to the STBus. In such products, the ST40 can only support 29-bit physical addressing. However, other features of the space-enhancement mode, for example the use of the PMB to translate virtual addresses in the P1 and P2 regions, are still available.

3.4.7 Effect of mode switching on the UTLB and ITLB

If the UTLB or ITLB contains any valid entry for the U0/P0 or P3 virtual address regions whose physical address (hence PPN) is outside the range 0x0000 0000 through 0x1BFF FFFF when the mode is changed, the effects of such an entry on subsequent translations in the new mode is undefined.

If the original mode is restored, the effect of such entries becomes well-defined again.

Safe operation is guaranteed if the UTLB and ITLB are invalidated (by writing 1 to MMUCR.TI) prior to a mode switch.

3.4.8 Effect of mode switching on the instruction cache

If the instruction cache contains physical addresses outside the range 0x0000 0000 through 0x1BFF FFFF when the mode is changed, the effect of such entries on subsequent cacheable operations is undefined.

Note: Valid instruction cache entries may exist even for instructions that have not been executed because of instruction pre-fetching.

If the mode where such entries were created is restored, the behavior of such entries becomes well-defined again. In particular, if a manual reset occurs in space-enhancement

a. The P1 region is also translated by the PMB in space-enhancement mode.

mode, this feature can be used to recover the state of the cache at the time of the reset, to assist in post-mortem debugging.

If there is any doubt about the addresses present in the instruction cache, safe behavior can be guaranteed if the instruction cache is invalidated prior to the mode switch, by writing 1 to CCR.ICI.

3.4.9 Effect of mode switching on the operand cache

If the operand cache contains physical addresses outside the range 0x0000 0000 through 0x1BFF FFFF when the mode is changed, the effect of such entries on subsequent cacheable operation is undefined.

If the mode where such entries were created is restored, the behavior of such entries becomes well-defined again. In particular, if a manual reset occurs in space-enhancement mode, this feature can be used to recover the state of the cache at the time of the reset, to assist in post-mortem debugging.

If there is any doubt about the addresses present in the operand cache, safe behavior can be guaranteed if the operand cache is invalidated or purged (as appropriate) prior to the mode switch, which can be achieved as follows:

- If the operand cache contains no live data, it may be invalidated by writing 1 to CCR.OCI.
- If the operand cache contains live data, a software loop must be used to purge all such entries to memory.

3.4.10 Effect of mode switching on bus operations

Due to the pipelined nature of the ST40, data load/store instructions may be processed in parallel with a mode switching request.

In this section, load/store instructions are taken to include:

- load instructions
- store instructions
- PREF instructions
- burst writes from the store queues to memory
- burst writes from the operand cache arising from operand cache evictions

To ensure that a sequentially *earlier* load/store is completed prior to the mode switch, the software must execute:

- an uncached load to any address outside of the ST40 itself (ST40-200 or ST40-500 series), or
- a SYNCO instruction (ST40-300 series)

between the earlier load/store and the mode switching instruction.

To ensure that a sequentially *later* load/store is not started until the mode switch is complete, software must contain a synchronisation point (see [Section 3.15: MMU state coherency on page 105](#)) between the mode switch and the later load/store.

If a load/store may be being processed when a mode switch occurs, and the physical address of the load/store is unchanged by the mode switch, the operation of the load/store is well-defined.

If a uncached load/store may be being processed when the mode switch occurs, and the physical address of the load/store is changed by the mode switch, it is undefined whether the old physical address or the new physical address is used for the operation.

If a cached load/store is being processed when the mode switch occurs, and the physical address of the load/store is changed by the mode switch, the resulting behavior is not guaranteed.

Virtual addresses in the range 0x0000 0000 through 0x1BFF FFFF, with MMUCR.AT=0, are guaranteed to retain their physical addresses across a mode switch. Other virtual addresses may or may not retain their physical addresses across a mode switch.

3.4.11 Mode switching when the physical address of the PC changes

For performance reasons, the ST40 pre-fetches instructions; see [Section 3.16: Side effects of instruction pre-fetch on page 107](#). There is no interlocking in the hardware between instruction pre-fetching and the transition between addressing modes. Therefore, the number of instructions following the mode switch that may have been pre-fetched in the previous mode is indeterminate.

Consequently, if the physical address of the PC changes due to the mode switch, all instructions from the mode switch up to and including the next coherency point ([Section 3.15.2: Achieving coherency \(ST40-100, ST40-200, ST40-400 and ST40-500 series cores\) on page 106](#) and [Section 3.15.3: Achieving coherency \(ST40-300 series cores\) on page 106](#)) must be duplicated at the old and new physical addresses to give guaranteed behavior.

Also, if the PC is in a cacheable region with the instruction cache enabled at the time of the mode switch, the physical address of the PC must not change due to the mode switch. Otherwise, subsequent behavior is not guaranteed. In other words, if the physical address of the PC changes due to the mode switch, the PC must be in an uncached region when the mode switch occurs.

If the **MOV.L** instruction that causes the mode switch and the instruction that causes the coherency point are fetched from memory by the same STBus request, the coherency point instruction is necessarily already fetched when the mode switch occurs, and there is no need to duplicate any instructions. For the ST40-200 and ST40-500 series, uncached instruction fetches occur in pairs, starting at 4-byte aligned addresses. For the ST40-300, uncached instruction fetches occur in quartets, starting at 8-byte aligned addresses. Hence, if the **MOV.L** that causes the mode switch and the instruction that causes the coherency point together form a 4-byte-aligned instruction pair, there is no need to duplicate any instructions, regardless of the variant of ST40 being used.

3.4.12 Examples of switching into space-enhancement mode

Two example code sequences are shown below. Both make the following assumptions:

- the SE bit is 0
- the PC is in P2 or P4
- If the PC is in P2, the PMB has already been set up to contain a mapping for the virtual address page containing the PC. This mapping is uncached. The examples are robust irrespective of whether the physical address of the PC changes during the mode switch.
- both caches are off and contain no valid entries
- MMUCR.AT = 0
- the UTLB contains no valid entries
- the ITLB contains no valid entries for virtual addresses in U0/P0 or P3

The example in [Figure 10](#) can be used to enable space enhancement mode on a ST40-300 series core:

Figure 10. Switching to SE mode on an ST40-300 series core

```

mov.l 0f, r0      ! r0 = address of PASCRCR
mov.l 1f, r1      ! r1 = value to write into PASCRCR
stc    sr, r2     ! r2 = SR
synco                ! ensure completion of all previous stores

! IMPORTANT
! Assemblers typically insert a single NOP instruction here if required
! to achieve the following alignment. If this is not the case, a NOP may
! have to be inserted by hand.

.balign 4          ! align PC to 4 bytes : see below for rationale
mov.l  r1, @r0     ! write SE=1 to PASCRCR
ldc    r2, sr      ! write unchanged value into SR
                        ! (provides a coherency point)

! ...
                        ! execution continues in space-enhancement (32-bit) mode
                        ! at the next sequential virtual address

.balign 4
0: .long 0xff000070 ! address of PASCRCR
1: .long 0x800000ff ! value to write to PASCRCR (SE=1)

```

The purpose of the `.balign 4` directive is to force the subsequent two instructions to be fetched by the same memory access. Therefore, even if the PMB configuration causes the physical address of the PC to change as a result of enabling space enhancement mode, no instructions would need to be duplicated.

The example in [Figure 11](#) can be used to enable space enhancement mode on the ST40-200 and ST40-500 series cores:

Figure 11. Switching to SE mode on the ST40-200 and -500 series cores

```

mov.l    0f, r0      ! r0 = address of MMUCR
                ! (This load is uncached and ensures no stores
                ! requests to the bus are pending.)
mov.l    @r0, r1     ! r1 = value from MMUCR
mov      #(1<<4), r3 ! SE bit of MMUCR is bit[4]
or       r3, r1      ! set SE bit in r1
stc      sr, r2      ! r2 = SR

! IMPORTANT
! Assemblers typically insert a single NOP instruction here if required
! to achieve the following alignment. If this is not the case, a NOP may
! have to be inserted by hand.

.balign 4          ! see above for rationale
mov.l    r1, @r0     ! write SE=1 to MMUCR
ldc      r2, sr      ! write unchanged value into SR
                ! (provides a coherency point)

! ...
                ! execution continues in space-enhancement (32-bit) mode
                ! at the next sequential virtual address

.balign 4
0: .long  0xff000010 ! address of MMUCR

```

3.4.13 Example of switching out of space-enhancement mode

A single example code sequence, for ST40-300, is shown in [Figure 12](#). It makes the following assumptions:

- the SE bit is 1
- the PC is in P2 or P4
- if the PC is in P2, it is currently in an uncached page
- both caches are off and contain no valid entries
- MMUCR.AT = 0
- the UTLB contains no valid entries
- the ITLB contains no valid entries for virtual addresses in U0/P0 or P3

Figure 12. Example of switching out of SE mode

```

mov.l 0f, r0      ! r0 = address of PASC
mov.l 1f, r1      ! r1 = value to write into PASC
stc   sr, r2      ! r2 = SR
synco              ! ensure completion of all previous stores

! IMPORTANT
! Assemblers typically insert a single NOP instruction here if required
! to achieve the following alignment. If this is not the case, a NOP may
! have to be inserted by hand.

.balign 4          ! align PC to 4 bytes : see sec. 3.4.10 above for rationale
mov.l r1, @r0      ! write SE=0 to PASC
ldc   r2, sr       ! write unchanged value into SR
                      ! (provides a coherency point)

! ...              ! execution continues in 29-bit mode
                      ! at the next sequential virtual address

.balign 4
0: .long 0xff000070 ! address of PASC
1: .long 0x000000ff ! value to write to PASC (SE=0)

```

3.5 Virtual addresses

The virtual address is defined as the address that is emitted by the CPU to the MMU and cache subsystem. For instruction fetches, the virtual address is the PC of the instruction. For data operations, the virtual address is the result of evaluating the address operands of the instruction. [Table 9](#) shows examples.

Table 9. Virtual addresses for data operations

| Instruction | Virtual address (VA) |
|------------------------|----------------------|
| MOV.L @Rm, Rn | Rm |
| MOV.W Rm, @(R0, Rn) | R0+Rn |
| MOV.W @(disp, GBR), R0 | GBR + (2*disp) |
| OCBP @Rn | Rn |

3.5.1 MMU operating modes

The MMU has four operating modes. Some ST40 core types support only some of these modes.

1. Real mode, 29-bit physical addressing. This is available on all variants.
2. Translated mode, 29-bit physical addressing.
3. Real mode, 32-bit physical addressing.
4. Translated mode, 32-bit physical addressing.

[Table 7: ST40 MMU variants on page 33](#) shows which operating modes are available on which ST40 variants.

3.5.2 Virtual address regions

The behavior of an instruction or data access depends on bits [31:29] of the virtual address and on the operating mode of the MMU. Bits [31:29] of the virtual address define the *address region* as shown in [Table 10](#).

Table 10. Virtual address region selection

| Virtual address bit | | | Virtual address region accessed | |
|---------------------|----|----|---------------------------------|------------------------------|
| 31 | 30 | 29 | Privileged mode (SR.MD = 1) | User mode (SR.MD = 0) |
| 0 | 0 | 0 | P0 | U0 |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | P1 | Address error |
| 1 | 0 | 1 | P2 | Address error |
| 1 | 1 | 0 | P3 | Address error |
| 1 | 1 | 1 | P4 | Address error ⁽¹⁾ |

1. Except for addresses from 0xE000 0000 - 0xE3FF FFFF which the user can use to access the store queues.

All virtual addresses with bit [31] set cause an address error in user mode, with the sole exception of the store queue area. The exception raised in the event of such an error will be IADDERR, RADDERR or WADDERR for instruction fetches, data loads or data stores respectively.

3.5.3 Virtual to physical address mapping

[Table 11](#) defines the mapping from virtual to physical address for each address region in each of the operating modes. *Instruction access* means all instruction fetches, and the lookup for the ICBI instruction (ST40-300 only). Data access means all loads and stores (MOV.B, MOV.W, MOV.L, FMOV.S, FMOV, LDC.L, STC.L, LDS.L, STS.L), and OCBWB, OCBP, OCBI, MOVCA.L, PREF, TAS.B, AND.B, OR.B, XOR.B, MAC.L, MAC.W.

The sections after [Table 11](#) explain each of the translation mechanisms identified in the table.

Table 11. Summary of virtual to physical translation in each virtual address region

| Access type | Address region (Table 10) | CCR.ORA | 29-bit physical addressing (SE=0) | | 32-bit physical addressing (Space-enhancement mode) (SE=1) | |
|--------------------|--|---|---|--|--|--|
| | | | Real MMUCR.AT=0 | Translated MMUCR.AT=1 | Real MMUCR.AT=0 | Translated MMUCR.AT=1 |
| Instruction access | U0/P0 0x0000 0000- 0x7FFF FFFF | | Mask to 29-bit <i>Cacheable</i> | Translate to 29-bit in ITLB (refill ITLB from UTLB on miss) | Identity map <i>Cacheable</i> | Translate to 32-bit in ITLB (refill ITLB from UTLB on miss) |
| | P1 0x8000 0000- 0x9FFF FFFF | | Mask to 29-bit <i>Cacheable</i> | Mask to 29-bit <i>Cacheable</i> | Translate to 32-bit in ITLB (refill ITLB from PMB on miss) | |
| | P2 0xA000 0000- 0xBFFF FFFF | | Mask to 29-bit <i>Non-cacheable</i> | Mask to 29-bit <i>Non-cacheable</i> | | |
| | P3 0xC000 0000- 0xDFFF FFFF | | Mask to 29-bit <i>Cacheable</i> | Translate to 29-bit in ITLB (refill ITLB from UTLB on miss) | Identity map <i>Cacheable</i> | Translate to 32-bit in ITLB (refill ITLB from UTLB on miss) |
| | P4 0xE000 0000- 0xFFFF FFFF | | Some devices may contain memory in P4 that allows instruction fetch. Refer to device manuals and datasheets. Such accesses are noncacheable. | | | |
| Data access | U0/P0 0x00000000- 0x7BFFFFFFF | 0 1 | Mask to 29-bit <i>Cacheable</i> | Translate to 29-bit in UTLB | Identity map <i>Cacheable</i> | Translate to 32-bit in UTLB |
| | U0/P0 0x7C000000- 0x7FFFFFFF | | Undefined | | | |
| | | Access operand cache RAM area (RAM mode on page 50) | | | | |
| | P1 0x8000 0000- 0x9FFF FFFF | | Mask to 29-bit <i>Cacheable</i> | Mask to 29-bit <i>Cacheable</i> | Translate to 32-bit in PMB | |
| | P2 0xA000 0000- 0xBFFF FFFF | | Mask to 29-bit <i>Non-cacheable</i> | Mask to 29-bit <i>Non-cacheable</i> | | |
| | P3 0xC000 0000- 0xDFFF FFFF | | Mask to 29-bit <i>Cacheable</i> | Translate to 29-bit in UTLB | Identity map <i>Cacheable</i> | Translate to 32-bit in UTLB |
| | P4 0xE000 0000- 0xFFFF FFFF | | Special behavior (see Section 3.7: P4 address region on page 54) <i>Non-cacheable</i> | | | |

Table 11. Summary of virtual to physical translation in each virtual address region (continued)

| Access type | Address region (<i>Table 10</i>) | CCR.ORA | 29-bit physical addressing (SE=0) | | 32-bit physical addressing (Space-enhancement mode) (SE=1) | |
|--|---------------------------------------|---------|--------------------------------------|----------------------------------|--|----------------------------------|
| | | | Real MMUCR.AT=0 | Translated MMUCR.AT=1 | Real MMUCR.AT=0 | Translated MMUCR.AT=1 |
| Availability of addressing mode for each variant | | | | | | |
| Variant | ST40-101/103 | | ✓ | ✓ | | |
| | ST40-202/210 | | ✓ | ✓ | ✓ | ✓ |
| | ST40-500 | | ✓ | ✓ | ✓ | ✓ |
| | ST40-400 | | ✓ | | | |
| | ST40-300 | | ✓ | ✓ | ✓ | ✓ |
| Illustrated by | | | <i>Figure 13 on page 110</i> | <i>Figure 14 on page 111</i> | <i>Figure 15 on page 112</i> | <i>Figure 16 on page 113</i> |

Cacheability

Where a cell in [Table 11](#) is marked *Cacheable*, it means that such an access may be cached if CCR.ICE=1 (for an instruction access) or CCR.OCE=1 (for a data access). If CCR.ICE=0, instruction accesses are never cached, regardless of the page or address region behavior. Similarly if CCR.OCE=0, data accesses are never cached, regardless of the page or address region behavior.

Where a cell is marked *Non-cacheable*, it means that such an access will always go direct to the STBus (bypassing the cache), regardless of the state of CCR.ICE or CCR.OCE.

For accesses involving a translation in the UTLB, ITLB or PMB, the cacheability is determined by the C-bit field in the page table entry and not solely by the virtual address region.

[Table 12](#) defines the cacheability for the instruction cache in all operating modes. Empty cells in the table are wildcard fields.

In cacheable regions that are not subject to translation, the write-through / copy-back behavior of the operand cache is determined by the CCR.WT and CCR.CB bits. [Table 13](#) defines the cacheability and write-through / copy-back behavior of the operand cache in all operating modes.

Table 12. Cacheability for the instruction cache

| Virtual address region | CCR.ICE | MMUCR.AT | MMUCR.SE / PASC.R.SE | UTLB/PMB entry | Cache mode |
|------------------------|---------|----------|-------------------------|-------------------|------------|
| | | | | C | |
| U0/P0 & P3 | 0 | | | | Uncached |
| | 1 | 1 | | 0 | |
| | | 0 | | | Cached |
| | | 1 | | 1 | |

Table 12. Cacheability for the instruction cache (continued)

| Virtual address region | CCR.ICE | MMUCR.AT | MMUCR.SE / PASCR.SE | UTLB/PMB entry | Cache mode |
|------------------------|---------|----------|------------------------|-------------------|------------|
| | | | | C | |
| P1 | 0 | | | | Uncached |
| | 1 | | 1 | 0 | |
| | | | 0 | | Cached |
| | | | 1 | 1 | |
| P2 | 0 | | | | Uncached |
| | 1 | | 1 | 0 | |
| | | | 0 | | |
| | | | 1 | 1 | Cached |

Table 13. Cacheability and write-through / copy-back behavior for operand cache

| Virtual address region | CCR.OCE | MMUCR.AT | MMUCR.SE / PASCR.SE | CCR.WT | CCR.CB | UTLB/PMB entry | | Cache mode |
|------------------------|---------|----------|------------------------|--------|--------|-------------------|----|---------------|
| | | | | | | C | WT | |
| U0/P0 & P3 | 0 | | | | | | | Uncached |
| | 1 | 1 | | | | 0 | | |
| | | 0 | | 1 | | | | Write-through |
| | | 1 | | | | 1 | 1 | |
| | | 0 | | 0 | | | | Copy-back |
| | | 1 | | | | 1 | 0 | |
| P1 | 0 | | | | | | | Uncached |
| | 1 | 1 | 1 | | | 0 | | |
| | | 0 | | | 0 | | | Write-through |
| | | 1 | | | | 1 | 1 | |
| | | 0 | | | 1 | | | Copy-back |
| | | 1 | | | | 1 | 0 | |
| P2 | 0 | | | | | | | Uncached |
| | 1 | 0 | | | | 0 | | |
| | | 1 | | | | 1 | 1 | Write-through |
| | | | 1 | | | 1 | 0 | Copy-back |
| P4 | | | | | | | | Uncached |

RAM mode

On ST40-100 and ST40-200 series cores, the operand cache RAM mode may be enabled using the CCR.ORA bit ([Section 4.2.1: Cache control register \(CCR\) on page 116](#)). In this case, virtual addresses between 0x7C00 0000 and 0x7FFF FFFF are used to access the operand cache RAM area. Only data accesses can be made to the operand cache RAM. Instruction accesses may never be made to the operand cache RAM and give undefined behavior.

On the ST40-200 series, the RAM mode facility is only available if space enhancement mode is off (SE=0).

When RAM mode is enabled, this overrides any other behavior for data accesses to this range of addresses.

Mask to 29-bit

The physical address is given by taking the virtual address and replacing bits [31:29] by 3 zero bits. This gives a physical address in the range 0x0000 0000 to 0x1FFF FFFF.

Only physical addresses in the range 0x0000 0000 to 0x1BFF FFFF may be safely accessed through a virtual address that is handled by masking to 29 bits.

If masking gives a physical address in the range 0x1C00 0000 to 0x1FFF FFFF, the behavior of the ST40 is undefined; the sole exception is for accesses to the operand cache RAM mode area 0x7C00 0000 to 0x7FFF FFFF when CCR.ORA=1.

The physical address range 0x1C00 0000 to 0x1FFF FFFF must only be accessed either:

- through a P4 virtual address or
- for the range 0x1D00 0000 to 0x1FFF FFFF, by setting MMUCR.AT=1 and using an address translation in the UTLB

When the ST40 is operating with MMUCR.AT=0 and SE=0, user mode has full visibility of external memory. Although user mode has no access to P4 (and hence to the peripheral and control register region) or to certain instructions (for example, `LDTLB`, `SLEEP`), a malicious user mode application can almost certainly subvert the operating system by modifying its instructions in memory.

Translate in UTLB

The virtual address is translated by the UTLB (see [Section 3.9: Unified TLB \(UTLB\) on page 69](#)) to give the physical address. When a 29-bit physical address is required, the UTLB provides a physical address with bits[31:29] all zero.

Physical addresses in the range 0x0000 0000 to 0x1BFF FFFF (29-bit mode) or 0x0000 0000 to 0xFBFF FFFF (space-enhancement (32-bit) mode) are allocated to memory and peripherals on a device-dependent basis. Refer to device-specific documentation for the specific address map. All physical addresses in these ranges can be generated as the result of a translation in the UTLB.

The physical address resulting from a translation in the UTLB may also lie in the range 0x1D00 0000 to 0x1FFF FFFF (29-bit mode) or 0xFD00 0000 to 0xFFFF FFFF (32-bit mode). This allows such physical addresses to be accessed through virtual addresses in the U0/P0 or P3 regions, by setting up suitable translations in the UTLB.

Failure to find a translation in the UTLB during a lookup will result in a RTLBMIS or WTLBMIS exception being raised (see [Section 3.12.5: WTLBMIS/RTLBMIS on page 96](#)).

Identity mapping

The physical address is identical to the virtual address.

Translate in PMB

The virtual address is translated by the PMB (see [Section 3.11: Privileged mapping buffer \(PMB\) on page 88](#)) to give the physical address.

Physical addresses in the range 0x0000 0000 to 0xFBFF FFFF are allocated to memory and peripherals on a device-dependent basis. Refer to device-specific documentation for the specific address map. All physical addresses in these ranges can be targeted by the PMB.

The physical address resulting from a translation in the PMB may also lie in the range 0xFD00 0000 to 0xFFFF FFFF. This allows such physical addresses to be accessed through virtual addresses in the P1 or P2 regions, by setting up suitable translations in the PMB.

Translate in ITLB

The virtual address of the instruction is translated in the instruction translation lookaside buffer (ITLB). The ITLB has four entries. If the lookup does not find a match, the hardware consults the UTLB or PMB to see if a match can be found there.

- The UTLB is consulted after an ITLB miss for an access to the virtual address regions U0/P0 and P3
- If SE = 1, the PMB is consulted after an ITLB miss for an access to the virtual address regions P1 or P2.

If a match is found in the UTLB or PMB, the matched data is copied by the hardware into the ITLB (see [Hardware refill from the UTLB on page 80](#) and [Hardware refill from the PMB on page 80](#)), and the translation succeeds. If no match is found, an ITLBMISS exception is raised (see [Section 3.12.2: ITLBMISS on page 95](#)).

Physical address generation

[Table 14](#) defines how the physical address (PA) is formed from the virtual address (VA). The address mapping depends on the addressing mode, the region in which the virtual address lies, and the size of the page containing the virtual address when the UTLB, ITLB or PMB is involved.

Table 14. Physical address formation

| MMUCR.SE/ PASC.R.SE | MMUCR.AT | Virtual Address Region | SZ ⁽¹⁾ | Page size | Source of physical address bits | | | | | | | | |
|------------------------|----------|------------------------------|-------------------|-----------|---------------------------------|------------|----------|----------|----------|----------|----------|---------|-------|
| | | | | | [31:29] | [28:27] | [26] | [25:24] | [23:20] | [19:16] | [15:12] | [11:10] | [9:0] |
| 0 | 0 | U0/P0, P3 | | | 000 | VA[28:0] | | | | | | | |
| | 1 | | 00 | 1 Kbyte | 000 | PPN[28:10] | | | | | | VA[9:0] | |
| | | | 01 | 4 Kbyte | 000 | PPN[28:12] | | | | | VA[11:0] | | |
| | | | 10 | 64 Kbyte | 000 | PPN[28:16] | | | | | VA[15:0] | | |
| | | | 11 | 1 Mbyte | 000 | PPN[28:20] | | | | VA[19:0] | | | |
| | | P1, P2 | | | 000 | VA[28:0] | | | | | | | |
| 1 | 0 | U0/P0, P3 | | | VA[31:0] | | | | | | | | |
| | 1 | | 00 | 1 Kbyte | PPN[31:10] | | | | | | VA[9:0] | | |
| | | | 01 | 4 Kbyte | PPN[31:12] | | | | | VA[11:0] | | | |
| | | | 10 | 64 Kbyte | PPN[31:16] | | | | | VA[15:0] | | | |
| | | | 11 | 1 Myte | PPN[31:20] | | | | | VA[19:0] | | | |
| | | P1, P2 | 00 | 16 Mbyte | PPN[31:24] | | | | VA[23:0] | | | | |
| | | | 01 | 64 Mbyte | PPN[31:26] | | | VA[25:0] | | | | | |
| | | | 10 | 128 Mbyte | PPN[31:27] | | | VA[26:0] | | | | | |
| | | | 11 | 512 Mbyte | PPN[31:29] | | VA[28:0] | | | | | | |

1. These are the SZ bits from the UTLB, ITLB or PMB entry which matches the virtual address during lookup.

3.5.4 Multiple virtual address spaces

For virtual addresses translated through the UTLB (and ITLB in the case of instruction accesses), the ST40 MMU provides support for multiple independent virtual address spaces to co-exist.

Each translation loaded into the UTLB and ITLB has an 8-bit address space identifier (ASID) associated with it. The MMU has a concept of a current ASID (actually, the ASID field of the PTEH register, see [Section 3.8.1: Page table entry high register \(PTEH\) on page 57](#)). When a translation is performed, the current ASID and page ASID are compared alongside the virtual address and VPN page. By this means, the UTLB may contain translations for more than one task at the same time, without the need to flush the entire UTLB on a context switch.

Individual pages may be shared between all virtual address spaces, regardless of current ASID, using the SH bit in the page table entries (see [Table 27: Fields in each UTLB data array entry on page 75](#)). Also, the MMU has a mode where the ASID is ignored entirely when the CPU is in privileged mode (see MMUCR.SV in [Section 3.8.5: MMU control register \(MMUCR\) on page 63](#)).

The PMB has no support for ASIDs. All PMB mappings are visible regardless of the current PTEH.ASID value.

3.6 Hardware and software functions

This section describes the division of labor in the ST40 MMU between hardware and software functions.

3.6.1 Hardware functions

The ST40 MMU hardware is responsible for the following actions.

- Determination of virtual address region from the instruction fetch address or data access address.
- Lookup in the ITLB, UTLB or PMB as required.
- Physical address determination based on the virtual address, page size and the result of the TLB or PMB lookup.
- Signalling exceptions to the CPU in the event of error conditions arising; setting PTEH.VPN and TEA in such events.
- Automatically refilling the ITLB from the UTLB or PMB in the event of a miss in the ITLB.
- Incrementing the MMUCR.URC counter on each UTLB lookup.
- Maintaining MMUCR.LRUI after each ITLB access.

3.6.2 Software functions

Software (typically an operating system kernel) is responsible for the following actions.

- Setting up the MMU in the bootstrap code.
- Setting up PMB entries in advance of their being needed.
- Handling exceptions.
- Refilling UTLB entries on demand when ITLBMISS, RTLBMISS or WTLBMISS exceptions arise. The refill data may come from page tables or be generated as needed.
- Removal of, and modification to, existing UTLB, ITLB and PMB entries as the mapping characteristics of pages change over time (for example, to handle page permission changes, copy-on-write page sharing and the like).

3.7 P4 address region

The behavior of the P4 virtual address region is unique. This region contains resources such as memory-mapped ST40 control registers and peripheral registers, and the address ranges used for examining and modifying the TLBs.

3.7.1 Resources accessible only through P4

The P4 addresses in the virtual range 0xE000 0000 to 0xFCFF FFFF refer to resources integral to the ST40 core. These resources can **only** be accessed through that P4 virtual address range.

3.7.2 Resources accessible through P4 and through translations

The P4 addresses in the range 0xFD00 0000 to 0xFFFF FFFF refer to ST40 control registers and peripheral registers. These resources also have an associated physical address.

- In 29-bit mode (SE=0), the physical address is the same as bits [28:0] of the P4 virtual address, with bits [31:29] replaced by zeros.
- In 32-bit mode (SE=1), the physical address is the same as the P4 virtual address.

These resources may be accessed through virtual addresses in other address regions, by means of a translation in the UTLB or PMB.

As an example, suppose the P4 registers starting at 0xFF20 0000 are to be mapped using virtual addresses in the P0 address region. In this case, a UTLB entry would be used with the required virtual address page (in U0/P0) as the VPN setting, and the PPN setting as follows:

- 0x1F20 0000 in 29-bit mode
- 0xFF20 0000 in 32-bit mode

P4 virtual address are always uncached.

Translations from other address regions to physical addresses in the range 0xFFC0 0000 to 0xFFFF FFFF must be uncached and are only defined when the C-bit field of such translations is set to 0.

For ST40-100, ST40-200 and ST40-500 series cores, the same uncacheable restriction applies to the range 0xFD00 0000 to 0xFEFF FFFF.

For the ST40-300 series, the 0xFD00 0000 to 0xFEFF FFFF range may be mapped with cacheable translations from other address regions. Cacheable translations are only suitable for certain types of memory target. For example, a block of conventional memory could be safely mapped with a cacheable translation. However, control registers for peripherals would normally only use uncacheable translations.

3.7.3 P4 virtual addressing structure

The structure of the P4 virtual address region is shown in detail in [Table 15](#).

Where a resource may also be accessed through a translation from another address region, the table indicates the physical address that must be used in the UTLB or PMB entry.

Where a resource can only be accessed from a P4 address (that is, a UTLB/PMB translation cannot be used to access it), the physical address columns indicate that no translation is possible.

Table 15. P4 virtual address region

| Virtual address range | Physical address range (29-bit) | Physical address range (32-bit) | Function |
|-------------------------|---|---------------------------------|---|
| 0xE000 0000-0xE3FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used to access the store queues (SQs). When the MMU is disabled (MMUCR.AT=0) or no MMU is present, the SQ access permissions are specified by the MMUCR.SQMD bit. For details, see Section 4.6: Store queues (SQs) on page 140 and Table 79 on page 144 . (Note that UTLB entries may exist with <i>virtual</i> addresses in the 0xE000 0000 - 0xE3FF FFFF range. Such entries control the store queue behavior when MMUCR.AT=1.) |
| 0xE400 0000-0xEFFF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xF000 0000-0xF0FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the instruction cache address array. For details, see Section 4.5.1: IC address array on page 133 . |
| 0xF100 0000-0xF1FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the instruction cache data array. For details, see Section 4.5.2: IC data array on page 135 . |
| 0xF200 0000-0xF2FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the instruction TLB address array. For details, see Section 3.10.3: ITLB address array on page 82 . |
| 0xF300 0000-0xF37F FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to instruction TLB data array. For details, see Section 3.10.4: ITLB data array on page 84 . |
| 0xF380 0000-0xF3FF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xF400 0000-0xF4FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the operand cache address array. For details, see Section 4.5.3: OC address array on page 136 . |
| 0xF500 0000-0xF5FF FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the operand cache data array. For details, see Section 4.5.4: OC data array on page 138 . |
| 0xF600 0000-0xF60F FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the unified TLB address array. For details, see Section 3.9.2: UTLB address array on page 72 . |

Table 15. P4 virtual address region (continued)

| Virtual address range | Physical address range (29-bit) | Physical address range (32-bit) | Function |
|-------------------------|---|---------------------------------|--|
| 0xF610 0000-0xF61F FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the privileged mapping buffer (PMB) address array on cores that support it. This area is reserved on cores that do not have a PMB. For details, see Section 3.11.2: PMB address array on page 89 . |
| 0xF620 0000-0xF6FF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xF700 0000-0xF70F FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to unified TLB data array. For details, see Section 3.9.3: UTLB data array on page 74 . |
| 0xF710 0000-0xF71F FFFF | Cannot be accessed through a translation from U0/P0/P1/P2/P3. | | Used for direct access to the privileged mapping buffer (PMB) data array on cores that support it. This area is reserved on cores that have no PMB. For details, see Section 3.11.3: PMB data array on page 91 . |
| 0xF720 0000-0xF7FF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xF800 0000-0xFCFF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xFD00 0000-0xFEFF FFFF | 0x1D00 0000-0x1EFF FFFF | 0xFD00 0000-0xFEFF FFFF | This area of P4 is passed to the STBus in all 200-, 300-, 400- and 500- series cores. A product may have non-ST40 peripheral registers in this region. Refer to product-specific documentation for valid addresses on the specific product. The behavior of this area is undefined for ST40-100 series cores. |
| 0xFF00 0000-0xFF2F FFFF | 0x1F00 0000-0x1F2F FFFF | 0xFF00 0000-0xFF2F FFFF | Control register area. This address range is used by control registers within the ST40 core itself. Addresses other than those specified in the architecture manuals and/or product datasheets are reserved and accesses to such addresses result in undefined behavior. |
| 0xFF30 0000-0xFFBF FFFF | Reserved | | Reserved (undefined behavior). |
| 0xFFC0 0000-0xFFFF FFFF | 0x1FC0 0000-0x1FFF FFFF | 0xFFC0 0000-0xFFFF FFFF | Core support peripherals area. In most products, it is routed by the STBus to the ST40's core support peripherals. For the ST40-300 series, the peripherals actually occupy a different range of addresses on the bus. The conversion between the physical addresses in this table and the addresses used on the bus is performed automatically by the hardware. The address conversion is described in <i>ST40-300 Core Support Peripherals (8011247), Peripheral Bridge, Address map</i> . |

3.8 Register descriptions

There are seven MMU-related registers. These registers are listed in [Table 16](#).

Table 16. MMU registers

| Name | Abbreviation | R/W | Initial value ⁽¹⁾ | P4 virtual address ⁽²⁾ | Access size |
|--|--------------|-----|------------------------------|-----------------------------------|-------------|
| Page table entry high register | PTEH | R/W | Undefined | 0xFF00 0000 | 32 |
| Page table entry low register | PTEL | R/W | Undefined | 0xFF00 0004 | 32 |
| Translation table base register | TTB | R/W | Undefined | 0xFF00 0008 | 32 |
| Translation table address register | TEA | R/W | See footnote ⁽³⁾ | 0xFF00 000C | 32 |
| MMU control register | MMUCR | R/W | 0x0000 0000 | 0xFF00 0010 | 32 |
| Physical address space control register | PASCR | R/W | 0x0000 0000 or 0x0000 00FF | 0xFF00 0070 | 32 |
| Instruction refetch inhibit control register | IRMCR | R/W | 0x0000 0000 | 0xFF00 0078 | 32 |

1. The initial value is the value after a power-on reset or manual reset.
2. Some of the registers may be accessed through a translation from another address region as described in [Section 3.7](#). Other registers are subject to restrictions (see [Section 3.14](#)) that prevent access through translations.
3. TEA is undefined after a power-on reset, but is preserved during a manual reset.

When no MMU is present:

- the PTEH, PTEL, PASCR and IRMCR registers always read as zero and any writes to them are ignored
- only the SQMD field in the MMUCR remains modifiable. See [Section 3.8.5: MMU control register \(MMUCR\) on page 63](#).

The TTB and TEA exist on all ST40 variants regardless of the presence of a MMU.

Note: Behavior is undefined if an area designated as a reserved area in this manual is accessed.

3.8.1 Page table entry high register (PTEH)

Note: The PTEH register is only present on ST40 cores that have an MMU.

Long-word access to the PTEH register can be performed at the virtual address 0xFF00 0000.

When an MMU exception or address error exception occurs, bits [31:10] of the virtual address at which the exception occurred are set in the VPN field of the PTEH by hardware. The VPN value set by the hardware allows for just the PTEL register to be set-up by software prior to executing an LDTLB instruction to establish a translation for the excepting address. This reduces some of the code needed in a typical refill-on-demand handler.

The address space identifier for the currently executing process is set in the ASID field by software. The ASID field is never updated by hardware.

The VPN and ASID are recorded in the UTLB when the LDTLB instruction is executed.

When no MMU is present the PTEH ignores all writes and all reads return zero.

The fields of the PTEH register are described in [Table 17](#).

Table 17. PTEH register description

| PTEH | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| ASID | [7:0] | 8 | Address space identifier | RW |
| | Operation | | The address space identifier of the current process. Whenever a UTLB lookup occurs, PTEH.ASID is compared with the ASID of the UTLB entries (for non-shared pages) as part of the matching process. See Section 3.5.4: Multiple virtual address spaces on page 52 . | |
| | Reset | | Undefined | |
| VPN | [31:10] | 22 | Virtual page number | RW |
| | Operation | | For 1-Kbyte: upper 22 bits of virtual address. For 4-Kbyte: upper 20 bits of virtual address. For 64-Kbyte: upper 16 bits of virtual address. For 1-Mbyte: upper 12 bits of virtual address. Lower-order bits are ignored. | |
| | Reset | | Undefined | |

3.8.2 Page table entry low register (PTEL)

Note: The PTEL register is only present on ST40 cores that have an MMU.

Long-word access to the PTEL register can be performed at the virtual address 0xFF00 0004, or through a translation as described in [Section 3.7: P4 address region on page 54](#).

The PTEL register is used to hold the physical page number and page management information to be recorded in a UTLB entry when a LDTLB instruction is subsequently executed.

The contents of the PTEL register are never modified by hardware.

When no MMU is present, the PTEL register ignores all writes and all reads return zero.

The fields of the PTEL register are described in [Table 18](#).

Table 18. PTEL register description

| PTEL | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| WT | 0 | 1 | Write-through bit | RW |
| | Operation | | Indicates whether a cacheable translation has write-through or copy-back behavior in the operand cache. 0: copy-back mode. 1: write-through mode. This bit is ignored for non-cacheable translations (those where the C-bit is zero) and when the operand cache is off (CCR.OCE=0). | |
| | Reset | | Undefined | |
| SH | 1 | 1 | Private page or shared page. | RW |
| | Operation | | Indicates whether the page is shared or private. 0: private page 1: shared page When a page is shared, its ASID is disregarded; only the virtual address comparison is used in determining whether the page matches or not. | |
| | Reset | | Undefined | |
| D | 2 | 1 | Dirty bit | RW |
| | Operation | | Indicates whether a store to an address in the page will cause a FIRSTWRITE exception. 0: store causes a FIRSTWRITE exception. 1: store does not cause a FIRSTWRITE exception. Refer to Table 29 on page 78 for details. | |
| | Reset | | Undefined | |

Table 18. PTEL register description (continued)

| PTEL | | | | |
|----------|------------------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| C | 3 | 1 | Cacheability bit | RW |
| | Operation | | <p>Indicates whether a page is cacheable.</p> <p>0: not cacheable.</p> <p>1: cacheable.</p> <p>When a translation is used to map P4 resources with virtual addresses in the range 0xFF00 0000 to 0xFFFF FFFF, this bit must be cleared to 0.</p> <p>If CCR.ICE=0 and/or CCR.OCE=0, instruction and data accesses respectively will be uncached, regardless of the setting of this bit.</p> | |
| | Reset | | Undefined | |
| SZ0 | 4 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 28 on page 77 for details. | |
| | Reset | | Undefined | |
| PR | [6:5] | 2 | Page protection | RW |
| | Operation | | <p>Specify the access rights to the page.</p> <p>Refer to Table 29 on page 78 for details.</p> | |
| | Reset | | Undefined | |
| SZ1 | 7 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 28 on page 77 for details. | |
| | Reset | | Undefined | |
| V | 8 | 1 | Validity bit | RW |
| | Operation | | <p>Indicates whether the entry is valid.</p> <p>0: invalid.</p> <p>1: valid.</p> <p>Cleared to 0 by a power-on reset.</p> <p>Not affected by a manual reset.</p> | |
| | Reset | | Undefined | |
| Reserved | 9 ⁽¹⁾ | 1 | Reserved | RW |
| | Operation | | <p>This is the behavior on all 100-, 200- and 500- series cores.</p> <p>All writes are ignored and all reads return an undefined value.</p> | |
| | Reset | | Undefined | |

Table 18. PTEL register description (continued)

| PTEL | | | | |
|----------|------------------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| UB | 9 | 1 | Unbuffered write control bit | RW |
| | Operation | | This bit is only defined on the ST40-300 core. Specify whether uncacheable and write-through stores must be unbuffered. Refer to Table 45 on page 104 for details. When PASC.R.SE=0, a read will return zero. When PASC.R.SE=1, a read will return the written value. | |
| | Reset | | Undefined | |
| PPN | [28:10] | 19 | Physical page number | RW |
| | Operation | | Indicates the physical page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the PPN will be ignored by the MMU during translation. For example, with a 64kB page, bits [15:10] of the PPN field will be ignored. The synonym problem must be taken into account when setting the PPN (Section 3.13: Interaction of MMU and cache on page 98). | |
| | Reset | | Undefined | |
| Reserved | [31:29] ⁽¹⁾ | 3 | Reserved | RW |
| | Operation | | This is the behavior on 100-series cores. Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | This is the behavior on 200- and 500-series cores. These bits indicate the most significant three bits of the physical page number. When MMUCR.SE=1, writes set the value and reads return the current value. When MMUCR.SE=0, writes are discarded and reads return an undefined value. | |
| | Reset | | Undefined | |

Table 18. PTEL register description (continued)

| PTEL | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | <p>This is the behavior on 300-series cores.</p> <p>These bits indicate the most significant three bits of the physical page number.</p> <p>When PASC.R.SE=1, writes set the value and reads return the current value.</p> <p>When PASC.R.SE=0, writes set the current value and reads return zero. (In this mode, any value is treated as zero when generating physical addresses for bus accesses). However, because writes are active, it allows UTLB entries to be correctly set up prior to switching the ST40 into space-enhancement mode. The LDTLB instruction will copy the current value into the selected UTLB entry.</p> | |
| | Reset | | Undefined | |

1. Double lines group variant meanings of the same bitfield.

3.8.3 Translation table base register (TTB)

Long word access to the TTB register can be performed at the virtual address 0xFF00 0008, or through a translation as described in [Section 3.7: P4 address region on page 54](#).

The contents of the TTB register are not changed by hardware. The TTB register is available for arbitrary use by software.

The expected function is for storing the base address of the page tables of the current process in memory to assist in refill processing.

This register is present on all ST40 variants, whether or not they have an MMU.

The fields of the TTB register are described in [Table 19](#).

Table 19. TTB register description

| TTB | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| TTB | [31:0] | 32 | Translation table base register. | RW |
| | Operation | | TTB is used, for example, to hold the base address of the currently used page table. | |
| | Reset | | Undefined | |

3.8.4 TLB exception address register (TEA)

Long-word access to the TEA register can be performed at the virtual address 0xFF00 000C, or through a translation as described in [Section 3.7: P4 address region on page 54](#).

The contents of the TEA register may be read and written by software. The contents are updated automatically by the hardware whenever an MMU-related exception occurs.

The TEA register is present on all ST40 variants, whether or not they have an MMU.

The contents of this register can be freely used by software even if no MMU is present.

The fields of the TEA register are described in [Table 20](#).

Table 20. TEA register description

| TEA | | | | |
|-------|----------------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| TEA | [31:0] | 32 | TLB exception address register. | RW |
| | Operation | | After an MMU exception or address error exception occurs, the virtual address at which the exception occurred is set in TEA by hardware. | |
| | Power-on reset | | Undefined | |
| | Manual reset | | Held | |

3.8.5 MMU control register (MMUCR)

Longword access to the MMUCR can be performed at virtual address 0xFF00 0010.

On ST40 variants without an MMU, only the MMUCR.SQMD bit is present. All other bits defined below are absent; writes to those bits are ignored and reads return zero.

On ST40-200 and ST40-500 series cores, bit [4] of the MMUCR is the SE bit, used for selecting 29-bit or 32-bit physical addressing mode. In other ST40 core variants, this bit is absent; writes to it are ignored and reads return zero.

The MMUCR contents can be changed by software. The LRUI and URC fields are also changed by the hardware.

There are restrictions on the region in which the PC may lie when the MMUCR is accessed. Refer to [Section 3.15: MMU state coherency on page 105](#).

The fields of the MMUCR are described in [Table 21](#).

Table 21. MMUCR register description

| MMUCR | | | | |
|-------|-----------|------|--|-----------|
| Field | Bits | Size | Synopsis | Type |
| AT | 0 | 1 | Address translation bit | RW/ RO |
| | Operation | | <p>Specifies whether translation of virtual addresses in U0/P0, P3 and the store queues is enabled or disabled.</p> <p>0: Translation disabled</p> <p>1: Translation enabled.</p> <p>On ST40 variants lacking an MMU, this bit always reads as 0 and ignores writes.</p> | |
| | Reset | | 0 | |
| TI | 2 | 1 | TLB invalidate | W |
| | Operation | | <p>Writing 1 to this bit invalidates (clears to 0) all valid UTLB/ITLB bits. This bit always returns 0 when read.</p> | |
| | Reset | | 0 | |
| SE | 4 | 1 | Space-enhancement mode enable | RW/ RO |
| | Operation | | <p>This bit is present only on 200- and 500-series ST40 cores. Specifies whether space-enhancement mode (32-bit physical addressing) is enabled or disabled.</p> <p>0: Space-enhancement mode disabled</p> <p>1: Space-enhancement mode enabled.</p> <p>On 100-, 300- and 400-series cores, this bit reads as zero and writes are ignored.</p> <p>See Section 3.4.3: Overview of mode switching on page 39 for further details of the behavior of this bit</p> | |
| | Reset | | 0 | |
| SV | 8 | 1 | Single virtual mode bit | RW/ RO |
| | Operation | | <p>The SV bit switches between single virtual memory mode and multiple virtual memory mode. In single virtual memory mode, when the ST40 is in privileged mode (SR.MD=1), the page ASIDs are ignored during ITLB/UTLB lookups, having the effect of making all pages shared.</p> <p>0: Multiple virtual memory mode.</p> <p>1: Single virtual memory mode.</p> <p>When no MMU is present this bit always reads as 0 and ignores writes.</p> | |
| | Reset | | 0 | |

Table 21. MMUCR register description (continued)

| MMUCR | | | | |
|-------|-----------|------|---|-----------|
| Field | Bits | Size | Synopsis | Type |
| SQMD | 9 | 1 | Store queue mode bit | RW |
| | Operation | | <p>Specifies the right of access to the store queues when MMUCR.AT=0</p> <p>0: Access possible in both user and privileged modes.</p> <p>1: Access possible in only privileged mode. Attempted access in user mode will raise an exception (see Table 79 on page 144), even if MMUCR.AT=1 and UTLB page protection bits appear to grant user mode access.</p> | |
| | Reset | | 0 | |
| URC | [15:10] | 6 | UTLB replace counter | RW/ RO |
| | Operation | | <p>Random counter for indicating the UTLB entry which will be replaced by the next LDTLB instruction. URC is incremented each time the UTLB is accessed. When URB > 0, URC is reset to 0 when the condition URC = URB occurs. Also note that, if a value is written to URC by software which results in the condition URC > URB, incrementing is first performed in excess of URB until URC = 0x3F. URC is not incremented by an LDTLB instruction.</p> <p>URC provides a pseudo-random replacement scheme for the UTLB.</p> <p>When no MMU is present this field always reads as 0 and ignores writes.</p> | |
| | Reset | | 0 | |
| URB | [23:18] | 6 | UTLB replace boundary. | RW/ RO |
| | Operation | | <p>When URB > 0, it indicates the upper limit of UTLB entries that may be replaced on a pseudo-random basis by the URC field and the LDTLB instruction. For example, if URB=62, entries 62 and 63 will not be subject to pseudo-random replacement. Thus, URB allows there to be 'locked-in' pages in the UTLB.</p> <p>If URB = 0, the upper bound facility is not used.</p> <p>When no MMU is present this field always reads as 0 and ignores writes.</p> | |
| | Reset | | 0 | |

Table 21. MMUCR register description (continued)

| MMUCR | | | | |
|-------|-------------------------------|------|---|-----------|
| Field | Bits | Size | Synopsis | Type |
| LRUI | [31:26] | 6 | ITLB least recently used state | RW/ RO |
| | Operation | | <p>This field is used to direct which ITLB entry is refilled by the hardware from the UTLB or PMB when an ITLB miss occurs. This field is updated by the hardware whenever an ITLB entry is matched by an instruction address lookup. It is read by the hardware whenever an ITLB refill occurs.</p> <p>It is fully described in Section 3.10.2: ITLB LRU state on page 81.</p> <p>If the field is updated by software, it must ensure that a “setting prohibited” value is not written, otherwise the behavior is undefined.</p> <p>After a power-on or manual reset the bits are initialized to 0, thus the first entry to be updated will be entry 3.</p> <p>When no MMU is present this field always reads as 0 and ignores writes.</p> | |
| | Reset | | 0 | |
| RES | 1, 3, [7:5], [17:16], [25:24] | 9 | <p>Bits reserved</p> <p>Writes are ignored and reads return undefined values.</p> | - |
| | Reset | | Undefined | |

3.8.6 Physical address space control register (PASCR)

Long-word access to the PASCR can be performed at the virtual address 0xFF00 0070.

The contents of the PASCR may be read and written by software. The contents are never modified by hardware.

The PASCR register is only present on the ST40-300 core. Access to this register on other variants gives undefined behavior.

There are restrictions on the region in which the PC may lie when the PASCR is accessed. These are described in [Section 3.15: MMU state coherency on page 105](#).

The fields of the PASCR register are described in [Table 22](#).

Table 22. PASCAR description

| PASCAR | | | | |
|----------|----------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| UB | [7:0] | 8 | Unbuffered write control | RW |
| | Operation | | <p>Specifies whether uncached & write-through stores must be unbuffered, for 29-bit physical addressing mode. Each bit controls a 64MB segment of the 512MB physical address space.</p> <p>0: uncached and write-through stores may be buffered</p> <p>1: uncached and write-through stores must be unbuffered</p> <p>See Write buffering, re-ordering and combining: UB bits on page 103.</p> | |
| | Power-on reset | | <p>0x00 or 0xFF</p> <p>The reset value depends on the characteristics of the ST40 bus interface and the STBus in the device with regard to the safety of legacy code execution.</p> <p>Where the write behavior of the bus is safe without explicit write serialization, the reset value may be 0x00.</p> <p>Where all writes must be serialized by the ST40 to ensure safe execution, the reset value will be 0xFF. New software, which will use other new ST40-300 features (such as the SYNCO instruction) to ensure safe execution, can clear these bits to 0x00. If write serialization is only required on particular 64 Mbyte physical address areas, selected UB bits can be left set.</p> | |
| | Manual reset | | Held | |
| SE | 31 | 1 | Space-enhancement mode enabled | RW |
| | Operation | | <p>Specifies whether space-enhancement mode (32-bit physical addressing) is enabled or disabled.</p> <p>0: Space-enhancement mode disabled</p> <p>1: Space-enhancement mode enabled.</p> <p>See Section 3.4.4: Switching into space-enhancement mode on page 40 for further details of the behavior of this bit</p> | |
| | Power-on reset | | 0 | |
| | Manual reset | | 0 | |
| Reserved | [30:8] | 23 | Reserved | - |
| | Operation | | Reads return 0 and writes are ignored. | |
| | Power-on reset | | 0 | |
| | Manual reset | | 0 | |

3.8.7 Instruction refetch inhibit control (IRMCR)

Long-word access to the IRMCR can be performed using a longword access (**MOV.L**) at the virtual address 0xFF00 0078.

The contents of the IRMCR may be read and written by software. The contents are never modified by hardware.

The IRMCR register is only present on the ST40-300 core. Access to this register on other variants gives undefined behavior.

The IRMCR provides a means to allow legacy software to execute correctly on a ST40-300 core. The situations covered by the IRMCR are those where legacy software might have relied on inserting eight **NOP** instructions to give a sufficient guard interval on earlier ST40 cores. Because the ST40-300 series has a different pipeline, eight **NOP** operations will not guarantee that the critical instruction will be fetched after the state change has occurred.

All new software targeted at the ST40-300 must use any one of the other documented methods described in [Section 3.15.3: Achieving coherency \(ST40-300 series cores\) on page 106](#) to ensure a refetch occurs where required.

There are restrictions on the region in which the PC may lie when the IRMCR itself is read or written. These are described further in [Section 3.15: MMU state coherency on page 105](#).

The fields of the IRMCR register are described in [Table 23](#).

Table 23. IRMCR description

| IRMCR | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| MC | 0 | 1 | Refetch control for memory-mapped I-cache update | RW |
| | Operation | | Inhibit refetch of the instruction following a store to the memory mapped instruction cache (that is, a store to addresses starting 0xF0 or 0xF1) whilst CCR.ICE=1. 0: The next instruction is refetched 1: The next instruction is not refetched | |
| | Reset | | 0 | |
| MT | 1 | 1 | Refetch control for memory-mapped TLBs | RW |
| | Operation | | Inhibit refetch of the instruction following a store to the memory-mapped ITLB, UTLB or PMB (that is, a store to addresses starting 0xF2, 0xF3, 0xF6, 0xF7) whilst MMUCR.AT=1. 0: The next instruction is refetched 1: The next instruction is not refetched | |
| | Reset | | 0 | |
| LT | 2 | 1 | Refetch control for LDTLB instruction | RW |
| | Operation | | Inhibit refetch of the instruction following a LDTLB instruction. 0: The next instruction is refetched 1: The next instruction is not refetched | |
| | Reset | | 0 | |

Table 23. IRMCR description (continued)

| IRMCR | | | | |
|----------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| R1 | 3 | 1 | Refetch control following a change to the 0xFF2 register space. | RW |
| | Operation | | Inhibit refetch of the instruction following a store to the P4 region 0xFF200000 through 0xFF2FFFFFF (or the equivalent addresses accessed through a translation) 0: The next instruction is refetched 1: The next instruction is not refetched | |
| | Reset | | 0 | |
| R2 | 4 | 1 | Refetch control following a change to key 0xFF0 registers. | RW |
| | Operation | | Inhibit refetch of the instruction following a store to the following registers: MMUCR, PASCAR, CCR, PTEH or RAMCR (either through their P4 addresses or through an equivalent translation) 0: The next instruction is refetched 1: The next instruction is not refetched | |
| | Reset | | 0 | |
| Reserved | [31:5] | 27 | Reserved | - |
| | Operation | | Reads return 0 and writes are ignored. | |
| | Reset | | 0 | |

3.9 Unified TLB (UTLB)

The unified translation look-aside buffer (UTLB) has 64 entries and is fully associative (that is, the translation for any virtual page may be stored in any of the entries).

Each entry consists of the following fields:

- VPN: virtual page number
- PPN: physical page number
- V: validity
- SZ: page size
- SH: whether page is shared or local to a single address space
- PR and D: page protection, see [Table 29 on page 78](#)
- C: whether page is cacheable
- WT: write-through or copy-back cache behaviour
- UB: write buffering control (ST40-300 series only)

The UTLB is so-named because of its use for the following two purposes:

- to translate a virtual address to a physical address in a data access,
- as a table of address translation information, to be recorded in the instruction TLB (ITLB) in the event of an ITLB miss.

Thus the UTLB is a unified table of translations for both instruction and data accesses, with the ITLB acting as a cache of the most recently used instruction address translations.

In many cases, an operating system will maintain a set of page tables in memory, with the UTLB contents being a recently-accessed subset of the page tables entries. Software is responsible for refilling the UTLB entries from the page tables when a UTLB miss event occurs.

3.9.1 UTLB operations

This section describes the management operations available for the UTLB.

Reading entries

The current contents of an entry may be read through the memory-mapped address and data arrays (see [Section 3.9.2](#) and [Section 3.9.3](#)). The V and D bits are visible through both arrays. The V and D bits in the address array and data array are the same physical bit. The V and D bits may be set through either the address array or the data array; the most recently written bit is stored.

Modifying a specified entry

A specific entry may be modified by writing to the memory-mapped address and data arrays. In this case, the ENTRY field of the address selects the entry to act on. For the address array, the ASSOC bit is set to zero.

The V and D bits may be set through either the address array or the data array. The most recently written value is stored in the UTLB.

Refilling the UTLB using pseudo-random replacement

When a LDTLB instruction is executed by the ST40, the data in the PTEH and PTEL registers (see [Section 3.8.1 on page 57](#) and [Section 3.8.2 on page 59](#)) is read and stored in the UTLB entry specified by the URC field of the MMUCR (see [Section 3.8.5 on page 63](#)). The URC is incremented by each UTLB access. Thus, the LDTLB instruction can be used as a means to implement UTLB refill with a pseudo-random replacement policy.

Note that software can set the URC to a specific value prior to a LDTLB instruction if required, allowing direct programming of a specific entry through data in the PTEH and PTEL registers.

The MMUCR.URB field allows a block of high-numbered UTLB entries to be protected from pseudo-random replacement, if required.

Use of the LDTLB instruction must adhere to the restrictions in [Section 3.14: Memory coherency on page 99](#).

Globally clearing the UTLB

When a 1 is written to MMUCR.TI, the V bit of every UTLB entry is cleared. (This action also clears the V bit of every ITLB entry).

Clearing a UTLB entry for a specific page

If a specific page is to be invalidated, the UTLB supports an associative write operation. This operation checks whether the page is present in the UTLB, and updates the page parameters if a match is found. If the page is not present, no further action is taken. Any matching entry in the ITLB is also modified.

The associative write operation avoids the need for software to examine each UTLB entry to look for a possible match.

Marking a specific page as clean or dirty

If the dirty/clean status of a specific page is to be changed, the same associative write operation may be used to carry out the update.

Changing the clean/dirty state of a page in this way is a common operation for operating systems that use copy-on-write techniques to share the physical memory used by pages that conceptually belong to a single process but which have never yet been written to.

The associative write operation again avoids the need for software to examine each UTLB entry to look for a possible match.

Lookup

The hardware automatically performs a lookup in the UTLB when required.

Entry *N* in the UTLB is said to *match* if and only if

```
[ (UTLB[N].ASID == PTEH.ASID) ||
  ((SR.MD == 1) && (MMUCR.SV == 1)) ||
  (UTLB[N].SH == 1) ] &&
[UTLB[N].V == 1] &&
[MMUCR.AT == 1] &&
[ (VA[31:20] == UTLB[N].VPN[31:20]) &&
  ((UTLB[N].SZ == 3) || (VA[19:16] == UTLB[N].VPN[19:16])) &&
  ((UTLB[N].SZ >= 2) || (VA[15:12] == UTLB[N].VPN[15:12])) &&
  ((UTLB[N].SZ >= 1) || (VA[11:10] == UTLB[N].VPN[11:10])) ]
```

where *VA* is the virtual address being looked-up.

If a single entry matches, the lookup is successful. If no entry matches, an exception is raised (ITLBMISS, RTLBMIS or WTLBMIS depending on the circumstances). If two or more entries match, a ITLBMULTIHIT or OTLBMULTIHIT exception is raised. Either of these multihit exceptions result in a reset of the ST40.

When a lookup is successful, the remaining fields of the UTLB entry define the physical address, access permission, cache behavior and write buffering behavior used for the access.

3.9.2 UTLB address array

The UTLB address array is allocated to virtual addresses 0xF600 0000 to 0xF60F FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (`MOV.L`) must be used when accessing the array. Loads are used to examine the contents of the UTLB. Stores are used to update the contents.

There is no UTLB address array on the ST40-400. Stores to its address range are ignored and reads return zero.

The address used for a load or store to the array defines the entry number to be acted on for addressed operations. Bit 7 of the address indicates whether an addressed operation or an associative write operation is to be performed. Bit 7 has no effect on reads.

The address is formed as shown in [Table 24](#).

Table 24. Addressing for the UTLB address array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|--|
| [31:20] | 12 | 0xF60 | Selects the UTLB address array within the P4 address region |
| [19:14] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [13:8] | 6 | ENTRY | These bits define the entry in the UTLB that is to be acted upon (0..63). This field is ignored by associative writes. |
| 7 | 1 | ASSOC | Specifies whether addressed or associative operation is required 0: addressed (non-associative) operation 1: associative operation Associative operation is only meaningful for writes. This bit is ignored for reads. |
| [6:2] | 5 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 25](#).

Table 25. Fields in each UTLB address array entry

| UTLB address array entry | | | | |
|--------------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| ASID | [7:0] | 8 | Address space identifier | RW |
| | Operation | | When an address is to be translated, the value in PTEH.ASID (Section 3.8.1: Page table entry high register (PTEH) on page 57) is compared with the ASID in each UTLB entry. Unless the page is shared or single virtual mode is active, the ASIDs must match for the UTLB entry to be considered a match | |
| | Reset | | Undefined | |

Table 25. Fields in each UTLB address array entry (continued)

| UTLB address array entry | | | | |
|--------------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| V ⁽¹⁾ | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Reset | | Undefined | |
| D ⁽¹⁾ | 9 | 1 | Dirty | RW |
| | Operation | | Indicates whether the page is considered dirty or clean 0: page is clean 1: page is dirty If a store is made to a page that is marked clean, a FIRSTWRITE exception is raised. This feature is typically used by operating systems to implement copy-on-write sharing of pages. | |
| | Reset | | Undefined | |
| VPN | [31:10] | 22 | Virtual page number | RW |
| | Operation | | Indicates the virtual page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the VPN will be ignored by the MMU during translation. For example, with a 64kB page, bits [15:10] of the VPN field will be ignored. | |
| | Reset | | Undefined | |

1. The V and D bits in the address array and data array are the same physical bit. The V and D bits may be set through either the address array or the data array; the most recently written bit is stored.

The following three kinds of operation can be used on the UTLB address array:

UTLB address array read

VPN, D, V, and ASID are read from the UTLB entry corresponding to the entry set in the ENTRY field of the address. The ASSOC bit of the address is ignored for reads; an addressed read is always performed.

UTLB address array write (non-associative)

The ASSOC bit in the address must be 0 to select this write mode.

VPN, D, V, and ASID specified in the stored value are written to the UTLB entry corresponding to the entry set in the ENTRY field of the address.

UTLB address array write (associative)

The ASSOC bit in the address must be 1 to select this write mode.

Note: The ENTRY field is ignored by associative writes.

Comparison of all UTLB entries is carried out using the VPN specified in the stored value and PTEH.ASID. The usual address comparison rules are followed. The ASID comparison is ignored for pages with SH=1 or if single virtual mode is active.

If a UTLB miss occurs, the result is no operation, and an exception is not generated. If the comparison identifies a matching UTLB entry, corresponding to the VPN specified in the stored value, D and V specified in the stored value are written to that entry. If there is more than one matching entry, a OTLBMULTIHIT exception is raised and the ST40 is reset.

The associative operation is simultaneously carried out on the ITLB, regardless of whether a UTLB match is found. If a matching entry is found in the ITLB, V is written to that entry.

If there is a match in both the UTLB and ITLB, the matching ITLB entry is refilled with the updated UTLB entry contents. Therefore, if the ITLB and UTLB entries were intentionally inconsistent before the operation, the former ITLB entry settings will be lost.

3.9.3 UTLB data array

The UTLB data array is allocated to virtual addresses 0xF700 0000 to 0xF70F FFFF in the P4 region. A translation from another region can not be used to access the array. Longword loads and stores (MOV.L) must be used when accessing the array. Loads are used to examine the contents of the UTLB. Stores are used to update the contents.

There is no UTLB data array on the ST40-400. Stores to its address range are ignored and reads return zero.

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

Note: There is no associative write facility for the data array.

A load returns the current settings for the specified entry.

A store updates all fields for the specified entry from the value to be stored.

The address is formed as shown in [Table 26](#).

Table 26. Addressing for the UTLB data array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|---|
| [31:20] | 12 | 0xF70 | Selects the UTLB data array within the P4 address region |
| [19:14] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [13:8] | 6 | ENTRY | These bits define the entry in the UTLB that is to be acted upon (0..63). |
| [7:2] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 27](#).

Table 27. Fields in each UTLB data array entry

| UTLB data array entry | | | | |
|-----------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| WT | 0 | 1 | Write-through or copy-back | RW |
| | Operation | | Indicates whether a cacheable translation has write-through or copy-back behavior in the operand cache. 0: copy-back behavior 1: write-through behavior This bit is ignored for non-cacheable translations (those where the C-bit is zero) and when the operand cache is off (CCR.OCE=0). | |
| | Reset | | Undefined | |
| SH | 1 | 1 | Private page or shared page | RW |
| | Operation | | Indicates whether the page is shared or private. 0: private page 1: shared page When a page is shared, its ASID is disregarded; only the virtual address comparison is used in determining whether the page matches or not. | |
| | Reset | | Undefined | |
| D ⁽¹⁾ | 2 | 1 | Dirty | RW |
| | Operation | | Indicates whether the page is considered dirty or clean 0: page is clean 1: page is dirty If a store is made to a page that is marked clean, a FIRSTWRITE exception is raised. This feature is typically used by operating systems to implement copy-on-write sharing of pages. | |
| | Reset | | Undefined | |
| C | 3 | 1 | Cacheability | RW |
| | Operation | | Indicates whether the page is cacheable 0: page is not cacheable 1: page is cacheable When a translation is used to map P4 resources with virtual addresses starting 0xFF, this bit must be cleared to 0. If CCR.ICE=0 and/or CCR.OCE=0, instruction and data accesses respectively will be uncached, regardless of the setting of this bit. | |
| | Reset | | Undefined | |
| SZ0 | 4 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 28 on page 77 for details. | |
| | Reset | | Undefined | |

Table 27. Fields in each UTLB data array entry (continued)

| UTLB data array entry | | | | |
|-----------------------|------------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| PR | [6:5] | 2 | Page protection | RW |
| | Operation | | Specify the access rights to the page. Refer to Table 29 on page 78 for details. | |
| | Reset | | Undefined | |
| SZ1 | 7 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 28 on page 77 for details. | |
| | Reset | | Undefined | |
| V ⁽¹⁾ | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Reset | | Undefined | |
| Reserved | 9 ⁽²⁾ | 1 | Reserved | RW |
| | Operation | | <i>This is the behavior on all 100-, 200- and 500- series cores</i> Writes are ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| UB | 9 | 1 | Unbuffered write control bit | RW |
| | Operation | | <i>This bit is only defined on the ST40-300 core.</i> Specify whether uncacheable and write through stores must be unbuffered. Refer to Table 45: Definition of UB bits on page 104 for details. When PASC.R.SE=0, a read will return zero. When PASC.R.SE=1, a read will return the written value. | |
| | Reset | | Undefined | |
| PPN | [28:10] | 22 | Physical page number | RW |
| | Operation | | Indicates the physical page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the PPN will be ignored by the MMU during translation. For example, with a 64kB page, bits [15:10] of the PPN field will be ignored. | |
| | Reset | | Undefined | |
| Reserved | [31:29] | 3 | Reserved | RW |
| | Operation | | <i>This is the behavior on 100-series cores.</i> Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |

Table 27. Fields in each UTLB data array entry (continued)

| UTLB data array entry | | | | |
|-----------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | This is the behavior on 200- and 500-series cores. These bits indicate the most significant 3 bits of the physical page number. When MMUCR.SE=1, writes set the value and reads return the current value. When MMUCR.SE=0, writes are discarded and reads return an undefined value. | |
| | Reset | | Undefined | |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | This is the behavior on 300-series cores. These bits indicate the most significant 3 bits of the physical page number. When PASC.R.SE=1, writes set the value and reads return the current value. When PASC.R.SE=0, writes set the current value and reads return zero. (In this mode, any value is treated as zero when generating physical addresses for bus accesses.) However, because writes are active, it allows UTLB entries to be correctly set up prior to switching the ST40 into space-enhancement mode. | |
| | Reset | | Undefined | |

1. The V and D bits in the address array and data array are the same physical bit. The V and D bits may be set through either the address array or the data array; the most recently written bit is stored.
2. Double lines group variant meanings of the same bitfield.

[Table 28](#) shows the page size behavior for the UTLB.

Note: P1, P2 and P4 do not appear in [Table 28](#) because they are never subject to translation by the UTLB, see [Table 11 on page 47](#).

Table 28. UTLB page size behavior and active VPN, PPN address bits

| SE | VPN [31:30] | Region | SZ1 | SZ0 | Page size | Active VPN bits | Active PPN bits | Restrictions |
|----|----------------------|----------------------|-----|-----|-----------|-----------------|-----------------|--|
| 0 | 0b00 0b01 0b11 | U0/P0 U0/P0 P3 | 0 | 0 | 1kB | [31:10] | [28:10] | |
| | | | 0 | 1 | 4kB | [31:12] | [28:12] | |
| | | | 1 | 0 | 64kB | [31:16] | [28:16] | |
| | | | 1 | 1 | 1MB | [31:20] | [28:20] | |
| 1 | 0b00 0b01 0b11 | U0/P0 U0/P0 P3 | 0 | 0 | 1kB | [31:10] | [31:10] | Only on 200-, 300- and 500-series cores. |
| | | | 0 | 1 | 4kB | [31:12] | [31:12] | |
| | | | 1 | 0 | 64kB | [31:16] | [31:16] | |
| | | | 1 | 1 | 1MB | [31:20] | [31:20] | |

Table 29 shows the types of access allowed to a page based on the values of the page table entry's PR and D bits and the ST40 operating mode. The table lists the exceptions raised (FIRSTWRITE, READPROT, WRITEPROT) if an access is not allowed, a tick indicates that an access is allowed.

Table 29. UTLB page access rights and exception types

| PR | | D | Privileged mode (SR.MD=1) access | | User mode (SR.MD=0) access | |
|--------|--------|--------|----------------------------------|------------|----------------------------|------------|
| Bit[6] | Bit[5] | Bit[2] | Read | Write | Read | Write |
| 0 | 0 | 0 or 1 | ✓ | WRITEPROT | READPROT | WRITEPROT |
| 0 | 1 | 0 | ✓ | FIRSTWRITE | READPROT | WRITEPROT |
| 0 | 1 | 1 | ✓ | ✓ | READPROT | WRITEPROT |
| 1 | 0 | 0 or 1 | ✓ | WRITEPROT | ✓ | WRITEPROT |
| 1 | 1 | 0 | ✓ | FIRSTWRITE | ✓ | FIRSTWRITE |
| 1 | 1 | 1 | ✓ | ✓ | ✓ | ✓ |

The following two kinds of operation can be used on the UTLB data array:

UTLB data array read

PPN, V, SZ0, SZ1, PR, C, D, SH and WT are read from the UTLB entry corresponding to the entry set in the ENTRY field of the address.

For a ST40-200, ST40-300 or ST40-500 series core operating in 32-bit physical addressing mode, PPN32 is also read.

For a ST40-300 series core operating in 32-bit physical addressing mode, UB is also read.

UTLB data array write

PPN, V, SZ0, SZ1, PR, C, D, SH and WT specified in the stored value are written to the UTLB entry corresponding to the entry set in the ENTRY field of the address.

For a ST40-200, ST40-300 or ST40-500 series core operating in 29-bit physical addressing mode, PPN32 is ignored when generating physical addresses for translations and when a data array read subsequently occurs. The architecture does not define whether the actual value written in PPN32 becomes visible after a transition to 32-bit physical addressing mode.

3.10 Instruction TLB (ITLB)

The instruction translation look-aside buffer (ITLB) has four entries and is fully associative (that is, the translation for any virtual page may be stored in any of the entries.)

The ITLB acts as a cache of the most recently used instruction address translations. When an instruction address lookup results in no match in the ITLB, a lookup is attempted in the UTLB or PMB. If a match is found in the UTLB/PMB, an ITLB entry is automatically refilled from the contents of the matched UTLB entry.

3.10.1 ITLB operations

This section describes the operations that observe or affect the ITLB state.

Reading entries

The current contents of an entry may be read through the memory-mapped address and data arrays (see [Section 3.10.3 on page 82](#) and [Section 3.10.4 on page 84](#)). Note that the V bit is visible through both arrays.

Modifying a specified entry

A specific entry may be modified by writing to the memory-mapped address and data arrays. In this case, the ENTRY field of the address selects the entry to act on.

The V bit may be set through either the address array or the data array. The most recently written value is stored in the ITLB entry.

Globally clearing the ITLB

When a 1 is written to MMUCR.TI, the V bit of every ITLB entry is cleared. (This action also clears the V bit of every UTLB entry).

Clearing a ITLB entry for a specific page

If a specific page is to be invalidated, and it might be present in the ITLB, the UTLB *associative write* operation should be used (see [Clearing a UTLB entry for a specific page on page 71](#)).

Lookup

The hardware automatically performs a lookup in the ITLB when required.

Entry *N* in the ITLB is said to *match* if and only if

```
[ (ITLB[N].ASID == PTEH.ASID) ||
  ((SR.MD == 1) && (MMUCR.SV == 1)) ||
  (ITLB[N].SH == 1) ] &&
[ITLB[N].V == 1] &&
{ [(MMUCR.AT == 1) &&
  ((VA[31] == 0) || (VA[31:29] == 0b110)) &&
  (VA[31:20] == ITLB[N].VPN[31:20]) &&
  ((ITLB[N].SZ == 3) || (VA[19:16] == ITLB[N].VPN[19:16])) &&
  ((ITLB[N].SZ >= 2) || (VA[15:12] == ITLB[N].VPN[15:12])) &&
  ((ITLB[N].SZ >= 1) || (VA[11:10] == ITLB[N].VPN[11:10]))] ||
  (PASC.R.SE == 1) &&
  (VA[31:30] == 0b10) &&
```

```
(VA[31:29] == ITLB[N].VPN[31:29]) &&
((ITLB[N].SZ == 3) || (VA[28:27] == ITLB[N].VPN[28:27])) &&
((ITLB[N].SZ >= 2) || (VA[26] == ITLB[N].VPN[26])) &&
((ITLB[N].SZ >= 1) || (VA[25:24] == ITLB[N].VPN[25:24]))}]}
```

where VA is the virtual address being looked-up. Note that the interpretation of the page size bits depends on whether the lookup is for a virtual address in the P1/P2 regions or in the U0/P0/P3 regions.

If a single entry matches, the lookup is successful. When a lookup is successful, the remaining fields of the ITLB entry define the physical address, access permission and cache behavior used for the access.

If two or more entries match, a ITLBMULTIHIT is raised (which results in a reset of the ST40.)

If no entry matches, a look-up will be performed in the UTLB or PMB (depending on the address region involved.) See [Hardware refill from the UTLB](#) and [Hardware refill from the PMB](#) below.

Hardware refill from the UTLB

The ITLB is automatically refilled from the UTLB by the ST40 core if a miss occurs in the ITLB.

When an translatable instruction fetch occurs, the address is first looked up in the ITLB. If a hit occurs in the ITLB, the translation information is read from the matched entry. If no hit occurs in the ITLB, and the translation is for an instruction address in the U0/P0 or P3 regions, the lookup is repeated in the UTLB. If a match is found in the UTLB, the contents of the matching entry are copied into the ITLB. The selection of the ITLB entry to refill is described in [Section 3.10.2: ITLB LRU state on page 81](#). If no UTLB match is found, an ITLBMISS exception is delivered.

If multiple matches are found in the ITLB during the ITLB lookup, an ITLBMULTIHIT exception is generated. If multiple matches are found in the UTLB during the UTLB lookup, an ITLBMULTIHIT exception is generated.

When a matching UTLB entry is refilled into the ITLB field, the D, PR[5] and UB (if present) bits are discarded, since they do not exist in the ITLB.

Only the VPN bits which correspond to multiples of the page size are guaranteed to be updated during this refill operation. The lower-order VPN bits have undefined values after a refill. This does not affect subsequent ITLB lookups, because the lower-order bits are ignored during the lookup process (see [Lookup on page 79](#)) directly (see [Section 3.10.3: ITLB address array on page 82](#)).

Hardware refill from the PMB

Note: This mechanism is only relevant on the ST40-200, ST40-300 and ST40-500 series cores when operating in 32-bit physical addressing mode.

The ITLB is automatically refilled from the PMB by the ST40 core if a miss occurs in the ITLB.

When an translatable instruction fetch occurs, the address is first looked up in the ITLB. If a hit occurs in the ITLB, the translation information is read from the matched entry. If no hit occurs in the ITLB, and the translation is for an instruction address in the P1 or P2 regions, the lookup is repeated in the PMB. If a match is found in the PMB, the contents of the

matching entry are copied into the ITLB. The selection of the ITLB entry to refill is described in [Section 3.10.2](#). If no PMB match is found, an ITLBMULTIHIT exception is delivered.

If multiple matches are found in the ITLB during the ITLB lookup, an ITLBMULTIHIT exception is generated. If multiple matches are found in the PMB during the PMB lookup, an ITLBMULTIHIT exception is generated.

When a matching PMB entry is refilled into the ITLB field, certain fields are discarded or created as shown in [Table 30](#).

Only the VPN bits which correspond to multiples of the page size are guaranteed to be updated during this refill operation. The lower-order VPN bits have undefined values after a refill. This does not affect subsequent ITLB lookups, because the lower-order bits are ignored during the lookup process (see [Lookup on page 79](#)) directly (see [Section 3.10.3: ITLB address array](#)).

Table 30. Refill of ITLB entry from PMB entry

| ITLB field | Source of data | Notes |
|------------|-------------------------|---------------------------------|
| VPN[31:30] | 0b10 | |
| VPN[29:24] | VPN[29:24] of PMB entry | |
| VPN[23:10] | all zero | |
| V | 1 | Always valid following a refill |
| ASID | 0x00 | Not used |
| PPN[31:24] | PPN[31:24] of PMB entry | |
| PPN[23:10] | all zero | |
| SZ | SZ of PMB entry | |
| PR | 0 | User mode access never allowed |
| C | C of PMB entry | |
| SH | 1 | All PMB pages are shared |

3.10.2 ITLB LRU state

The choice of ITLB entry to refill after a UTLB (or PMB) match is made using a least-recently used (LRU) scheme. The LRU state is maintained in the LRUI bits of the MMUCR register (see [Section 3.8.5: MMU control register \(MMUCR\) on page 63](#)).

The LRU (least recently used) method is used to choose the ITLB entry to be replaced in the event of an ITLB miss. Information about the usage history of the 4 ITLB entries is encoded in the 6 bits of the MMUCR.LRUI field. When an ITLB entry is used in a lookup, bits [0] to [5] of the LRUI are updated according to [Table 31](#). A dash in this table means that no update is performed.

Table 31. MMUCR.LRUI modification after ITLB entry match

| Event | MMUCR.LRUI bit | | | | | |
|---------------------------|----------------|-----|-----|-----|-----|-----|
| | [5] | [4] | [3] | [2] | [1] | [0] |
| When ITLB entry 0 is used | 0 | 0 | 0 | - | - | - |
| When ITLB entry 1 is used | 1 | - | - | 0 | 0 | - |
| When ITLB entry 2 is used | - | 1 | - | 1 | - | 0 |
| When ITLB entry 3 is used | - | - | 1 | - | 1 | 1 |
| Other than the above | - | - | - | - | - | - |

The entry to be purged from the ITLB can be identified from the state of the LRUI bits. When a new translation needs to be added, the ITLB selects an entry according to [Table 32](#). An asterisk in this table means “don’t care”.

Table 32. Selection of ITLB entry that is replaced by a refill operation

| ITLB entry selected for refill | MMUCR.LRUI bit | | | | | |
|--------------------------------|--------------------|-----|-----|-----|-----|-----|
| | [5] | [4] | [3] | [2] | [1] | [0] |
| ITLB entry 0 is updated | 1 | 1 | 1 | * | * | * |
| ITLB entry 1 is updated | 0 | * | * | 1 | 1 | * |
| ITLB entry 2 is updated | * | 0 | * | 0 | * | 1 |
| ITLB entry 3 is updated | * | * | 0 | * | 0 | 0 |
| Other than the above | Setting prohibited | | | | | |

3.10.3 ITLB address array

The *ITLB address array* is allocated to virtual addresses 0xF200 0000 to 0xF2FF FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (MOV.L) must be used when accessing the array. Loads are used to examine the contents of the ITLB. Stores are used to update the contents.

There is no ITLB address array on the ST40-400. Stores to its address range are ignored and reads return zero.

On cores that support the ITLB, it may be accessed regardless of the state of the MMUCR.AT bit.

Access to the ITLB address array must adhere to the restrictions in [Section 3.15: MMU state coherency on page 105](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

The address is formed as shown in [Table 33](#).

Table 33. Addressing for the ITLB address array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|---|
| [31:24] | 8 | 0xF2 | Selects the ITLB address array within the P4 address region |
| [23:10] | 14 | zero | Unused bits. These should be zero for future compatibility |
| [9:8] | 2 | ENTRY | These bits define the entry in the ITLB that is to be acted upon (0..3) |
| [7:2] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 34](#).

Table 34. Fields in each ITLB address array entry

| ITLB address array entry | | | | |
|--------------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| ASID | [7:0] | 8 | Address space identifier | RW |
| | Operation | | When an address is to be translated, the value in PTEH.ASID (Section 3.8.1 on page 57) is compared with the ASID in each ITLB entry. Unless the page is shared or single virtual mode is active, the ASIDs must match for the ITLB entry to be considered a match | |
| | Reset | | Undefined | |
| V | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Reset | | Undefined | |
| Reserved | 9 | 1 | Reserved | RW |
| | Operation | | Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| VPN | [31:10] | 22 | Virtual page number | RW |
| | Operation | | Indicates the virtual page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the VPN will be ignored by the MMU during translation. For example, with a 64kB page, bits [15:10] of the VPN field will be ignored. | |
| | Reset | | Undefined | |

The following two kinds of operation can be used on the ITLB address array:

- ITLB address array read
VPN, V, and ASID are read from the ITLB entry corresponding to the entry set in the ENTRY field of the address.
- ITLB address array write
VPN, V, and ASID specified in the stored value are written to the ITLB entry corresponding to the entry set in the ENTRY field of the address.

3.10.4 ITLB data array

The ITLB data array is allocated to virtual addresses 0xF300 0000 to 0xF37F FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (`MOV.L`) must be used when accessing the array. Loads are used to examine the contents of the ITLB. Stores are used to update the contents.

There is no ITLB data array on the ST40-400. Stores to its address range are ignored and reads return zero.

Access to the ITLB data array must adhere to the restrictions in [Section 3.15: MMU state coherency on page 105](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations. There is no associative write facility for the ITLB data array.

A load returns the current settings for the specified entry.

A store updates all fields for the specified entry from the stored value. If an entry mapping P1 or P2 is set up, the behavior is undefined, if fields are configured inconsistently with regard to [Table 30](#), for example, if the SH bit is zero.

The address is formed as shown in [Table 35](#).

Table 35. Addressing for the ITLB data array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|--|
| [31:24] | 8 | 0xF3 | Selects the ITLB data array within the P4 address region |
| 23 | 1 | 0 | |
| [22:10] | 13 | zero | Unused bits. These should be zero for future compatibility |
| [9:8] | 6 | ENTRY | These bits define the entry in the ITLB that is to be acted upon (0..3). |
| [7:2] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 36](#).

Table 36. Fields in each ITLB data array entry

| ITLB data array entry | | | | |
|-----------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Reserved | 0 | 1 | Reserved | RW |
| | Operation | | Writes are ignored, reads return zero. | |
| | Reset | | Undefined | |
| SH | 1 | 1 | Private page or shared page | RW |
| | Operation | | Indicates whether the page is shared or private. 0: private page 1: shared page When a page is shared, its ASID is disregarded; only the virtual address comparison is used in determining whether the page matches or not. | |
| | Reset | | Undefined | |
| Reserved | 2 | 1 | Reserved | RW |
| | Operation | | Writes are ignored, reads return zero. | |
| | Reset | | Undefined | |
| C | 3 | 1 | Cacheability | RW |
| | Operation | | Indicates whether the page is cacheable 0: page is not cacheable 1: page is cacheable If CCR.ICE=0, instruction accesses will be uncached, regardless of the setting of this bit. | |
| | Reset | | Undefined | |
| SZ0 | 4 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 37 on page 87 for details. | |
| | Reset | | Undefined | |
| Reserved | 5 | 1 | Reserved | RW |
| | Operation | | Writes are ignored, reads return zero. | |
| | Reset | | Undefined | |
| PR | 6 | 1 | Page protection | RW |
| | Operation | | Defines the access rights to the page. See Table 38 on page 88 . | |
| | Reset | | Undefined | |
| SZ1 | 7 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 37 on page 87 for details. | |
| | Reset | | Undefined | |

Table 36. Fields in each ITLB data array entry (continued)

| ITLB data array entry | | | | |
|-----------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| V | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Reset | | Undefined | |
| Reserved | 9 | 1 | Reserved | RW |
| | Operation | | Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| PPN | [28:10] | 22 | Physical page number | RW |
| | Operation | | Indicates the physical page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the PPN will be ignored by the MMU during translation. For example, with a 64kB page, bits [15:10] of the PPN field will be ignored. | |
| | Reset | | Undefined | |
| Reserved | [31:29] | 3 | Reserved | RW |
| | Operation | | <i>This is the behavior on 100-series cores.</i> Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | <i>This is the behavior on 200- and 500-series cores.</i> These bits indicate the most significant 3 bits of the physical page number. When MMUCR.SE=1, writes set the value and reads return the current value. When MMUCR.SE=0, writes are discarded and reads return an undefined value. | |
| | Reset | | Undefined | |

Table 36. Fields in each ITLB data array entry (continued)

| ITLB data array entry | | | | |
|-----------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| PPN32 | [31:29] | 3 | Physical page number (32-bit extension) | RW |
| | Operation | | <i>This is the behavior on 300-series cores.</i> These bits indicate the most significant 3 bits of the physical page number. When PASC.R.SE=1, writes set the value and reads return the current value. When PASC.R.SE=0, writes set the current value and reads return zero. In this mode, any value is treated as zero when generating physical addresses for bus accesses. However, because writes are active, it allows UTLB entries to be correctly set up prior to switching the ST40 into space-enhancement mode. | |
| | Reset | | Undefined | |

Table 37 shows the page size behavior for the ITLB.

Table 37. ITLB page size behavior and active VPN, PPN address bits

| SE | VPN [31:30] | Region | SZ1 | SZ0 | Page size | Active VPN bits | Active PPN bits | Restrictions |
|----|----------------------|----------------------|-----|-----|-----------|-----------------|-----------------|--|
| 0 | 0b00 0b01 0b11 | U0/P0 U0/P0 P3 | 0 | 0 | 1 Kbyte | [31:10] | [28:10] | Only on 200-, 300- and 500-series cores. |
| | | | 0 | 1 | 4 Kbyte | [31:12] | [28:12] | |
| | | | 1 | 0 | 64 Kbyte | [31:16] | [28:16] | |
| | | | 1 | 1 | 1 Mbyte | [31:20] | [28:20] | |
| 1 | 0b00 0b01 0b11 | U0/P0 U0/P0 P3 | 0 | 0 | 1 Kbyte | [31:10] | [31:10] | |
| | | | 0 | 1 | 4 Kbyte | [31:12] | [31:12] | |
| | | | 1 | 0 | 64 Kbyte | [31:16] | [31:16] | |
| | | | 1 | 1 | 1 Mbyte | [31:20] | [31:20] | |
| 1 | 0b10 | P1/P2 | 0 | 0 | 16 Mbyte | [31:24] | [31:24] | |
| | | | 0 | 1 | 64 Mbyte | [31:26] | [31:26] | |
| | | | 1 | 0 | 128 Mbyte | [31:27] | [31:27] | |
| | | | 1 | 1 | 512 Mbyte | [31:29] | [31:29] | |

[Table 38](#) shows the types of access allowed to a page based on the page table entry's PR bits and the ST40 operating mode, and the type of exception raised if a type of access is not allowed.

Table 38. ITLB page access rights

| PR[6] | Privileged mode (SR.MD=1) | User mode (SR.MD=0) |
|-------|---------------------------|---------------------|
| | Instruction fetch | Instruction fetch |
| 0 | ✓ | EXECPROT |
| 1 | ✓ | ✓ |

The following two kinds of operation can be used on the ITLB data array:

ITLB data array read

PPN, V, SZ, PR, C, and SH are read from the ITLB entry corresponding to the entry set in the ENTRY field of the address.

For a ST40-200, ST40-300 or ST40-500 series core operating in 32-bit physical addressing mode, PPN32 is also read.

ITLB data array write

PPN, V, SZ, PR, C, and SH specified in the stored value are written to the ITLB entry corresponding to the entry set in the ENTRY field of the address.

For a ST40-200, ST40-300 or ST40-500 series core operating in 29-bit physical addressing mode, PPN32 is ignored when generating physical addresses for translations and when a data array read subsequently occurs. The architecture does not define whether the actual value written in PPN32 becomes visible after a transition to 32-bit physical addressing mode.

3.11 Privileged mapping buffer (PMB)

Note: The description of the PMB applies only to the ST40-200, ST40-300 and ST40-500 series cores.

The *privileged mapping buffer* (PMB) has 16 entries and is fully associative (that is, the translation for any virtual page may be stored in any of the entries).

The PMB is used to translate the virtual address of a data access to the P1 or P2 address regions, when space-enhancement mode (32-bit physical addressing) is enabled (SE=1).

The PMB is not used for translation when SE=0. However, the PMB state itself may be read and written whilst SE=0. This allows at least one PMB translation to be programmed prior to switching SE to 1.

In addition, if a miss occurs when looking up a P1 or P2 instruction address in the ITLB when SE=1, the lookup will be repeated in the PMB, and the matching entry's contents copied into the ITLB as described in section [Hardware refill from the PMB on page 80](#).

The PMB has no global invalidation mechanism apart from power-on reset. Each entry must be explicitly written to clear all the V bits.

All entries in the PMB are considered to be shared pages. There are no ASIDs in the PMB entries.

3.11.1 PMB operations

This section describes the operations that observe or affect the PMB state.

Reading entries

The current contents of an entry may be read through the memory-mapped address and data arrays (see [Section 3.11.2](#) and [Section 3.11.3 on page 91](#)). The V bit is visible through both arrays.

Modifying a specified entry

A specific entry may be modified by writing to the memory-mapped address and data arrays. In this case, the ENTRY field of the address selects the entry to act on.

The V bit may be set through either the address array or the data array. The most recently written value is stored in the PMB entry.

Lookup

The hardware automatically performs a lookup in the PMB when required.

Entry *N* in the PMB is said to *match* if and only if

```
[PMB[N].V == 1] &&
[PASCR.SE == 1] &&
[(VA[31:29] == PMB[N].VPN[31:29]) &&
 ((PMB[N].SZ == 3) || (VA[28:27] == PMB[N].VPN[28:27])) &&
 ((PMB[N].SZ >= 2) || (VA[26] == PMB[N].VPN[26])) &&
 ((PMB[N].SZ >= 1) || (VA[25:24] == PMB[N].VPN[25:24]))]
```

where *VA* is the virtual address being looked-up.

If a single entry matches, the lookup is successful. If no entry matches, or more than one entry matches, a ITLBMULTIHIT or OTLBMULTIHIT exception is raised. Either of these multihit exceptions will result in a reset of the ST40.

When a lookup is successful, the remaining fields of the PMB entry define the physical address, cache behavior and write buffering behavior used for the access.

3.11.2 PMB address array

The *PMB address array* is allocated to virtual addresses 0xF610 0000 to 0xF61F FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (MOV.L) must be used when accessing the array. Loads are used to examine the contents of the PMB. Stores are used to update the contents.

On cores that support the PMB, it may be accessed regardless of the state of the SE bit (MMUCR.SE for the ST40-200 and ST40-500 series and PASCR.SE for the ST40-300 series).

Access to the PMB address array must adhere to the restrictions in [Section 3.15: MMU state coherency on page 105](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

There is no associative write facility for the PMB.

The address is formed as shown in [Table 39](#).

Table 39. Addressing for the PMB address array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|---|
| [31:20] | 12 | 0xF61 | Selects the PMB address array within the P4 address region |
| [19:12] | 14 | zero | Unused bits. These should be zero for future compatibility |
| [11:8] | 4 | ENTRY | These bits define the entry in the PMB that is to be acted upon (0..15) |
| [7:2] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The fields in each address array entry are shown in [Table 40](#).

Table 40. Fields in each PMB address array entry

| PMB address array entry | | | | |
|-------------------------|----------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Reserved | [7:0] | 8 | Reserved | RW |
| | Operation | | Writes ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| V ⁽¹⁾ | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Power-on reset | | 0 | |
| | Manual reset | | Preserved | |
| Reserved | [23:9] | 15 | Reserved | RW |
| | Operation | | Writes are ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| VPN | [31:24] | 22 | Virtual page number | RW |
| | Operation | | Indicates the virtual page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the VPN will be ignored by the MMU during translation. For example, with a 64MB page, bits [25:24] of the VPN field will be ignored. Bits [31:30] of VPN must be programmed to 0b10. The behavior is undefined if any other value is programmed into an entry that also has V=1. | |
| | Reset | | Undefined | |

1. The V bit in the address array and data array are the same physical bit. The V bit may be set through either the address array or the data array; the most recently written bit is stored.

The following two kinds of operation can be used on the PMB address array:

- **PMB address array read**
VPN and V are read from the PMB entry corresponding to the entry set in the ENTRY field of the address.
- **PMB address array write**
VPN and V specified in the stored value are written to the PMB entry corresponding to the entry set in the ENTRY field of the address.

3.11.3 PMB data array

The *PMB data array* is allocated to virtual addresses 0xF710 0000 to 0xF71F FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (**MOV.L**) must be used when accessing the array. Loads are used to examine the contents of the PMB. Stores are used to update the contents.

The address used for a load or store to the array defines the entry number to be acted on for addressed operations. There is no associative write facility for the PMB data array.

Access to the PMB data array must adhere to the restrictions in [Section 3.15: MMU state coherency on page 105](#).

A load returns the current settings for the specified entry.

A store updates all fields for the specified entry from the stored value.

The address is formed as shown in [Table 41](#).

Table 41. Addressing for the PMB data array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|--|
| [31:20] | 8 | 0xF71 | Selects the PMB data array within the P4 address region |
| [19:12] | 8 | zero | Unused bits. These should be zero for future compatibility |
| [11:8] | 6 | ENTRY | These bits define the entry in the PMB that is to be acted upon (0..15). |
| [7:2] | 6 | zero | Unused bits. These should be zero for future compatibility |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 42](#).

Table 42. Fields in each PMB data array entry

| PMB data array entry | | | | |
|----------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| WT | 0 | 1 | Write-through or copy-back | RW |
| | Operation | | Indicates whether a cacheable translation has write-through or copy-back behavior in the operand cache. 0: copy-back behavior 1: write-through behavior This bit is ignored for non-cacheable translations (those where the C-bit is zero) and when the operand cache is off (CCR.OCE=0). | |
| | Reset | | Undefined | |
| Reserved | [2:1] | 1 | Reserved | RW |
| | Operation | | Writes are ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| C | 3 | 1 | Cacheability | RW |
| | Operation | | Indicates whether the page is cacheable 0: page is not cacheable 1: page is cacheable When a translation is used to map P4 resources that have virtual addresses starting 0xFF, this bit must be cleared to 0. If CCR.ICE=0 and/or CCR.OCE=0, instruction and data accesses respectively will be uncached, regardless of the setting of this bit. | |
| | Reset | | Undefined | |
| SZ0 | 4 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 43 for details. | |
| | Reset | | Undefined | |
| Reserved | [6:5] | 2 | Reserved | RW |
| | Operation | | Writes are ignored, reads return an undefined value. | |
| | Reset | | Undefined | |
| SZ1 | 7 | 1 | Page size bit | RW |
| | Operation | | Specify page size. Refer to Table 43 for details. | |
| | Reset | | Undefined | |

Table 42. Fields in each PMB data array entry (continued)

| PMB data array entry | | | | |
|----------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| V ⁽¹⁾ | 8 | 1 | Valid | RW |
| | Operation | | Indicates whether the entry is valid. 0: invalid 1: valid Only valid entries can be considered as matches during translation. | |
| | Reset | | Undefined | |
| Reserved | 9 | 1 | Reserved | RW |
| | Operation | | This is the behavior on all 100-, 200- and 500- series cores Writes are ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| UB | 9 | 1 | Unbuffered write control bit | RW |
| | Operation | | This is the behavior on the 300-series core. Specify whether uncacheable and write through stores must be unbuffered. Refer to Table 45 on page 104 for details. | |
| | Reset | | Undefined | |
| Reserved | [23:10] | 15 | Reserved | RW |
| | Operation | | Writes are ignored and reads return an undefined value. | |
| | Reset | | Undefined | |
| PPN | [31:24] | 22 | Physical page number | RW |
| | Operation | | Indicates the physical page number for the translation. Pages must always be aligned on a multiple of their size. Hence, depending on the page size, some lower-order bits of the PPN will be ignored by the MMU during translation. For example, with a 64MB page, bits [25:24] of the PPN field will be ignored. See Table 43 . | |
| | Reset | | Undefined | |

1. The V bit in the address array and data array are the same physical bit. The V bit may be set through either the address array or the data array; the most recently written bit is stored.

[Table 43](#) shows the page size behavior for the PMB.

Table 43. PMB page size behavior and active VPN, PPN address bits

| SE | VPN [31:30] | Region | SZ1 | SZ0 | Page size | Active VPN bits | Active PPN bits | Restrictions |
|----|-------------|--------|-----|-----|-----------|-----------------|-----------------|--|
| 1 | 0b10 | P1/P2 | 0 | 0 | 16 Mbyte | [31:24] | [31:24] | Only on 200-, 300- and 500-series cores. |
| | | | 0 | 1 | 64 Mbyte | [31:26] | [31:26] | |
| | | | 1 | 0 | 128 Mbyte | [31:27] | [31:27] | |
| | | | 1 | 1 | 512 Mbyte | [31:29] | [31:29] | |

The following two kinds of operation can be used on the PMB data array:

PMB data array read

For the ST40-200 and ST40-500 series cores: PPN, V, SZ, C and WT are read from the PMB entry corresponding to the entry set in the ENTRY field of the address.

For the ST40-300 series cores: PPN, UB, V, SZ, C and WT are read from the PMB entry corresponding to the entry set in the ENTRY field of the address.

PMB data array write

For the ST40-200 and ST40-500 series cores: PPN, V, SZ, C and WT specified in the stored value are written to the PMB entry corresponding to the entry set in the ENTRY field of the address.

For the ST40-300 series cores: PPN, UB, V, SZ, C and WT specified in the stored value are written to the PMB entry corresponding to the entry set in the ENTRY field of the address.

3.12 Handling MMU exceptions

There are nine MMU related exceptions:

| | |
|--------------|---|
| ITLBMULTIHIT | Instruction TLB multiple-hit exception. |
| OTLBMULTIHIT | Operand TLB multiple-hit exception. |
| RTLBMISS | Read data TLB miss exception. |
| WTLBMISS | Write data TLB miss exception. |
| ITLBMISS | Instruction TLB miss exception. |
| FIRSTWRITE | Initial page write exception. |
| READPROT | Read data TLB protection violation exception. |
| EXECPROT | Instruction TLB protection violation exception. |
| WRITEPROT | Write data TLB protection violation exception. |

These exceptions can only occur when an MMU is present, a description of how each is handled by the hardware can be found in [Section 5.5: Description of exceptions on page 153](#). This section describes how one would expect the exceptions to be handled by an operating system kernel implementing virtual memory.

3.12.1 ITLBMULTIHIT

An instruction TLB multiple hit exception occurs when, more than one ITLB entry matches the virtual address to which an instruction access has been made. If multiple hits occur when the UTLB or PMB is searched by hardware, in hardware ITLB miss handling, a instruction TLB multiple hit exception will also result.

When an instruction TLB multiple hit exception occurs a reset is executed, and cache coherency is not guaranteed.

Hardware processing

See [Chapter 5: Exceptions, ITLBMULTIHIT - Instruction TLB Multiple-Hit Exception on page 155](#).

Software processing (reset routine)

The ITLB entries which caused the multiple hit exception are checked in the reset handling routine. This exception is intended for use in program debugging, and should not normally be generated.

3.12.2 ITLBMISS

An instruction TLB miss exception occurs when, address translation information for the virtual address to which an instruction access is made, is not found in the UTLB entries by the hardware ITLB miss handling procedure. The instruction TLB miss exception processing, carried out by software, is shown below. This is the same as the processing for a data TLB miss exception.

Hardware processing

See [Section 5.5.2: General exceptions on page 156](#).

Software processing (instruction TLB miss exception handling routine)

Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

1. Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.
2. When the entry to be replaced is specified explicitly by software, write that value to URC in the MMUCR register. If the URC is selected is greater than URB, once the LDTLB instruction has been issued, the URC should be changed back to an appropriate value.
3. Execute the LDTLB instruction, which will write the contents of PTEH and PTEL to the TLB.
4. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

3.12.3 EXECPROT

An instruction TLB protection violation exception occurs when, even though an ITLB entry contains address translation information matching the virtual address to which an instruction access is made, the actual access type is not permitted by the access right specified by the PR bit. The instruction TLB protection violation exception processing, carried out by software, is shown in the following text.

Hardware processing

See [Section 5.5.2: General exceptions on page 156](#).

Software processing (instruction TLB protection violation exception handling routine)

Resolve the instruction TLB protection violation, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

3.12.4 OTLBMULTIHIT

An operand TLB multiple hit exception occurs when more than one UTLB entry matches the virtual address to which a data access has been made.

When an operand TLB multiple hit exception occurs, a reset is executed, and cache coherency is not guaranteed. The contents of PPN in the UTLB prior to the exception may also be corrupted.

An operand TLB multiple hit exception also occurs when a data lookup is made in the PMB and either no matches or multiple matches are found.

Hardware processing

See [Chapter 5: Exceptions, OTLBMULTIHIT - operand TLB multiple-hit exception on page 156](#).

Software processing (reset routine)

The UTLB entries which caused the multiple hit exception are checked in the reset handling routine. This exception is intended for use in program debugging, and should not normally be generated.

3.12.5 WTLBMISS/RTLBMIS

A data TLB miss exception occurs when, address translation information for the virtual address to which a data access is made is not found in the UTLB entries. The data TLB miss exception processing, carried out by software, is shown in the following text.

Hardware processing

See [Section 5.5.2: General exceptions on page 156](#).

Software processing (data TLB miss exception handling routine)

Software is responsible for searching the external memory page table and assigning the necessary page table entry. Software should carry out the following processing in order to find and assign the necessary page table entry.

1. Write to PTEL the values of the PPN, PR, SZ, C, D, SH, V, and WT bits in the page table entry recorded in the external memory address translation table.
2. When the entry to be replaced is specified explicitly by software, write that value to the URC field in the MMUCR register. If the URC is greater than URB, once the LDTLB instruction has been issued the URC should be changed back to an appropriate value.
3. Execute the LDTLB instruction, which will write the contents of PTEH and PTEL to the UTLB.
4. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The **RTE** instruction should be issued at least one instruction after the LDTLB instruction.

3.12.6 READPROT/WRITEPROT

A data TLB protection violation exception occurs when, even though a UTLB entry contains address translation information matching the virtual address to which a data access is made, the actual access type is not permitted by the access right specified by the PR bit. The data TLB protection violation exception processing, carried out by software, is shown in the following text.

Hardware processing

See [Section 5.5.2: General exceptions on page 156](#).

Software processing (data TLB protection violation exception handling routine)

Resolve the data TLB protection violation, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow. The RTE instruction should be issued at least one instruction after the LDTLB instruction.

3.12.7 FIRSTWRITE

An initial page write exception occurs when, the D bit is 0 even though a UTLB entry contains address translation information matching the virtual address to which a data access (write) is made, and the access is permitted. The initial page write exception processing, carried out by software, is shown in the following text.

Hardware processing

See [Section 5.5.2: General exceptions on page 156](#).

Software processing (initial page write exception handling routine)

The following processing should be carried out as the responsibility of software.

1. Retrieve the address on which the fault occurred from the PTEH register.
2. Perform any steps necessary to allow the memory page to become writable (for example, copy-on-write processing in a typical operating system).
3. Modify the value read from PTEH to set the D-bit to 1.
4. Perform an associative write to the UTLB to set the D-bit of the matching page entry.
5. Finally, execute the exception handling return instruction (RTE), terminate the exception handling routine, and return control to the normal flow.

3.13 Interaction of MMU and cache

The ST40 caches use virtual address indexing. This means that bits from the virtual address are used to determine the cache set where the associated data may be stored.

If there are bits common to both the cache set indexing and TLB address translation, a phenomenon termed *cache synonyms* may arise. In this situation, the same physical memory may become cached in more than one cache set. [Table 44](#) shows the bits common to indexing and translation for various cache sizes and the two page sizes that are relevant (for the range of cache sizes available on the ST40), namely 4 Kbytes and 1 Kbyte.

Table 44. Synonym bits as a function of cache size and page size

| Way size | Total cache size | | | Number of cache sets | Page size | | | |
|--|-------------------------|-------------------------|-----------|----------------------|--------------|---------|-----------------------------|---------|
| | | | | | 4 Kbyte | 1 Kbyte | 4 Kbyte | 1 Kbyte |
| | Direct-mapped | 2-way | 4-way | | Synonym bits | | Number of possible synonyms | |
| 2 Kbyte | 2 Kbyte | 4 Kbyte | 8 Kbyte | 64 | - | [10] | - | 2 |
| 4 Kbyte | 4 Kbyte | 8 Kbyte | 16 Kbyte | 128 | - | [11:10] | - | 4 |
| 8 Kbyte | 8 Kbyte ⁽¹⁾ | 16 Kbyte ⁽²⁾ | 32 Kbyte | 256 | [12] | [12:10] | 2 | 8 |
| 16 Kbyte | 16 Kbyte ⁽³⁾ | 32 Kbyte ⁽⁴⁾ | 64 Kbyte | 512 | [13:12] | [13:10] | 4 | 16 |
| 32 Kbyte | 32 Kbyte | 64 Kbyte | 128 Kbyte | 1024 | [14:12] | [14:10] | 8 | 32 |
| 64 Kbyte | 64 Kbyte | 128 Kbyte | 256 Kbyte | 2048 | [15:12] | [15:10] | 16 | 64 |
| Number of sets between potential synonyms: | | | | | 128 | 32 | | |

1. For example, ST40-103 instruction cache
2. For example, ST40-202 instruction cache
3. For example, ST40-103 operand cache
4. For example, ST40-202 operand cache; ST40-300 instruction and operand caches

As an example, consider a 32Kbyte 2-way cache when 4Kbyte pages are being used. A 32-byte block of physical memory that has address bits [11:5] all zero can potentially be loaded into any of the cache sets numbered 0, 128, 256 or 384, depending on the value of bits [13:12] of the virtual address used to access it.

For the instruction cache, the result of cache synonyms is merely a loss of useful capacity, because the duplicate copies are wasteful. A similar situation applies to read-only data in the operand cache.

However, for writeable data in the operand cache, synonyms can make it impossible to guarantee data integrity: if two different lines caching the same physical address are both written to, it is timing-dependent which one will be written back to memory last and hence the program behavior will be unpredictable.

Great care must therefore be taken whenever translations are set up which could cause synonyms. In particular, if two operand translations are to the same physical page but their virtual addresses differ in their synonym bits, either

- do not allow both the translations to be active at the same time
- always separate activations of the two translations by an appropriate cache purge

Note: The effect can also occur when one of the accesses is translated in a TLB and the other is through an identity mapping or masking operation. For example, with SE=0 and AT=1, translated accesses through U0/P0 virtual addresses can cause synonyms in conjunction with accesses to the same physical memory through P1 virtual addresses.

3.14 Memory coherency

The ST40 cores all have some store buffering at their STBus interface to allow the CPU to continue executing whilst stores are written out in parallel. The degree of buffering varies between core families. In addition, the ST40 bus interface and/or the STBus interconnect may introduce re-ordering of stores or merging of multiple stores to the same quadword into one (so-called write-combining).

Because stores may be deferred in this way, there are situations when software needs to know that a store has definitely been committed to the target device. Examples of such situations are:

- when software is writing to peripheral device registers, it needs to know that the stores will reach the peripheral in the order they were issued (otherwise the peripheral may behave incorrectly)
- when setting up data in memory that will be accessed by a DMA engine, software needs to know that the data is actually in the memory before initiating the DMA operation
- when storing instruction code into memory, software needs to know that the instructions will be visible to the instruction fetch hardware prior to branching to those instructions

3.14.1 Coherency mechanisms on ST40-100, ST40-200, ST40-400 and ST40-500 series cores

These ST40 variants issue loads and stores to the bus in order.

Store coherency

To ensure a store to a memory location has completed, software can read back the value from the same address.

Instruction coherency

When instruction memory is modified, care must be taken to ensure that the newly written instructions are correctly executed when the PC reaches the addresses of the modified instructions.

For guaranteed operation, software must handle the following issues.

1. There is no inter-lock in hardware from the data load/store pipeline to the instruction fetch. Therefore, the modified instructions must be written out to memory^(b) to make them visible to a subsequent instruction fetch.
2. The instruction cache may contain the previous instructions from the modified address(es). If this is the case, steps must be taken to ensure the stale copy in the instruction cache cannot be used when execution from the modified addresses occurs.
3. For performance reasons, the ST40 pre-fetches instructions; see [Section 3.16: Side effects of instruction pre-fetch on page 107](#). If the address of a modified instruction is executed shortly after the modification, it may be possible for the old instruction to already be partly-executed within the CPU pipeline.

The following sections describe how software may handle these issues.

Modifications to instruction memory made from privileged mode

For step [1.](#) above, software must write the modified instructions to memory^(c). Any of the following may be used:

- uncached stores
- stores through cacheable memory in write-through mode
- stores through cacheable memory in copy-back mode. A subsequent write of the cache lines to backing memory has to be assured by the use of one of:
 - **OCBP** instructions applied to every cache line in the modified range
 - **OCBWB** instructions applied to every cache line in the modified range
 - a purge of the entire operand cache
- the store queues

After the stores, software should perform a load (cached or uncached) from within the range written to memory. The STBus ordering of this load following the stores will ensure that all the stores have reached the memory before the following steps occur. If the **OCBWB** instruction was used to cause writes from the operand cache, this load must use an uncached virtual address, to ensure that it cannot be serviced from the operand cache.

For step [2.](#), software may use one of the following.

- Invalidation of the entire instruction cache using the CCR.ICI bit (see [Section 4.2.1: Cache control register \(CCR\) on page 116](#))
- Use of the instruction cache memory-mapped address array (see [Section 4.5.1: IC address array on page 133](#)). By clearing the appropriate V-bits of entries in this array, the appropriate lines are invalidated.

Step [2.](#) may be skipped if software can be absolutely certain that the affected addresses cannot be in the instruction cache. This would be the case, for example, if the ST40 had

-
- b. If the device contains a unified level-2 cache outside the ST40, it is sufficient if the instructions are at least written into the level-2 cache.
 - c. or level-2 cache.

operated with the instruction cache off since the most recent invalidation of the cache. To determine whether addresses could be in the instruction cache whilst operating with the cache enabled, refer to [Conditions when an instruction may be pre-fetched on page 102](#).

For step 3., privileged mode software may use any of the following:

- an RTE instruction
- a LDC Rm, SR instruction
- the launch of any exception or interrupt

The methods described below for achieving steps 2. and 3. from user mode may also be used.

Modifications to instruction memory from user mode

For step 1., the handling is the same as that shown for privileged mode above. For software that will run under an operating system, it should be assumed that the memory involved is in copy-back mode, and that the stores have to be followed by OCBWB or OCBP instructions.

For step 2., software must ensure the modified address(es) are not in the instruction cache by generating capacity conflicts in the instruction cache. Such conflicts will cause any cache entries corresponding to the modified addresses to be discarded.

To illustrate how capacity conflicts might be produced, suppose that a modified instruction will be executed at the virtual address VA, which maps to the physical address PA.

For a direct-mapped instruction cache, select a virtual address VA1 and physical address PA1 such that:

- a) VA1 maps to PA1, and
- b) VA1 is cacheable, and
- c) PA1[31:10] is not equal to PA[31:10], and
- d) the set selector bits of VA and VA1 are the same, where the set selector bits are defined by [Table 65: Instruction cache - virtual address bits used for set selection in various conditions on page 128](#) in [Section 4.4.1: Configuration on page 128](#).

For an 8kB direct-mapped instruction cache, step (d) is achieved if:

$VA[12:5] == VA1[12:5]$

or, in terms of C-like syntax:

$(VA \ \& \ 0x1FE0) == (VA1 \ \& \ 0x1FE0)$

then execute at least one instruction from the cache line containing VA1.

This is most easily achieved if:

- a contiguous region of virtual addresses is set aside
- the region is equal to the size of the instruction cache, and
- the physical addresses that map the region are disjoint from the regions used by the normal instructions

Then there is a unique 32-byte range of virtual addresses within the region that fulfils the properties required for VA1.

For a 2-way associative instruction cache, select two virtual addresses VA1 and VA2 with physical addresses PA1 and PA2 such that all the following are true:

- a) VA1 maps to PA1
- b) VA2 maps to PA2
- c) VA1 is cacheable
- d) VA2 is cacheable
- e) the values PA1[31:10], PA2[31:10] and PA[31:10] are all different
- f) the set selector bits of VA, VA1 and VA2 are the same, where the set selector bits are defined by [Table 65: Instruction cache - virtual address bits used for set selection in various conditions on page 128](#) in [Section 4.4.1: Configuration on page 128](#).

For an 16kB 2-way set associative instruction cache, step (f) is achieved if:

$VA[12:5] == VA1[12:5] == VA2[12:5]$

or, in terms of C-like syntax:

$(VA \ \& \ 0x1FE0) == (VA1 \ \& \ 0x1FE0) == (VA2 \ \& \ 0x1FE0)$

then execute at least one instruction from the cache line containing VA1 and at least one instruction from the cache line containing VA2.

This is most easily achieved if a range of virtual addresses is set aside having the same properties as those listed above for direct-mapped caches. Then there are precisely two 32-byte ranges of virtual addresses within the region that fulfil the properties required for VA1 and VA2. For a 16kB 2-way set associative instruction cache, the required addresses VA1 and VA2 are 8kB apart.

Step 2. can be avoided if the software can be sure, by some means, that the modified addresses cannot possibly be in the instruction cache. Refer to the following section: [Conditions when an instruction may be pre-fetched on page 102](#).

For step 3., software must ensure that at least 16 instructions are executed between the load at the end of step 1 and the execution of the first modified instruction. These 16 instructions may include the instructions used to implement step 2.

Conditions when an instruction may be pre-fetched

In step 2 above, it may be useful for software to be able to determine whether an address might have entered the instruction cache due to instruction pre-fetching.

A physical address “PA1” may enter the instruction cache when:

- a cacheable virtual address VA2 is pre-fetched (see below), and
- VA2 maps to a physical address PA2, and
- PA2[31:5] and PA1[31:5] are the same (i.e. they are in the same 32-byte region of memory.)

An address “A” can be pre-fetched for any of the following:

- the current PC is “P” and “A” lies in the range [P+2, P+32], regardless of whether any branches or exceptions occur within that interval
- the current PC is a delayed branch whose target is “T”, an exception occurs in the delay slot of the branch, and “A” lies in the range [T, T+30]
- the current PC “T” is the target of a branch that has just been taken, “T” experiences any type of exception, and “A” lies in the range [T, T+30]. (This is actually a subset of the first case in this list, but is shown separately for emphasis.)

3.14.2 Coherency mechanisms on ST40-300 series cores

These cores ensure coherency using the **SYNCO** and **ICBI** instructions and the UB bits in the TLB, PMB and PASCRC registers.

Store coherency: SYNCO instruction

The **SYNCO** instruction provides a data memory barrier. Software can use it to guarantee that every outstanding data operation has completed. If the ST40-300 has previously issued any load or store request to the STBus for which the response is still awaited, the **SYNCO** instruction will cause the ST40-300 to pause until all such responses have arrived. Any data access made by software after the **SYNCO** is guaranteed to see the effects of all loads and stores that preceded the **SYNCO**.

The **SYNCO** instruction does not guarantee instruction synchronization. There may be instructions in the ST40-300 pipeline that were fetched prior to the **SYNCO** instruction waiting for the bus responses. See [Instruction coherency: ICBI instruction](#) below.

The **SYNCO** instruction is only available on the ST40-300 core (that is, not on any earlier variant).

Instruction coherency: ICBI instruction

The **ICBI @Rn** instruction is provided for achieving instruction fetch coherency with respect to memory modifications made through stores or by other on-chip bus masters. It is described in [Section 4.4.4: Explicit cache controls on page 130](#).

Write buffering, re-ordering and combining: UB bits

The UTLB, PMB and PASCRC register contain UB bits (see [Table 27 on page 75](#), [Table 42 on page 92](#) and [Table 22 on page 67](#) respectively). These UB bits control whether uncacheable and write-through stores must be *unbuffered*, or whether they may be *buffered*.

If stores are unbuffered, the architecture guarantees that no write-combining or write-reordering occurs in the ST40 bus interface or in the STBus interconnect.

If stores are buffered, they may be subject to write-combining or write-reordering by either the ST40 bus interface or the STBus interconnect. (However, there is no guarantee that write-combining or write-reordering will actually occur; the behavior may in reality be equivalent to unbuffered writes.)

The UB bits do not control the effect on stores that are cacheable in copy-back mode. In particular, neither of the following writes to the bus are affected by the UB bit settings:

- cache lines written back to memory as the result of a store miss
- cache lines written back to memory as the result of an **OCBP** or **OCBWB** instruction

The UB bits provide a less strong guarantee of coherency than the **SYNCO** instruction, see [Store coherency: SYNCO instruction on page 103](#).

The assignment of the states 0 and 1 to the UB bits is defined in [Table 45](#).

The word completed in [Table 45](#) means that the ST40 has received a STBus response corresponding to the earlier store request. The STBus router should be designed to make this response a guarantee that any later access to the store target by any initiator is certain to observe the effect of the store.

Table 45. Definition of UB bits

| UB bit value | Meaning |
|--------------|---|
| 0 | The CPU may initiate a later bus access before the earlier store has completed, that is buffered. |
| 1 | The CPU must wait for the earlier store to be completed before initiating any later bus access, that is unbuffered. |

When 29-bit physical addressing is selected (PASC.R.SE=0), unbuffered or buffered writes may be selected individually for each of the 64 Mbyte segments comprising the 512 Mbyte physical address space. Software configures these settings using PASC.R.UB[7:0], where [Table 46](#) shows the relationship between the PASC.R.UB bits and the 29-bit physical address space.

Table 46. Relationship of PASC.R.UB bits to 29-bit physical addresses

| Bit in PASC.R.UB | Physical address start | Physical address end | Notes |
|------------------|------------------------|----------------------|---|
| 0 | 0x0000 0000 | 0x03FF FFFF | Memory or peripherals (device-dependent; consult product-specific documentation). |
| 1 | 0x0400 0000 | 0x07FF FFFF | |
| 2 | 0x0800 0000 | 0x0BFF FFFF | |
| 3 | 0x0C00 0000 | 0x0FFF FFFF | |
| 4 | 0x1000 0000 | 0x13FF FFFF | |
| 5 | 0x1400 0000 | 0x17FF FFFF | |
| 6 | 0x1800 0000 | 0x1BFF FFFF | Peripherals accessed through P4 addresses 0xFC00 0000 - 0xFFFF FFFF or by a UTLB translation. |
| 7 | 0x1C00 0000 | 0x1FFF FFFF | |

The UB bits in the UTLB and PMB entries are only used if space-enhancement mode (32-bit physical addressing) is enabled (PASC.R.SE=1). These UB bit for a page entry indicates whether stores to that page are required to be unbuffered.

The selection of the applicable UB bit is shown in [Table 47](#).

Table 47. UB bit derivation from address region and MMU operating mode

| Virtual address region | PASCR.SE | MMUCR.AT | Source of UB bit defining store behavior |
|------------------------|----------|----------|---|
| U0/P0, P3 | 0 | 0 or 1 | Bit of PASCR.UB[7:0] corresponding to the physical address of the store (see Table 46) |
| | 1 | 0 | UB bit is always treated as 1 in this case |
| | | 1 | UB bit of UTLB entry matched in translation |
| P1, P2 | 0 | 0 or 1 | Bit of PASCR.UB[7:0] corresponding to the physical address of the store (see Table 46) |
| | 1 | | UB bit of PMB entry matched in translation |
| P4 | 0 or 1 | 0 or 1 | From PASCR.UB[7] |

3.15 MMU state coherency

This section describes the requirements for MMU coherency and means of achieving it.

3.15.1 Coherency requirements

Software must ensure coherence between the instruction stream and access to the MMU state by adhering to restrictions regarding the address region and cacheability of the instruction address of the **MOV.L** instruction used to access the MMU state. If these restrictions are not followed:

- when the MMU state is changed by software, some subsequent instructions may be executed without taking account of the state change
- when the MMU state is read by software, an inconsistent value may be read
- on the ST40-100, ST40-200, ST40-400 and ST40-500 series cores, it is also possible for the **MOV.L** instruction which reads or writes the MMU state to interact with nearby instructions so as to put the MMU into an undefined state

The restrictions define the virtual address region in which the PC must be at the time of the **MOV.L**, the instruction cacheability of that region, and the precautions that must be taken afterwards before making an instruction or data access to another region. [Table 48](#) shows these restrictions on PC placement and later access to other regions.

[Table 48](#) shows P4 as a valid region because some devices have memory in the P4 region that can support uncached instruction fetch.

Table 48. Coherency requirements for access to MMU state

| Resource | Operation | PC of accessing instruction must be in | | Cohere later instruction access to | | Cohere later data access to | |
|---|-----------------------------|--|-----------------------------------|------------------------------------|-------------------|------------------------------|-------------------|
| | | 29-bit (SE=0) | 32-bit (SE=1) | U0/P0, P3 | P1 ⁽²⁾ | U0/P0, P3, SQ ⁽¹⁾ | P1 ⁽²⁾ |
| MMUCR ⁽³⁾ , PTEH, PASCRC ⁽³⁾ , IRMCRC | read | no restriction | no restriction | N/A | N/A | N/A | N/A |
| | write | P1, P2 or P4 | P1, P2 or P4 | Yes | No | Yes | No |
| UTLB | update with LDTLB | P1, P2 or P4 | P1, P2 or P4 | No | No | Yes | No |
| UTLB, ITLB and PMB | memory-mapped read or write | P2 or P4 | P1 or P2 (uncacheable page) or P4 | Yes | Yes | Yes | Yes |

1. SQ = store-queues

2. Or a cacheable page in P2 when in space-enhancement mode

3. Refer to [Section 3.4.3](#), [Section 3.4.4](#) and [Section 3.4.5 on page 40](#) for additional restrictions on modifying PASCRC.SE or MMUCRC.SE.

3.15.2 Achieving coherency (ST40-100, ST40-200, ST40-400 and ST40-500 series cores)

For ST40-100, ST40-200, ST40-400 and ST40-500 series cores, coherency with subsequent instruction fetches and data accesses is achieved by any one of the following:

- executing at least four instructions (typically **NOP** instructions) before making a data access to a region that requires later accesses to be cohered, as defined in [Table 48](#)
- executing at least eight instructions (typically **NOP** instructions) before making an instruction fetch to a region that requires later accesses to be cohered, as defined in [Table 48](#)
- executing an **RTE** instruction
- executing a **LDC Rm, SR** instruction
- launching the handler for any type of interrupt or exception

3.15.3 Achieving coherency (ST40-300 series cores)

For the ST40-300 series cores, coherency with subsequent instruction fetches and data accesses is achieved by any one of the following:

- executing an **RTE** instruction
- executing an **ICBI** instruction for an address in any region, not necessarily cacheable
- executing a **LDC Rm, SR** instruction
- launching the handler for any type of interrupt or exception
- using the appropriate bit in the IRMCRC ([Section 3.8.7: Instruction refetch inhibit control \(IRMCRC\) on page 68](#)) to force a refetch of the next instruction for the particular type of MMU change being made

- Note:
- 1 The **IRMCR** is primarily a means to allow legacy software to run safely on the ST40-300 series cores. One of the other methods is to be preferred for new software.
 - 2 There is no need to disable interrupts around an access to an MMU resource solely to guard against coherency hazards.

3.16 Side effects of instruction pre-fetch

This section describes the side effects that can arise during instruction pre-fetch operations.

3.16.1 Inline fetching near ends of memory regions

For performance reasons, all ST40s have an internal buffer for holding pre-fetched instructions. Consequently, there is a risk that untranslated instruction pre-fetches could run beyond the limits of the memory allocated to code. To prevent this, program code must not be located in:

- the final 36 bytes of any memory area for the ST40-300 series
- the final 24 bytes of any memory area for the other ST40 series

If code is located in the final bytes of a memory area, as defined above, instruction pre-fetching may initiate a bus access for an address outside the memory area. An example of this problem is shown in [Table 49](#).

Table 49. Example of instruction pre-fetch crossing a memory boundary

| Address | Instruction | Notes |
|-------------|-------------|---------------------------------------|
| 0x03FF FFF8 | ADD R1, R4 | PC of currently executing instruction |
| 0x03FF FFFA | JMP @R2 | |
| 0x03FF FFFC | NOP | |
| 0x03FF FFFE | NOP | |
| 0x0400 0000 | | Instruction prefetch address |
| 0x0400 0002 | | |

In the example, the end of the memory area is 0x03FF FFFF. Before the **JMP** has taken effect, the instruction pre-fetch may initiate bus accesses to 0x0400 0000 and beyond. The behavior in this situation is undefined and will depend on what resource, if any, starts at address 0x0400 0000.

Note: Architecturally, the **JMP** causes execution to remain safely inside the memory area.

Instruction prefetch side effects

1. It is possible that an external bus access caused by an instruction prefetch may result in mis-operation of an external device, such as a FIFO, connected to the area concerned.
2. If there is no device to reply to an external bus request caused by an instruction prefetch, hang-up will occur.

Remedies

1. These illegal instruction fetches can be avoided by using the MMU. Instruction pre-fetch will never occur for a translated address lying outside the pages currently mapped by the ITLB or UTLB/PMB.
2. For untranslated (real mode) addresses, the problem can be avoided by not locating program code in the last 256 bytes of a memory area.

3.16.2 Branches following exceptions

For performance reasons, all ST40s fetch the targets of taken branches before being certain that all instructions prior to the branch are free of exceptions.

In most cases, compilers and other software tools generate the code sequence assuming that it is exception-free at runtime, except in certain well-defined cases. Consequently, branches following exceptions are normally such that their target addresses are valid and safe to fetch from, even though the branch will not take effect due to the exception before it.

However, there are situations in which an unexpected branch can be decoded, even though it architecturally it would never be part of the program. These situations can arise when an exception occurs near the end of a block of instructions that is followed by a block of data. If a word amongst the data has a bit pattern that appears to be a branch instruction, a prefetch of the target of that apparent branch instruction may be initiated before the earlier exception takes effect and cancels it.

[Table 50](#) and [Table 51 on page 109](#) define what happens in cases such as:

```
...
Instruction "X"
...
Branch "Y"
...
```

where "Y" can be any form of branch (**BT**, **BT/S**, **BF**, **BF/S**, **BRA**, **BRAF**, **JMP**, **BSR**, **BSRF**, **JSR**, **RTS**).

"X" can be either :

- a single instruction
- a branch (separate from "Y") and its delay slot considered together

For the ST40-300 series, there may be one or more branches in the interval between "X" and "Y", as long as "Y" lies on the predicted path through all of them.

For the ST40-100, ST40-200, ST40-400 and ST40-500 series, there may be one or more not-taken **BT**, **BT/S**, **BF** or **BF/S** branches in the interval between "X" and "Y".

Suppose an exception occurs at "X". [Table 50](#) groups the possible types of exception at "X" into three behavioral classes. [Table 51](#) defines whether the target address of the branch "Y" can potentially be prefetched or not. This depends on the behavioral class from [Table 50](#), and what "X" is (taken branch, not-taken branch, or single instruction).

In addition, a pre-fetch of the target of “Y” does **not** occur if any of the following three conditions are true:

- there are at least six instructions in the interval between “X” and “Y” (for ST40-100, ST40-200, ST40-400 and ST40-500 series)
- there are at least eight instructions in the interval between “X” and “Y” (for ST40-300 series)
- “Y” is a **BRAF**, **JMP**, **BSRF**, **JSR** or **RTS** instruction, and the register supplying the branch target address is itself the destination operand of a load instruction (**MOV.L** or **LDS.L**), which is in the interval between “X” and “Y”.

Table 50. Table classes for fetch suppression

| Class | Exceptions |
|-------|---|
| A | ITLBMULTIHIT, IADDERR, ITLBMISS, EXECPROT, TRAP |
| B | OTLBMULTIHIT, UBRKBEFORE, RESINST, FPUDIS, RADDERR, WADDERR, RTLBMISS, WTLBMISS, READPROT, WRITEPROT, FPUEXC, FIRSTWRITE, UBRKAFTER |
| C | ILLSLOT, SLOTFPUDIS |

Table 51. Exception class definitions

| Scenario | Prefetch of branch “Y” is: | |
|-----------------------------------|----------------------------|--------------------|
| | Suppressed by classes | Allowed by classes |
| Taken branch or delay slot | A B C | - |
| Nontaken branch or delay slot | A | B C |
| Single instruction ⁽¹⁾ | A | B |

1. Class C not applicable.

Figure 13. Real mode address map (AT=0)

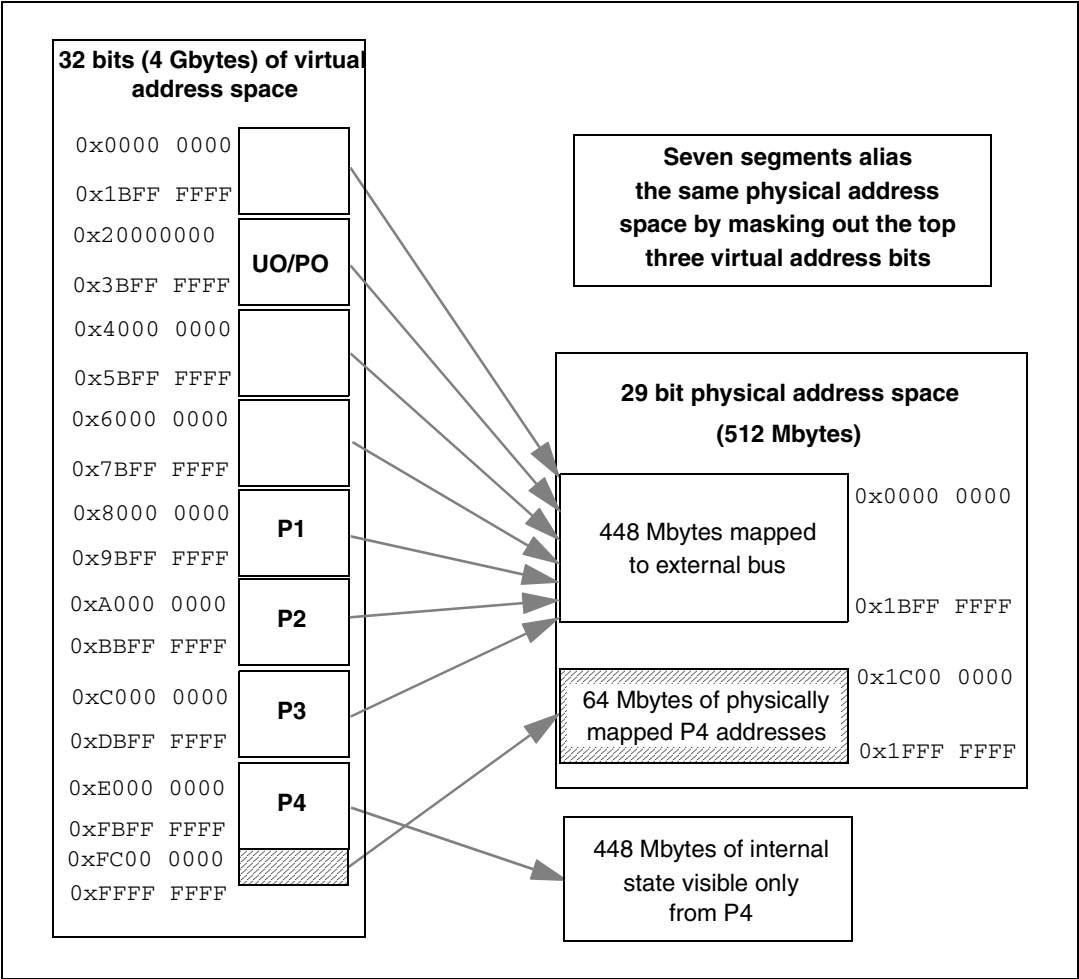


Figure 14. Translated mode address map (AT=1)

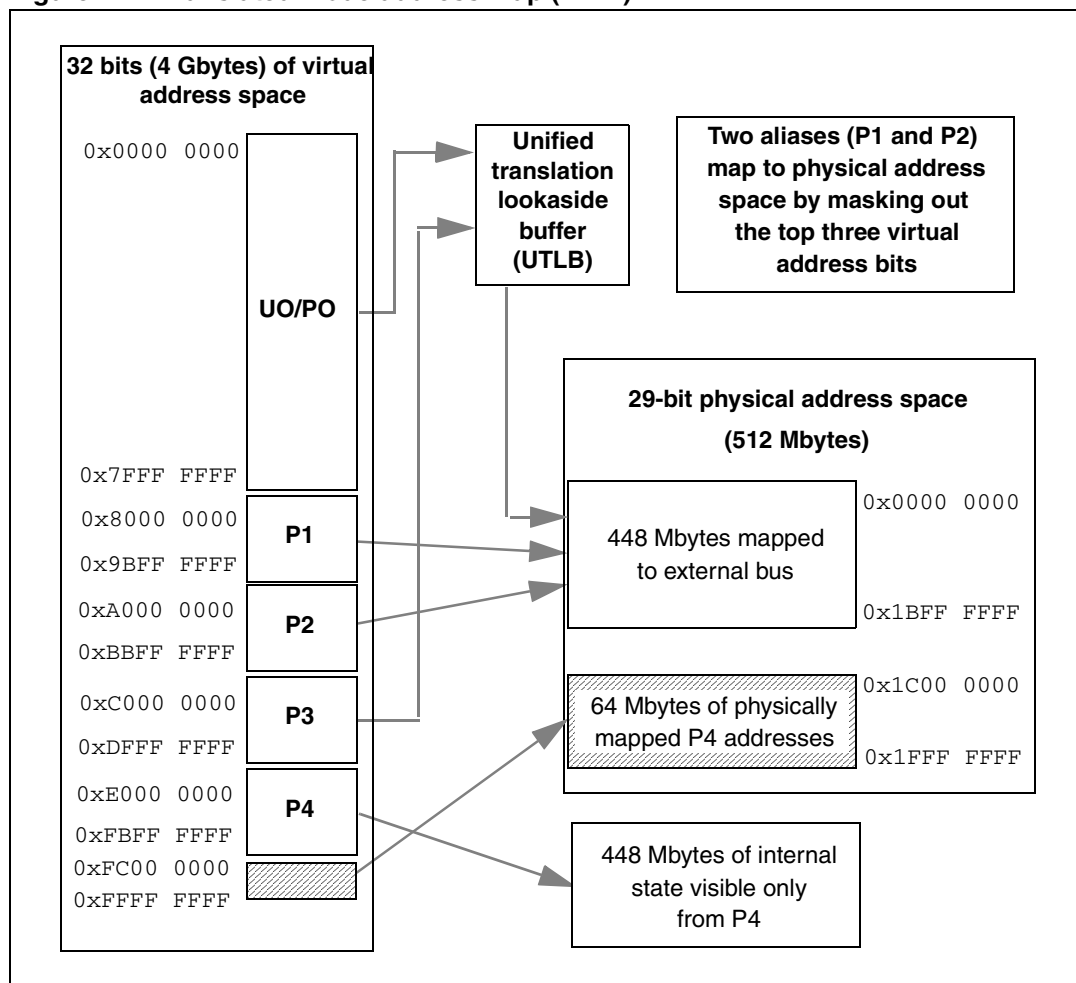


Figure 15. Space enhanced virtual mode address map

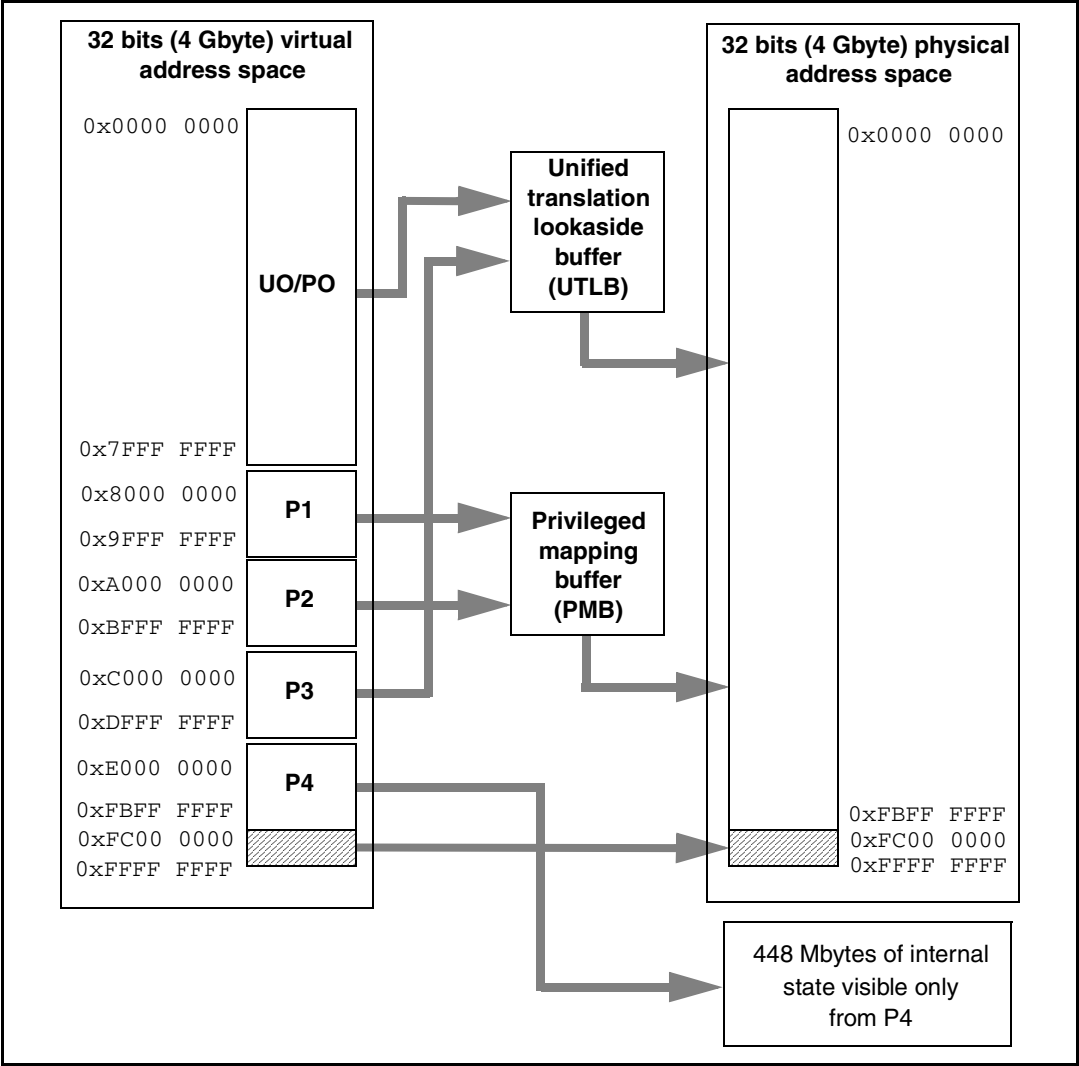
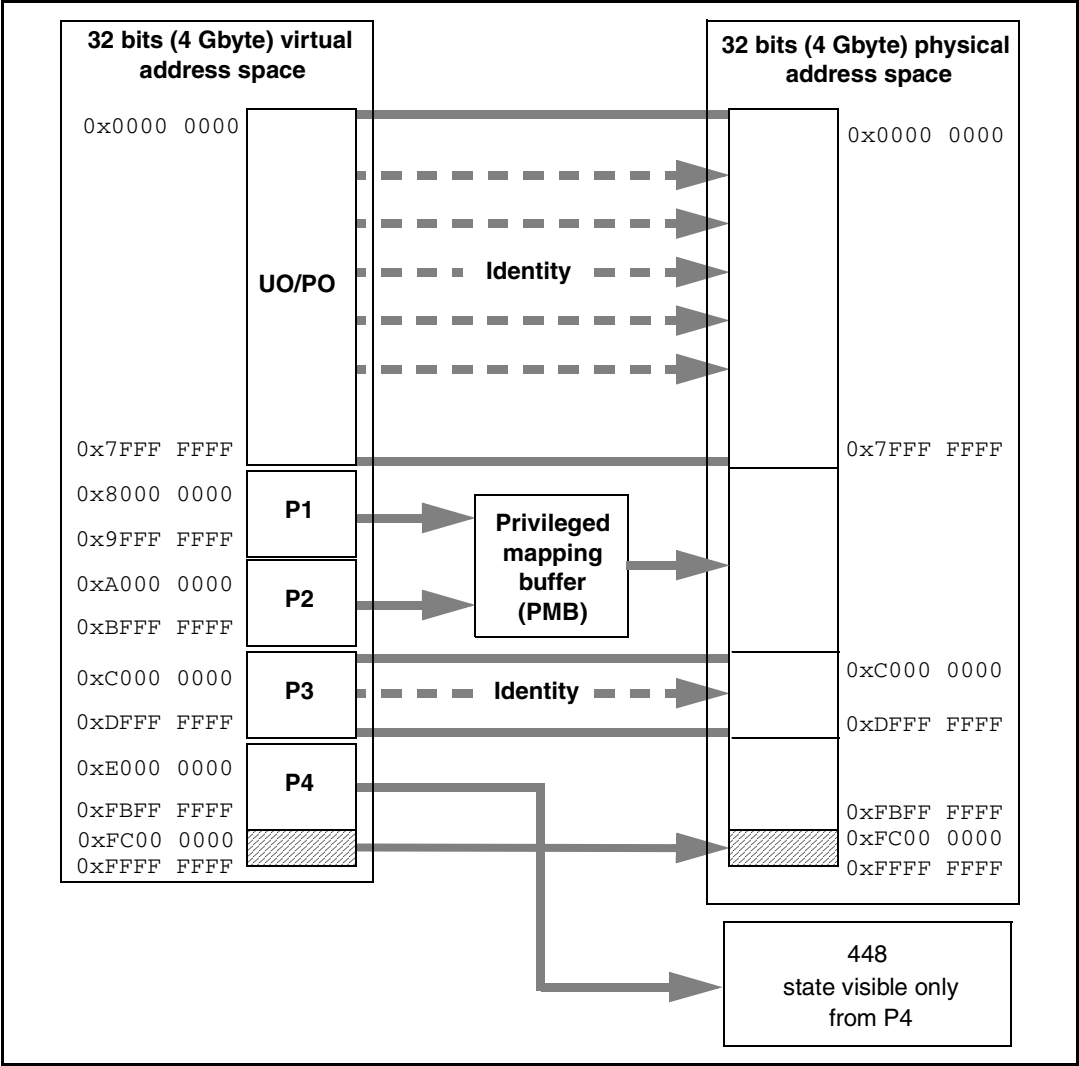


Figure 16. Space enhanced real mode address map



4 Caches

This chapter provides a description of the ST40 operand and instruction caches.

4.1 Overview

This section provides an overview of the material in this chapter.

4.1.1 Features

The ST40 has separate operand and instruction caches. The operand cache is abbreviated to OC and the instruction cache to IC. The caches vary between ST40 variants in terms of size (8 Kbyte, 16 Kbyte, 32 Kbyte and 64 Kbyte), associativity (direct-mapped, 2-way and 4-way) and modality (for example, compatibility and index). In all configurations the line size is 32 bytes and the cache line replacement policy, where relevant, is least recently used (LRU). Configuration information for a particular product can be found in its datasheet. A summary of the configurations for various products is given in [Table 52](#).

Table 52. Cache configuration and modes for ST40 core variants

| Core variant | Associativity (number of ways) | | | Instruction cache (IC) size (Kbytes) | Operand cache (OC) size (Kbytes) | Supported modes | | |
|---|--------------------------------|---|------------------|--------------------------------------|----------------------------------|-----------------|-------|-----|
| | 1 ⁽¹⁾ | 2 | 4 ⁽²⁾ | | | Compati-bility | Index | RAM |
| ST40-101 ST40-103 | ✓ | | | 8 | 16 | | ✓ | ✓ |
| ST40-202 | ✓ | ✓ | | 16 | 32 | ✓ | ✓ | ✓ |
| ST40-400 | | ✓ | | 16 | 32 | | ✓ | ✓ |
| ST40-200/400/500 series in general ⁽³⁾ | | ✓ | | 4, 8, 16, 32, 64 | 4, 8, 16, 32, 64 | | | ✓ |
| ST40-300 | | ✓ | ✓ | 4, 8, 16, 32, 64, 128 | 4, 8, 16, 32, 64, 128 | | | |

1. Direct-mapped

2. Optional extension for ST40-300; not present on any current implementation.

3. The ST40 cache sizes can be customized for each product. Refer to product-specific documentation for the cache sizes.

4.1.2 Cache modes

The following sections describe the cache modes. Some ST40 variants only support some of the modes.

Direct-mapped mode (compatibility mode)

This mode places the cache in a mode in which it behaves in the same way as an ST40-103 cache, that is, 8 Kbyte of direct-mapped instruction cache and 16 Kbyte of direct-mapped operand cache. Compatibility cannot be controlled independently for the operand and instruction caches. Currently, the only product that supports this mode is the ST40-202. When supported, this is the default mode at reset. However, there is a signal sampled at

reset that can disable compatibility mode. If this signal is active, 2-way set associative mode (enhanced mode) will be selected at reset.

Index mode

When index mode is active, bit[25] of the virtual address is used as the most significant bit of the set selector, instead of the bit that would usually have this function. By placing data/code segments on suitably aligned boundaries (32 + 64n Mbytes apart), it is possible to ensure they never conflict with each other in the cache.

Index mode can be controlled independently for the operand and instruction caches.

RAM mode

In RAM mode, half of the operand cache to be reassigned as a local operand memory. This is achieved by reassigning half the cache sets. The associativity of the cache is never affected by whether RAM mode is enabled or not. RAM mode is described in detail in [Section 4.3.6: RAM mode on page 125](#).

The ST40-100 series cores fully supports the use of RAM mode in conjunction with OC index mode.

On any other ST40 variant, RAM mode and OC index mode can not be used together, even in compatibility mode. An attempt to enable both modes together results in undefined behavior.

Write-through mode and copy-back mode

The ST40 supports two modes for handling stores to cacheable addresses: write-through mode and copy-back mode.

- **Write-through mode**

A store always updates external memory. The cache entry is updated too if there is a cache hit. It is always safe to invalidate cache entries operating in this mode because the memory is guaranteed to hold an up-to-date copy of the most recent store.

- **Copy-back mode**

A store is not immediately propagated to external memory. The data is only stored in the cache, and the cache line is marked *dirty*. At some time in the future, the whole cache line will be written out to external memory. It is not safe to invalidate dirty cache lines, because this will usually corrupt the memory state as seen by the running software.

When the store is to a virtual address for which translation is enabled, the selection of these modes is made through page-level information in the memory management unit (MMU); see [Table 13 on page 49](#).

When the virtual address is not subject to translation, the mode is selected based on the virtual address region by means of the CCR.WT and CCR.CB bits (see [Section 4.2.1: Cache control register \(CCR\)](#)).

4.1.3 Store queues

The ST40 architecture supports two 32-byte store queues (SQ) to perform high-speed burst writes to external memory without polluting the cache. The store queues are described in detail in [Section 4.6: Store queues \(SQs\) on page 140](#).

4.2 Register descriptions

There are four registers relating to the cache and store queues. These registers are described in [Table 53](#).

Table 53. Cache control registers

| Purpose | Name | R/W | Initial value ⁽¹⁾ | P4 address ⁽²⁾ | Access size | Restriction |
|----------------------------------|-------|-----|------------------------------|---------------------------|-------------|------------------------|
| Cache control register | CCR | R/W | 0x0000 0000 | 0xFF00 001C | 32 | |
| On-chip memory control register | RAMCR | R/O | 0x0000 00C0 | 0xFF00 0074 | 32 | Only on ST40-300 cores |
| Queue address control register 0 | QACR0 | R/W | Undefined | 0xFF00 0038 | 32 | |
| Queue address control register 1 | QACR1 | R/W | Undefined | 0xFF00 003C | 32 | |

1. The initial value is the value after a power-on or manual reset.

2. This is the address when using the virtual/virtual address space P4 area.

4.2.1 Cache control register (CCR)

The CCR can be accessed by a long-word access at 0xFF00 001C in the P4 region. The CCR bits are used to modify the cache settings described below. Restrictions apply when the CCR is being accessed; these are described in [Section 4.7: Cache state coherency on page 145](#). The CCR register fields are described in [Table 54](#).

Table 54. CCR register description

| CCR | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| OCE | 0 | 1 | Operand cache (OC) enable | RW |
| | Operation | | Indicates whether or not the OC is to be used. When address translation is performed, the OC cannot be used unless the C bit in the page management information is also 1. 0: OC not used. 1: OC used. | |
| | Reset | | 0 | |
| WT | 1 | 1 | Write-through enable | RW |
| | Operation | | Indicates the cache write mode for virtual addresses in U0/P0 and P3 when MMUCR.AT=0. When MMUCR.AT=1, the value of the WT bit in the matching UTLB entry (see Section 3.9.3 on page 74) has priority. 0: Copy-back mode. 1: Write-through mode. | |
| | Reset | | 0 | |

Table 54. CCR register description (continued)

| CCR | | | | |
|-------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| CB | 2 | 1 | Copy-back bit | RW |
| | Operation | | <p>Indicates the P1 region cache write mode.</p> <p>For ST40-100 and -400 series cores, this value always applies</p> <p>For ST40-200 and -500 series cores, this value applies if MMUCR.SE=0. If MMUCR.SE=1, the write mode is determined by the WT bit of the matching PMB entry, see Section 3.11.3 on page 91.</p> <p>For ST40-300 series cores, this value applies if PASC.R.SE=0. If PASC.R.SE=1, the write mode is determined by the WT bit of the matching PMB entry, see Section 3.11.3.</p> <p>0: Write-through mode.</p> <p>1: Copy-back mode.</p> | |
| | Reset | | 0 | |
| OCI | 3 | 1 | Operand cache invalidation bit | RW |
| | Operation | | When 1 is written to this bit, the V and U bits of all OC entries are cleared to 0. This bit always returns 0 when read. | |
| | Reset | | 0 | |
| ORA | 5 | 1 | Operand cache RAM enable bit | RW |
| | Operation | | <p>This bit only applies on cores that support RAM mode.</p> <p>0: Normal mode (all of OC is used as cache).</p> <p>1: RAM mode (half of OC is used as cache, the other half is used as RAM. Refer to Section 4.3.6 on page 125).</p> <p>If the core does not support RAM mode, this bit is read-only and always reads as zero.</p> <p>If the core supports both RAM mode and space-enhancement mode, CCR.ORA is ignored (RAM mode is disabled) if space-enhancement mode is enabled.</p> | |
| | Reset | | 0 | |
| OIX | 7 | 1 | Operand cache index enable bit | RW |
| | Operation | | <p>This bit only applies on cores that support index mode.</p> <p>0: Normal OC entry selection.</p> <p>1: Virtual address bit [25] used for most significant bit of entry selector.</p> <p>Table 69 on page 132 describes the bits used to form the set selector, depending on the operating mode.</p> <p>On cores that do not support index mode, this bit is read-only and always reads as zero.</p> <p>It is not possible to have CCR.ORA and CCR.OIX both set to 1 at the same time on the ST40-202. Refer to Section 4.3.7 on page 127.</p> | |
| | Reset | | 0 | |

Table 54. CCR register description (continued)

| CCR | | | | |
|-------|-----------|------|--|-----------|
| Field | Bits | Size | Synopsis | Type |
| ICE | 8 | 1 | Instruction cache (IC) enable bit | RW |
| | Operation | | Indicates whether or not the IC is to be used. When address translation is to be performed, the IC cannot be used unless the C bit in the page management information is also 1. 0: IC not used. 1: IC used. | |
| | Reset | | 0 | |
| ICI | 11 | 1 | Instruction cache (IC) invalidation bit | RW |
| | Operation | | When 1 is written to this bit, the V bits of all IC entries are cleared to 0. This bit always returns 0 when read. | |
| | Reset | | 0 | |
| IIX | 15 | 1 | Instruction cache (IC) index enable bit | RW |
| | Operation | | This bit only applies on cores that support index mode. 0: Normal IC entry selection. Address bits [12:5] used for IC entry selection. 1: Virtual address bit[25] used for most significant bit of entry selector. Table 69 on page 132 describes the bits used to form the set selector, depending on the operating mode. On cores that do not support index mode, this bit is read-only and always reads as zero. | |
| | Reset | | 0 | |
| EMODE | 31 | 1 | Direct-mapped / 2-way control | RW/ RO |
| | Operation | | Indicates whether or not the IC and the OC are to be used in direct-mapped or 2-way set associative mode. 0: Direct-mapped mode (also known as <i>compatibility mode</i>) 1: 2-way set associative mode (also known as <i>enhanced mode</i>) For the ST40-100 series cores, this bit is read only and returns 0. For the ST40-200 series (except 202), 300, 400, and 500 -series cores, this bit is read-only and returns 1. For the ST40-202, a hardware signal is sampled at reset which determines whether this bit is read-write, or read-only fixed at 1. ST40-202 behavior in direct-mapped mode differs from ST40-100 series in the following ways: 1. OC index mode and RAM mode cannot be used together. 2. The address map in RAM mode is different. | |
| | Reset | | Depends on the ST40 variant (see above). On an ST40-202 when EMODE is modifiable, the reset value is 0. | |

Table 54. CCR register description (continued)

| CCR | | | | |
|---------------|---------------------------------------|-----------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Reserved bits | 4, 6, [10:9] [14:12] [30:16] | 23 | For maximum forward compatibility preserve values on write, otherwise write 0. Read is undefined. | |
| | Reset | Undefined | | |

4.2.2 Queue address control register 0 (QACR0)

QACR0 can be accessed by long-word-size access from 0xFF00 0038 in the P4 region, or by using an address translation as described in [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#). The function of QACR0 is described in [Section 4.6.6: Physical address for external transfer on page 143](#).

Table 55. QACR0 register description

| QACR0 | | | | |
|---------------|------------------|---|--|------|
| Field | Bits | Size | Synopsis | Type |
| Area | [4:2] | 3 | Queue address control register 0 | RW |
| | Operation | Specifies bits[28:26] of the physical address onto which store queue 0 (SQ0) is mapped when MMUCR.AT=0. | | |
| | Reset | Undefined | | |
| Area32 | [7:5] | 3 | Queue address control register 0, 32-bit extension | RW |
| | Operation | Specifies bits[31:29] of the physical address onto which store queue 0 (SQ0) is mapped when MMUCR.AT=0 and MMUCR.SE=1 (200-/500-series core) / PASC.R.SE=1 (300-series core). This field does not exist on ST40 cores that do not have space-enhancement mode. | | |
| | Reset | Undefined | | |
| Reserved bits | [1:0], [31:8] | 29 | | |
| | Reset | Undefined | | |

4.2.3 Queue address control register 1 (QACR1)

QACR1 can be accessed by a long-word access from 0xFF00 003C in the P4 region, or by using an address translation as described in [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#). The function of QACR1 is described in [Section 4.6.6: Physical address for external transfer on page 143](#).

Table 56. QACR1 register description

| QACR1 | | | | |
|---------------|---------------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Area | [4:2] | 3 | Queue address control register 1 | RW |
| | Operation | | Specifies bits[28:26] of the physical address onto which store queue 1(SQ1) is mapped when MMUCR.AT=0. | |
| | Reset | | Undefined | |
| Area32 | [7:5] | 3 | Queue address control register 1, 32-bit extension | RW |
| | Operation | | Specifies bits[31:29] of the physical address onto which store queue 1(SQ1) is mapped when MMUCR.AT=0 and MMUCR.SE=1 (200-/500-series core) / PASC.R.SE=1 (300-series core). This field does not exist on ST40 cores that do not have space-enhancement mode. | |
| | Reset | | Undefined | |
| Reserved bits | [1:0], [31:8] | 29 | | |
| | Reset | | Undefined | |

4.2.4 On-chip memory control register (RAMCR)

RAMCR can be accessed by a long-word access from 0xFF00 0074 in the P4 region. Restrictions apply when the RAMCR is being accessed; these are described in [Section 4.7: Cache state coherency on page 145](#).

The RAMCR exists only on the ST40-300 core. Access to its address on other ST40 variants is undefined. The fields of the RAMCR are described in [Table 57](#).

Table 57. RAMCR register description

| RAMCR | | | | |
|-------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| OC2W | 6 | 1 | Operand cache (OC) 2-way / 4-way selection | RO |
| | Operation | | This bit is only defined for the ST40-300 core. It allows for a later extension to 4-way caches. 0: Operand cache is 4-way set associative 1: Operand cache is 2-way set associative | |
| | Reset | | 1 | |

Table 57. RAMCR register description (continued)

| RAMCR | | | | |
|---------------|---------------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| IC2W | 7 | 1 | Instruction cache (IC) 2-way / 4-way selection | RO |
| | Operation | | This bit is only defined for the ST40-300 core. It allows for a later extension to 4-way caches. 0: Instruction cache is 4-way set associative 1: Instruction cache is 2-way set associative | |
| | Reset | | 1 | |
| Reserved bits | [5:0], [31:8] | 30 | For maximum forward compatibility preserve values on write, otherwise write 0. Read is undefined. | |
| | Reset | | Undefined | |

4.3 Operand cache (OC)

Each OC line consists of an address tag, a V bit, a U bit, and 32-bytes of data. Lines are grouped in to sets, the number of lines in each set is the associativity of the cache. On products with a 2-way set associative cache, there is an additional bit per cache set to record which of the two lines was least recently used.

4.3.1 Configuration

Set selection during a cache accesses is performed by taking bits from the virtual address. The bits of the virtual address used depend upon the number of bytes in each cache way and whether the cache is operating in index and/or RAM mode. The construction of the set selector in various conditions/configurations is summarized in [Table 58](#).

Table 58. Operand cache - virtual address bits used for set selection in various conditions

| Way size ⁽¹⁾ | Index mode | RAM mode | Set selector |
|-------------------------|------------|----------|--------------|
| 2 Kbyte | No | No | [10:5] |
| | | Yes | [9:5] |
| 4 Kbyte | No | No | [11:5] |
| | No | Yes | [10:5] |
| 8 Kbyte | No | No | [12:5] |
| | No | Yes | [11:5] |
| 16 Kbyte | No | No | [13:5] |
| | Yes | No | [25,12:5] |
| | No | Yes | [12:5] |
| | Yes | Yes | [25,11:5] |

Table 58. Operand cache - virtual address bits used for set selection in various conditions (continued)

| Way size ⁽¹⁾ | Index mode | RAM mode | Set selector |
|-------------------------|------------|----------|--------------|
| 32 Kbyte | No | No | [14:5] |
| | No | Yes | [13:5] |
| 64 Kbyte | No | No | [15:5] |

1. Way size = (size of cache) / associativity

Tag

The tag stores the upper bits of the physical address of the data currently being stored in the cache line. During a lookup, the tag is compared against the physical address returned by the MMU to determine whether a hit has occurred. The tag is not initialized by a power-on or manual reset.

The size of the tag is determined by the smallest page size supported by the MMU. As the smallest page size is 1 Kbyte, the lowest address bit in the tag is bit 10. The highest active bit in the tag depends on whether space-enhancement mode ([Section 3.4.2: 32-bit physical address space \(space-enhancement mode\) on page 38](#)) is active. When space-enhancement mode is off, bit 28 is the highest active bit. When space-enhancement mode is on, bit 31 is the highest active bit.

V bit (validity bit)

Setting this bit to 1 indicates that valid data is stored in the cache line. The V bit is initialized to 0 by a power-on reset, but retains its value in a manual reset.

U bit (dirty bit)

The U bit is set to 1 if data is written to the cache line while that line is being used in copy-back mode. That is, the U bit indicates a mismatch between the data in the cache line and the data in external memory.

The U bit is never set to 1 by the hardware while the cache line is being used in write-through mode. The only way this can occur is by explicit software action through a modification to the memory-mapped cache, see [Section 4.5: Memory-mapped cache configuration on page 131](#).

The U bit is initialized to 0 by a power-on reset, but retains its value in a manual reset.

Data field

The data field holds 32 bytes (256 bits) of data per cache line. The data array is not reinitialized by a power-on or manual reset.

LRU (only in products with 2-way support)

For 2-way set-associative caches, an additional state bit is implemented to keep track of which of the two ways in each cache set was least recently used (LRU). These additional LRU bits can not be read or written by software.

The LRU bits provide guidance to the implementation regarding which line in a set to select for replacement following a cache miss. The architecture does not define how the implementation should perform this selection.

4.3.2 Read operation

When the OC is enabled (CCR.OCE = 1) and data is read by means of a virtual address from a cacheable area, the cache operates as follows:

1. The tag, V bit, and U bit are read from the cache lines indexed by the set selection bits (see [Table 58](#)) in the virtual address.
2. The tag is compared with the upper bits the physical address resulting from virtual address translation by the MMU.

When space-enhancement mode is off (SE=0), bits [28:10] of the tag and physical address are compared.

When space-enhancement mode is on (SE=1), bits [31:10] of the tag and physical address are compared.

Operation is as described in [Table 59](#).

Table 59. OC read operation

| Tag match | V bit | U bit | Operation | Description |
|-----------|-------|-------|------------------------------|---|
| Yes | 1 | - | Cache hit | The data indexed by bits [4:0] of the virtual address, is read from the cache line in accordance with the access size (quad-word/long-word/word/byte). |
| Yes | 0 | - | Cache miss (no write-back) | Data from the external memory corresponding to the virtual address, is written into the cache line. It is read critical word first and the data passed to the CPU as soon as it arrives in the cache. The CPU continues to execute subsequent instructions while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the virtual address is recorded in the cache, and the V bit is set to 1. |
| No | 0 | - | | |
| No | 1 | 0 | Cache miss (with write-back) | The least recently used line in the set is selected for eviction and its tag and data fields saved in the write-back buffer. Then, data from the external memory space corresponding to the virtual address, is written into the cache line. Data is read using the critical word first method, and when the data arrives in the cache, the read data is returned to the CPU. The CPU continues to execute subsequent instructions while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the virtual address is recorded in the cache, the V bit is set to 1, and the U bit is set to 0. The data in the write-back buffer is then written back to the external memory. |

4.3.3 Write operation

When the OC is enabled (CCR.OCE = 1) and data is written by means of a virtual address to a cacheable area, the cache operates as follows:

1. The tag, V bit, and U bit are read from the cache line indexed by the set selection bits in the virtual address, see [Table 58 on page 121](#).
2. The tag is compared with the upper bits of the physical address resulting from the translation of the virtual address performed by the MMU.

When space-enhancement mode is off (SE=0), bits [28:10] of the tag and physical address are compared.

When space-enhancement mode is on (SE=1), bits [31:10] of the tag and physical address are compared.

In copy back, operation is described in [Table 60](#). In write through mode it is described in [Table 61](#).

Table 60. OC write operation, with copy-back

| Tag match | V bit | U bit | Operation | Description |
|-----------|-------|-------|--|---|
| Yes | 1 | - | Cache hit (copy-back) | A data write for the data indexed by bits [4:0] is performed, in accordance with the access size (quad-word/long-word/word/byte). The U bit is set to 1. |
| Yes | 0 | - | Cache miss (no copy-back/write-back) | A data write for the data indexed by bits [4:0] is performed, in accordance with the access size (quad-word/long-word/word/byte). Then, data from the external memory corresponding to the virtual address, is read into the cache line excluding the written data. The CPU continues to execute subsequent instructions while the cache line of data is being read. Once reading of one line of data is completed, the tag corresponding to the virtual address is recorded in the cache, the V bit and U bit are both set to 1. |
| No | 0 | - | | |
| No | 1 | 0 | Cache miss (no copy-back/write-back) | The least recently used line in the set is selected for eviction and its tag and data fields saved in the write-back buffer. Then a data write for the data indexed by bits [4:0] is performed in accordance with the access size (quad-word/long-word/word/byte). Data from the external memory space corresponding to the virtual address is read into the cache line, excluding the written data. The CPU continues to execute subsequent instructions while the cache line of data is being read. When reading of one line of data is completed, the tag corresponding to the virtual address is recorded in the cache, the V bit and U bit are both set to 1. The data in the write back buffer is then written back to external memory. |
| No | 1 | 1 | Cache miss (with copy-back/write-back) | |

Table 61. OC write operation, with write-through

| Tag match | V bit | U bit | Operation | Description |
|-----------|-------|-------|----------------------------|---|
| Yes | 1 | - | Cache-hit (write-through) | A data write for the data indexed by bits [4:0], is performed in accordance with the access size (quad-word/long-word/word/byte). |
| Yes | 0 | - | Cache miss (write-through) | A write is performed to the external memory, corresponding to the virtual address. A write to cache is not performed. |
| No | 0 | - | | |
| No | 1 | 0 | | |
| No | 1 | 1 | | |

4.3.4 Write-back buffer

The write-back buffer enables priority to be given to data reads, and improves performance. When a cache miss makes the purge of a dirty cache entry into external memory necessary, the cache entry is held in the write-back buffer. The write-back buffer contains one cache line of data and the virtual address of the purge destination.

4.3.5 Write-through buffer

When writing data in write-through mode or writing to a non-cacheable area, data is held in a 64-bit buffer. This allows the CPU to proceed to the next operation as soon as the write to the write-through buffer is completed, without waiting for completion of the write to external memory.

4.3.6 RAM mode

Note: RAM mode is only available on some ST40 core variants. See [Table 52 on page 114](#).

There are a number of differences in the implementation of RAM-mode between direct-mapped and 2-way operand caches. Each is therefore documented separately in the following sections.

The address range defined as RAM can be accessed using byte, word, long-word, and quad-word size data reads and writes. Instruction fetches **cannot** be performed in this area and give architecturally undefined behavior.

RAM mode is inaccessible if space-enhancement mode ([Section 3.4.2: 32-bit physical address space \(space-enhancement mode\) on page 38](#)) is enabled.

RAM mode in 2-way operand cache operation

Note: On ST40s with 2-way operand caches RAM mode cannot be used in conjunction with OC index mode, even when in direct-mapped (compatibility) mode.

RAM-mode is controlled by the CCR.ORA bit. At reset this bit is clear (RAM mode off). Writing a one to the bit enables RAM-mode. RAM mode may be turned off again by writing a zero to CCR.ORA.

In RAM mode, the first half of the entries in each way combine to implement a 2-way set associative cache. The second half of the entries in each way combine to implement a RAM. The entries taken from way 0 form RAM1, which has a base address at 0x7C00 0000 and is a quarter of the size of the entire cache. This is followed in the address map by RAM2, formed by taking the entries from way 1, to provide, in total, a contiguous region of memory

half the size of the cache. The address map then wraps around back on to RAM1 and does so every half-cache size up to and including the address 0x7FFF FFFF. [Table 62](#) summarizes this for each standard cache configuration.

The cache entries that are not used for the RAM area continue to operate as a regular cache.

Table 62. Standard cache RAM-mode implementations

| Size of entire cache | Cache and RAM size in RAM mode | Cache sets used to construct RAM | Base address of RAM1 (from way 0) | Base address of RAM2 (from way 1) |
|----------------------|--------------------------------|----------------------------------|-----------------------------------|-----------------------------------|
| 4 Kbytes | 2 Kbytes | 32 to 63 | 0x7C00 0000 | 0x7C00 0400 |
| 8 Kbytes | 4 Kbytes | 64 to 127 | 0x7C00 0000 | 0x7C00 0800 |
| 16 Kbytes | 8 Kbytes | 128 to 255 | 0x7C00 0000 | 0x7C00 1000 |
| 32 Kbytes | 16 Kbytes | 256 to 511 | 0x7C00 0000 | 0x7C00 2000 |
| 64 Kbytes | 32 Kbytes | 512 to 1023 | 0x7C00 0000 | 0x7C00 4000 |

When direct-mapped (compatibility) mode is used in conjunction with RAM mode, only RAM1 is mapped in to memory (with a base address of 0x7C00 0000) and wraps around every quarter cache size (for example, every 8 Kbytes on an ST40-202).

Direct-mapped operand cache (ST40-100 series)

Currently all ST40 direct-mapped operand caches are 16Kbytes in size. This section therefore assumes this in its description of the RAM mode implementation in direct-mapped caches.

Setting CCR.ORA to 1 enables half of the operand cache to be used as RAM. The operand cache entries (sets) used as RAM are entries 128 to 255 and 384 to 511. Other entries continue to be used as cache.

An example of RAM use is shown in [Table 63](#) and [Table 64](#). Here, the 4 Kbytes comprising OC entries 128 to 255 are designated as RAM area 1, and the 4 Kbytes comprising OC entries 384 to 511 as RAM area 2.

Table 63. RAM use when OC index mode is off (CCR.OIX = 0)

| Address start | Address end | Size | RAM area |
|---------------|-------------|----------|------------------|
| 0x7C00 0000 | 0x7C00 0FFF | 4 Kbytes | 1 |
| 0x7C00 1000 | 0x7C00 1FFF | | 1 |
| 0x7C00 2000 | 0x7C00 2FFF | | 2 |
| 0x7C00 3000 | 0x7C00 3FFF | | 2 |
| 0x7C00 4000 | 0x7C00 4FFF | | 1 ⁽¹⁾ |

1. RAM areas 1 and 2 then repeat every 8 Kbytes up to 0x7FFF FFFF.

To secure a continuous 8 Kbyte RAM area, the area from 0x7C00 1000 to 0x7C00 2FFF can be used, for example.

Table 64. RAM use when OC index mode is on (CCR.OIX = 1)

| Address start | Address end | Size | RAM area |
|---------------|-------------|----------|----------|
| 0x7C00 0000 | 0x7C00 0FFF | 4 Kbytes | 1 |
| 0x7C00 1000 | 0x7C00 1FFF | | 1 |
| 0x7C00 2000 | 0x7C00 2FFF | | 1 |
| ... | ... | | 1 |
| 0x7DFF F000 | 0x7DFF FFFF | | 1 |
| 0x7E00 0000 | 0x7E00 0FFF | | 2 |
| 0x7E00 1000 | 0x7E00 1FFF | 4 Kbytes | 2 |
| ... | ... | | 2 |
| 0x7FFF F000 | 0x7FFF FFFF | | 2 |

As the distinction between RAM areas 1 and 2 is indicated by address bit [25], the area from 0x7DFF F000 to 0x7E00 0FFF should be used to secure a continuous 8 Kbyte RAM area.

4.3.7 OC index mode

Note: OC index mode is only available on some ST40 core variants. See [Table 52 on page 114](#).

This mode is enabled by setting CCR.OIX.

It causes bit 25 of the virtual address to be used as the highest bit in the set selector. Some examples for various cache configurations can be found in [Table 58 on page 121](#). By placing data segments on suitably aligned boundaries (32+64n Mbytes apart), it is possible to ensure they never conflict with each other in the cache.

4.3.8 Explicit cache controls

In the ST40 CPU core, five instructions are provided for explicitly controlling the state of the operand cache from the instruction stream. Fuller details of these instructions are given in [Chapter 9: Instruction descriptions on page 213](#).

- **Cache invalidate (OCBI @Rn)**
If data at the specified virtual address is present in the operand cache then the line is invalidated and any dirty data in that line is lost. If the specified address is not present in the cache the instruction has no effect.
- **Cache purge (OCBP @Rn)**
If data at the specified virtual address is present in the operand cache then the line is flushed, if it contains dirty data, and the line invalidated. If the specified address is not present in the cache the instruction has no effect.
- **Cache flush (OCBWB @Rn)**
If data at the specified virtual address is present in the operand cache then the line is flushed if it contains dirty data, otherwise it has no effect.
- **Cache prefetch (PREF @Rn)**
The ST40 supports a prefetch instruction, to reduce the operand cache fill penalty incurred as the result of a cache miss. If it is known that a cache miss will result from a read or write operation, it can be prevented by using the prefetch instruction to fill the operand cache with data before the operation, and so improve software performance. If

a prefetch instruction is executed for data already held in the cache, or if the prefetch address results in a UTLB miss or a protection violation, the result is no operation, and an exception is not generated.

- Cache allocate + store (**MOVCA.L R0, @Rn**)

This instruction stores the long-word to memory at the specified virtual address. It provides a hint to the implementation that it is not necessary to retrieve the data of this operand cache block from memory.

4.4 Instruction cache (IC)

Each instruction cache line consists of an address tag, a V bit and 32-bytes of data. Lines are then grouped into sets, the number of lines in each set being the associativity of the cache. On products with a 2 way set associative cache there is an additional bit per cache set to record which of the two lines was least recently used.

4.4.1 Configuration

Set selection during a cache access is performed by taking bits from the virtual address of the instruction being fetched. The bits of the virtual address used depend upon the size of each cache way and whether the cache is operating in index mode. The construction of the set selector in various conditions/configurations is summarized in [Table 58](#).

Table 65. Instruction cache - virtual address bits used for set selection in various conditions

| Way size | Index mode | Set selector |
|----------|------------|--------------|
| 2 Kbyte | No | [10:5] |
| 4 Kbyte | No | [11:5] |
| 8 Kbyte | No | [12:5] |
| | Yes | [25,11:5] |
| 16 Kbyte | No | [13:5] |
| 32 Kbyte | No | [14:5] |
| 64 Kbyte | No | [15:5] |

Tag

The tag stores the upper bits of the physical address of the data currently being stored in the cache line. During a lookup, the tag is compared against the physical address returned by the MMU to determine whether a hit has occurred. The tag is not initialized by a power-on or manual reset.

The size of the tag is determined by the smallest page size supported by the MMU. As the smallest page size is 1kbyte, the lowest address bit in the tag is bit 10. The highest active bit in the tag depends on whether space-enhancement mode is active, see [Section 3.4.2: 32-bit physical address space \(space-enhancement mode\) on page 38](#). When space-enhancement mode is off, bit 28 is the highest active bit. When space-enhancement mode is on, bit 31 is the highest active bit.

V bit (validity bit)

Setting this bit to 1 indicates that valid data is stored in the cache line. The V bit is initialized to 0 by a power-on reset, but retains its value in a manual reset.

Data field

The data field holds 32 bytes (256 bits) of data per cache line. The data array is not reinitialized by a power-on or manual reset.

LRU (only in products with 2-way support)

For 2-way set-associative caches, an additional state bit is implemented to keep track of which of the two ways in each cache set was least recently used (LRU). These additional LRU bits can not be read or written by software.

The LRU bits provide guidance to the implementation regarding which line in a set to replace following a cache miss. The architecture does not define how the implementation should perform this selection.

4.4.2 Read operation

When the IC is enabled (CCR.ICE = 1) and instruction fetches are made to a virtual address in a cacheable area, the instruction cache operates as follows:

1. The tag and V bit are read from the cache line indexed by the set selection bits in the virtual address, see [Table 65](#).
2. The tag is compared with the upper bits the physical address resulting from virtual address translation by the MMU.

When space-enhancement mode is off (SE=0), bits [28:10] of the tag and physical address are compared.

When space-enhancement mode is on (SE=1), bits [31:10] of the tag and physical address are compared.

Operation is as described in [Table 66](#).

Table 66. IC read operation

| Tag | V bit | Operation | Description |
|----------------|-------|------------|--|
| Matches | 1 | Cache hit | Data indexed by virtual address bits [4:2] is read as an instruction. |
| Matches | 0 | Cache miss | Data is read, critical word first, into the cache line from the external memory space corresponding to the virtual address. Once the data arrives in the cache, it is passed back to the CPU. When reading of one line of data is completed, the tag corresponding to the virtual address is recorded in the cache, and 1 is written to the V bit. |
| Does not match | 0 | | |
| Does not match | 1 | | |

4.4.3 IC index mode

Note: IC index mode is only available on certain ST40 core variants. See [Table 52 on page 114](#).

This mode is enabled by setting CCR.IIX.

It causes bit 25 of the virtual address to be used as the highest bit in the set selector. Some examples for various cache configurations can be found in [Table 65 on page 128](#). This partitioning makes it possible for software to make more efficient use of the cache.

4.4.4 Explicit cache controls

- Instruction cache invalidate (**ICBI @Rn**)

Note: This instruction is only present on the ST40-300 core.

Regardless of the effect on the cache, the **ICBI** instruction always causes the following instructions to be refetched to ensure full instruction coherency.

ICBI has two functions.

- If the virtual address specified by **Rn** is present in the instruction cache, the line containing the address is invalidated. If the specified address is not present in the instruction cache, the instruction has no effect on the cache. Likewise, if an uncacheable virtual address is specified, there is no effect on the cache.

If the address in **Rn** is subject to translation in the current MMU mode, it is the result of the translation (in the ITLB, including refill from the UTLB/PMB as required) that is used for the cache tag comparison.

- The pipeline is flushed and the instruction following the **ICBI** is re-fetched.

The refetch behavior happens whether or not a change to the cache state occurred. To ensure a refetch of the following instruction for coherency purposes (for example, after a modification to the MMU state), any suitable address may be specified for **Rn**. For example, an even P4 address will never affect the cache state but will achieve a refetch in privileged mode.

ICBI may be used in both user mode and privileged mode.

ICBI may not be used in a branch delay slot.

The exception order for **ICBI @Rn** at PC = A is shown in [Table 67](#).

Table 67. Exception priorities for the ICBI instruction

| Priority | Exception type | EXPEVT | TEA | Situation |
|----------|----------------|--------|-----|--|
| Highest | ITLBMULTIHIT | 0x140 | A | If multiple TLB entries match when looking up the PC address 'A' |
| | IADDERR | 0x0E0 | A | If an address error is associated with the PC address 'A' |
| | ITLBMISS | 0x040 | A | If no TLB entry matches when looking up the PC address 'A' |
| | EXECPROT | 0x0A0 | A | If the page protection disallows access to the PC address 'A' |
| | ILLSLOT | 0x1A0 | - | If the ICBI occurs in a delay slot |
| | ITLBMULTIHIT | 0x140 | Rn | If multiple TLB entries match when looking up the address in Rn |
| | IADDERR | 0x0E0 | Rn | If an address error is associated with the address in Rn |
| | ITLBMISS | 0x040 | Rn | If no TLB entry matches when looking up the address in Rn |
| Lowest | EXECPROT | 0x0A0 | Rn | If the page protection disallows access to the address Rn |

4.5 Memory-mapped cache configuration

To allow the IC and OC contents to be controlled by software, the ST40 architecture allows direct access to the data and address arrays of the caches.

There are restrictions on the address region containing the current PC when the IC and OC contents are accessed by software. These are documented in [Section 4.7: Cache state coherency on page 145](#).

The IC and OC data and address arrays are mapped to the P4 region of the virtual address space. Only long word data accesses can be used to access all address and data arrays. Instruction fetches cannot be performed in these address regions and give undefined behavior.

Which array is selected depends upon the top 8 bits of the address (see [Table 68](#)).

Table 68. Virtual address map of cache arrays

| Virtual address bits [31:24] | Cache array |
|------------------------------|------------------|
| 0xF0 | IC address array |
| 0xF1 | IC data array |
| 0xF4 | OC address array |
| 0xF5 | OC data array |

When any reserved bits referred to in this section are accessed they should be assumed undefined when read. When written, a previously read value should be written back if available, otherwise 0 should be used.

For the purposes of this specification the term *entry* is used to refer to all the state associated with a particular cache line, the term *set* refers to a group of entries associated with the same address selector, and the term *way* identifies a particular entry in a set.

A particular entry in the cache array is selected using bits in the address by which it is accessed. The address bits used in each of the supported cache configurations is in [Table 69](#).

Table 69. Bits in address used to index in to memory-mapped cache arrays

| Cache size | Associativity | RAM mode off | | RAM mode on | | Examples of use |
|------------|---------------|--------------------------------|---------------------------------|--------------------------------|---------------------------------|---|
| | | Address bit used to select way | Address bits used to select set | Address bit used to select way | Address bits used to select set | |
| 8 Kbytes | 1 | - | [12:5] | - | [12:5] | ST40-103 IC or ST40-202 in compatibility mode |
| 16 Kbytes | 1 | - | [13:5] | - | [13:5] | ST40-103 OC or ST40-202 in compatibility mode |
| 4 Kbytes | 2 | 11 | [10:5] | 10 | [9:5] | |
| 8 Kbytes | 2 | 12 | [11:5] | 11 | [10:5] | |
| 16 Kbytes | 2 | 13 | [12:5] | 12 | [11:5] | ST40-202 IC |
| 32 Kbytes | 2 | 14 | [13:5] | 13 | [12:5] | ST40-202 OC |
| 64 Kbytes | 2 | 15 | [14:5] | 14 | [13:5] | |

On all products other than the 100-series, RAM mode alters the indexing so that the cache arrays are accessed as if the cache is half the size it actually is. For example, an ST40-202 in RAM mode uses bit 13 as its way selector and bits [12:5] as its set selector. However, on a 100-series product (or the 202 operating in compatibility mode) the indexing is unaffected and it is possible to modify the contents of the RAM by writes in to the RAM-assigned entries in the operand cache data array. For reasons of upward compatibility this facility should never be exploited.

On all products the use of index mode has no affect on the bits of the address used to index in to the memory-mapped cache arrays.

All unspecified address field bits must be zero when accessing memory-mapped cache arrays. The result of performing a read with one of these bits set is that it will return an undefined value. The results of performing a write with one of these address bits set is that the state of the cache might be modified in an unspecified way.

4.5.1 IC address array

The IC address array is allocated to virtual addresses 0xF000 0000 to 0xF0FF FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (**MOV.L**) must be used when accessing the array. Loads are used to examine the contents of the IC. Stores are used to update the contents.

Access to the IC address array must adhere to the restrictions in [Section 4.7: Cache state coherency on page 145](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

The address is formed as shown in [Table 70](#). N is determined by the cache size. Refer to [Table 69](#) for the bits used for SET and WAY in each cache configuration.

Note: The setting of CCR.IIX has no effect on the entry selection for a memory-mapped access.

Table 70. Addressing for the IC address array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|---|
| [31:24] | 8 | 0xF0 | Selects the IC address array within the P4 address region |
| [23:N+1] | 23-N | zero | Unused bits. These should be zero for future compatibility |
| [N] | 1 | WAY | When the cache is 2-way set associative, this bit defines the way to act on. |
| [N-1:5] | N-5 | SET | These bits define the set in the IC that is to be acted upon. |
| [4], [2] | 1,1 | zero | Unused bits. These should be zero for future compatibility |
| [3] | 1 | ASSOC | Specifies whether addressed or associative operation is required 0: addressed (non-associative) operation 1: associative operation Associative operation is only meaningful for writes. This bit is ignored for reads. |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 71](#).

Table 71. Fields in each IC address array entry

| IC address array entry | | | | |
|------------------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| V | 0 | 1 | Valid | RW |
| | Operation | | Indicates whether the cache entry is valid. 0: invalid 1: valid | |
| | Reset | | Undefined | |
| Reserved | [9:1] | 8 | Reserved | RW |
| | Operation | | Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| TAG | [31:10] | 22 | Physical cache tag | RW |
| | Operation | | Indicates the tag for the cache entry. During an associative write, this represents the virtual address that will be translated to yield the physical address for comparison with all the tags in the indexed set. | |
| | Reset | | Undefined | |

Three kinds of operation can be used on the IC address array. These are IC address array read, IC address array write (non-associative) and IC address array write (associative).

IC address array read

The TAG and V bit are read from the IC entry corresponding to the SET and WAY fields of the address. Regardless of the setting of the A-bit, array reads are never performed associatively.

IC address array write (non-associative)

The ASSOC bit in the address must be 0 to select this operation.

The TAG and V bit specified in the stored value are written to the IC entry corresponding to the SET and WAY fields of the address.

IC address array write (associative)

The ASSOC bit in the address must be 1 to select this operation.

The TAG field of the stored value is examined. It is treated as bits [31:10] of a virtual address, with bits [9:0] treated as zero. This virtual address is translated to a physical address by the MMU as though it were a regular instruction access, see [Table 11 on page 47](#). The upper bits resulting from the mapping are compared against the TAG field in all cache entries in the set defined by the SET field of the address. The WAY field of the address is ignored. If a matching entry is found with its V bit set to 1, the V bit from the stored value is written to the V bit of the matching cache entry. If no match is found, no further action occurs.

If the MMU lookup would have caused an ITLBMISS exception, this is ignored and no further action is taken.

If the MMU lookup detects a multiple hit in the ITLB, an ITLBMULTHIT exception is launched.

4.5.2 IC data array

The IC data array is allocated to virtual addresses 0xF100 0000 to 0xF1FF FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (**MOV.L**) must be used when accessing the array. Loads are used to examine the contents of the IC. Stores are used to update the contents.

Access to the IC data array must adhere to the restrictions in [Section 4.7: Cache state coherency on page 145](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

The address is formed as shown in [Table 72](#). N is determined by the cache size. Refer to [Table 69 on page 132](#) for the bits used for SET and WAY in each cache configuration.

Note: The setting of CCR.IIX has no effect on the entry selection for a memory-mapped access.

Table 72. Addressing for the IC data array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|--|
| [31:24] | 8 | 0xF1 | Selects the IC data array within the P4 address region |
| [23:N+1] | 23-N | zero | Unused bits. These should be zero for future compatibility |
| [N] | 1 | WAY | When the cache is 2-way set associative, this bit defines the way to act on. |
| [N-1:5] | N-5 | SET | These bits define the set in the IC that is to be acted upon. |
| [4:2] | 3 | WORD | Define the longword to access within the cache entry. |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data operand of the load or store contains the selected long word data element of the cache line.

Two kinds of operation can be used on the IC data array. These are IC data array read and IC data array write.

IC data array read

The cache long word selected by the WAY, SET and WORD of the address is returned.

IC data array write

The long word supplied in the stored value is written to the cache long word selected by the WAY, SET and WORD fields of the address.

4.5.3 OC address array

The OC address array is allocated to virtual addresses 0xF400 0000 to 0xF4FF FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (**MOV.L**) must be used when accessing the array. Loads are used to examine the contents of the OC. Stores are used to update the contents.

Access to the OC address array must adhere to the restrictions in [Section 4.7: Cache state coherency on page 145](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

The address is formed as shown in [Table 73](#). N is determined by the cache size. Refer to [Table 69 on page 132](#) for the bits used for SET and WAY in each cache configuration. Note that the setting of CCR.OIX has no effect on the entry selection for a memory-mapped access.

However, CCR.ORA may affect the indexing. For a ST40-100 series part, or a ST40-202 operating in direct-mapped (compatibility) mode, the cache entries used as RAM may still be accessed through the memory-mapped address and data arrays. However for other variants, the indexing behaves as though the cache were half its natural size when CCR.ORA=1 for indexing. [Table 69 on page 132](#) defines the indexing and way selection bits based on the cache configuration.

Table 73. Addressing for the OC address array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|---|
| [31:24] | 8 | 0xF4 | Selects the OC address array within the P4 address region |
| [23:N+1] | 23-N | zero | Unused bits. These should be zero for future compatibility |
| [N] | 1 | WAY | When the cache is 2-way set associative, this bit defines the way to act on. |
| [N-1:5] | N-5 | SET | These bits define the set in the OC that is to be acted upon. |
| [4], [2] | 1,1 | zero | Unused bits. These should be zero for future compatibility |
| [3] | 1 | ASSOC | Specifies whether addressed or associative operation is required 0: addressed (non-associative) operation 1: associative operation Associative operation is only meaningful for writes. This bit is ignored for reads. |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data fields in each address array entry are shown in [Table 74](#).

Table 74. Fields in each OC address array entry

| OC address array entry | | | | |
|------------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| V | 0 | 1 | Valid | RW |
| | Operation | | Indicates whether the cache entry is valid. 0: invalid 1: valid | |
| | Reset | | Undefined | |
| U | 1 | 1 | Used | RW |
| | Operation | | Indicates whether the cache entry is dirty. 0: clean 1: dirty If the line is accessed in write-through mode, the U bit should always be zero. A line with U=V=1 will be written back to memory when it is evicted. | |
| | Reset | | Undefined | |
| Reserved | [9:2] | 7 | Reserved | RW |
| | Operation | | Writes are ignored and reads return zero. | |
| | Reset | | Undefined | |
| TAG | [31:10] | 22 | Physical cache tag | RW |
| | Operation | | Indicates the tag for the cache entry. During an associative write, this represents the virtual address that will be translated to yield the physical address for comparison with all the tags in the indexed set. | |
| | Reset | | Undefined | |

The following three kinds of operation can be used on the OC address array:

OC address array read

The TAG, U and V bits are read from the OC entry corresponding to the SET and WAY fields of the address. Regardless of the setting of the A-bit, array reads are never performed associatively.

OC address array write (non-associative)

The ASSOC bit in the address must be 0 to select this operation.

The TAG, U and V bits specified in the stored value are written to the OC entry corresponding to the SET and WAY fields of the address.

If the write is to an entry in which both the U and V bits are set beforehand, a write-back of the associated cache line is always performed, even when the new values of the U, V and tag fields match the old ones. The tag, U bit, and V bit are then updated to the values specified in the data field.

OC address array write (associative)

The ASSOC bit in the address must be 1 to select this operation.

The TAG field of the stored value is examined. It is treated as bits [31:10] of a virtual address, with bits [9:0] treated as zero. This virtual address is translated to a physical address by the MMU as though it were a regular data access, see [Table 11: Summary of virtual to physical translation in each virtual address region on page 47](#). The upper bits resulting from the mapping are compared against the TAG field in all cache entries in the set defined by the SET field of the address. If a matching entry is found with its V bit set to 1, the V and U bits from the stored value are written to the V and U bits of the matching cache entry. If no match is found, no further action occurs.

If the write is to an entry in which both the U and V bits are set beforehand, a write-back of the associated cache line is always performed, even when the new values of the U, V and tag fields match the old ones.

If the MMU lookup would have caused an RTLBMISSE/WTLBMISSE exception, this is ignored and no further action is taken.

If the MMU lookup detects a multiple hit in the UTLB/PMB, an OTLBMULTHIT exception is launched.

4.5.4 OC data array

The OC data array is allocated to virtual addresses 0xF500 0000 to 0xF5FF FFFF in the P4 region. A translation from another region cannot be used to access the array. Longword loads and stores (**MOV.L**) must be used when accessing the array. Loads are used to examine the contents of the OC. Stores are used to update the contents.

Access to the OC data array must adhere to the restrictions in [Section 4.7: Cache state coherency on page 145](#).

The address used for a load or store to the array defines the entry number to be acted on for addressed operations.

The address is formed as shown in [Table 75](#). N is determined by the cache size. Refer to [Table 69 on page 132](#) for the bits used for SET and WAY in each cache configuration. Note that the setting of CCR.OIX has no effect on the entry selection for a memory-mapped access.

However, CCR.ORA may affect the indexing. For a ST40-100 series part, or a ST40-202 operating in direct-mapped (compatibility) mode, the cache entries used as RAM may still be accessed through the memory-mapped address and data arrays. However for other variants, the indexing behaves as though the cache were half its natural size when CCR.ORA = 1 for indexing. [Table 69 on page 132](#) defines the indexing and way selection bits based on the cache configuration. [Table 76](#) defines the addressing of the cache entries used as RAM for the ST40-100 series and ST40-202 in compatibility mode.

Table 75. Addressing for the OC data array

| Address bit range | Field size | Field name | Purpose |
|-------------------|------------|------------|--|
| [31:24] | 8 | 0xF5 | Selects the OC data array within the P4 address region |
| [23:N+1] | 23-N | zero | Unused bits. These should be zero for future compatibility |
| [N] | 1 | WAY | When the cache is 2-way set associative, this bit defines the way to act on. |
| [N-1:5] | N-5 | SET | These bits define the set in the OC that is to be acted upon. |
| [4:2] | 3 | WORD | Define the longword to access within the cache entry. |
| [1:0] | 2 | zero | Must be zero for longword address alignment. |

The data operand of the load or store contains the selected long word data element of the cache line.

Table 76. Address mappings for the 100-series and the ST40-202 in compatibility mode

| Configuration | ORA | OC data array addresses | Entries | Usage | RAM addresses |
|-----------------------------------|-----|-----------------------------|------------|-------|-----------------------------|
| 100-series 16Kbyte OC | 0 | 0xF500 0000- 0xF500 3FFF | 0 to 511 | Cache | - |
| | 1 | 0xF500 0000- 0xF500 0FFF | 0 to 127 | Cache | - |
| | | 0xF500 1000- 0xF500 1FFF | 128 to 255 | RAM | 0x7C00 1000- 0x7C00 1FFF |
| | | 0xF500 2000- 0xF500 2FFF | 256 to 383 | Cache | - |
| | | 0xF500 3000- 0xF500 3FFF | 384 to 511 | RAM | 0x7C00 2000- 0x7C00 2FFF |
| ST40-202 in compatibility mode | 0 | 0xF500 0000- 0xF500 3FFF | 0 to 511 | Cache | - |
| | 1 | 0xF500 0000- 0xF500 1FFF | 0 to 255 | Cache | - |
| | | 0xF500 2000- 0xF500 2FFF | 256 to 383 | RAM | 0x7C00 2000- 0x7C00 2FFF |
| | | 0xF500 3000- 0xF500 3FFF | 384 to 511 | RAM | 0x7C00 1000- 0x7C00 1FFF |

The following two kinds of operation can be used on the OC data array:

- OC data array read
The cache long word selected by the WAY, SET and WORD of the address is returned.
- OC data array write
The long word supplied in the stored value is written to the cache long word selected by the WAY, SET and WORD fields of the address.

4.6 Store queues (SQs)

The SQs are a pair of software-controlled write buffers. Software can load each buffer with 32 bytes of data, and then initiate a burst write of the buffer to memory. The CPU can continue to store data into one buffer whilst the other is being written out to memory, allowing efficient back-to-back operation for large data transfers.

When software never needs to examine the existing contents of some memory before overwriting it, the SQs are more efficient than the normal cache mechanisms:

- use of the SQ avoids polluting the cache with write-only data
- if a store were made to a cache line operating in copy-back mode, the current memory contents would be loaded into the cache at the beginning, potentially delaying the CPU whilst this load completes
- the store queues can be used with external resources that cannot respond to load requests

The store queue features are summarized in [Table 77](#).

Table 77. Store queue features

| Item | Store queues |
|-------------------|--|
| Capacity | 2 * 32 bytes |
| Virtual addresses | 0xE000 0000 to 0xE3FF FFFF |
| Write | Store instruction (MOV.L , STS.L , STC.L , FMOV.S or FMOV) (1-cycle write) |
| Write-back | Prefetch instruction (PREF) |
| Access right | MMU off: according to MMUCR.SQMD MMU on: according to MMUCR.SQMD and individual page protection bits. |

4.6.1 Power-save functions

The store queues functions are driven by a clock separate from the rest of the cache controller functions. If software is not using the SQs, a low power mode in which the SQ clock is stopped can be used. Product-specific documentation describes how the clock may be stopped.

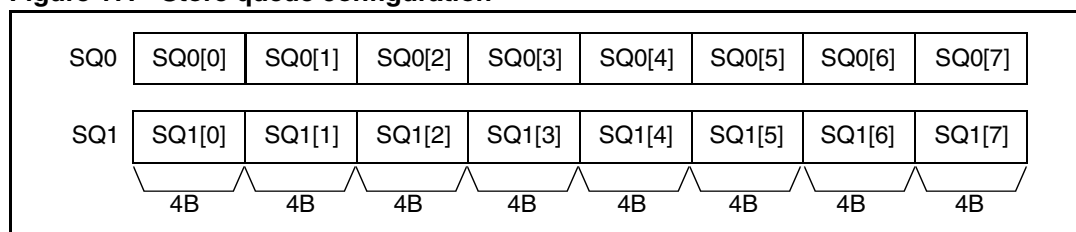
When the clock is stopped, the following functions cannot be used:

- reading and writing QACR0 and QACR1
- stores to the store queue virtual address region (0xE000 0000 through 0xE3FF FFFF)
- loads from the store queue (read) virtual address region (0xF000 1000 through 0xFF00 103C)
- prefetch (**PREF**) to the store queue virtual address region (0xE000 0000 through 0xE3FF FFFF)

4.6.2 SQ configuration

There are two 32-byte store queues, SQ0 and SQ1, as shown in [Figure 17](#). These two store queues can be used independently.

Figure 17. Store queue configuration



4.6.3 SQ writes

A SQ write is used to copy data from the CPU/FPU registers into the SQ.

A write is performed using a store instruction to a virtual address in the P4 region 0xE000 0000 to 0xE3FF FFFC. A long-word (**MOV.L**, **STS.L**, **STC.L** or **FMOV.S** (with FPSCR.SZ=0)) or quad-word (**FMOV** with FPSCR.SZ=1) must be used.

The meaning of the address bits are as follows:

| | | |
|----------|------------------|---|
| [31:26]: | 111000 | Store queue area |
| [25:10]: | | When MMUCR.AT=0: ignored When MMUCR.AT=1: used for UTLB lookup to determine access rights. |
| [9:6]: | Don't care | Ignored |
| [5]: | 0/1 | 0: selects SQ0 1: selects SQ1 |
| [4:2]: | LW specification | Specifies the required longword in SQ0/SQ1 |
| [1:0] | 00 | Must be 0 for longword address alignment. |

Access rights for SQ writes are described in [Section 4.6.7: Access rights for store queue operations on page 144](#).

On a 100-series core, if an exception occurs during a SQ write, the SQ contents may be corrupted. However, on all other variants, the SQ write access is cancelled and the data before the SQ write access is preserved.

4.6.4 SQ reads

A read from the SQs is used to examine the current SQ contents.

A read is performed using a load instruction on P4 area 0xFF00 1000 to 0xFF00 103C. Such a read can only be performed from privileged mode; there is no other access control governing this operation. Access rights for SQ reads are described in [Section 4.6.7 on page 144](#).

A long-word (**MOV.L**) instruction must be used. The meaning of the address bits is as follows:

| | | |
|---------|------------------|---|
| [31:8]: | 0xFF0 0100 | Store queue read area |
| [7:6]: | 0b00 | Zero |
| [5]: | 0/1 | 0: selects SQ0 1: selects SQ1 |
| [4:2]: | LW specification | Specifies the required long word in SQ0/SQ1 |
| [1:0]: | 00 | Must be 0 for longword address alignment. |

4.6.5 Transfer to external memory

Transfer from a SQ to external memory can be performed with the prefetch instruction (**PREF**). Issuing a **PREF** instruction operating on a virtual address in the P4 range 0xE000 0000 to 0xE3FF FFFC starts a burst transfer from the SQ to external memory.

The burst transfer has a fixed length of 32 bytes, and the start address of the overwritten memory is always at a 32-byte boundary. However, bits [4:0] of the virtual address used with the **PREF** instruction are ignored.

Access rights for SQ transfers to memory are described in [Section 4.6.7: Access rights for store queue operations on page 144](#).

While the contents of one SQ are being transferred to external memory, the other SQ can be written to independently by the CPU. The write to the other SQ does not wait for the external transfer to complete in this case.

Conversely, if the CPU writes to the SQ that is being transferred to external memory, the CPU pipeline will wait until the external transfer is completed so that the SQ contents can be updated without conflicting with the transfer.

4.6.6 Physical address for external transfer

The physical address to which the SQ external transfer is made using 'PREF @Rn' is defined in [Table 78](#).

When MMUCR.AT = 0, the physical address is derived from the QACR registers and Rn directly, as shown in the table.

When MMUCR.AT = 1, the virtual address Rn is looked up in the UTLB. [Table 79](#) defines the outcome of the lookup. If the lookup is successful, [Table 78](#) assumes that UTLB[i] is the matching entry. Bits [19:10] of the physical address are calculated in the usual way as a mixture of bits from UTLB[i].PPN and Rn, depending on the page size of UTLB[i]. The C and WT bits of UTLB[i] have no relevance for a SQ transfer.

Table 78. Determination of physical address for store queue external transfer

| MMUCR.AT | MMUCR.SE / PASC.R.SE | Rn[5] | Physical address bits | | | | | |
|----------|-------------------------|-------|-----------------------|--------------------|----------|---|---------|------------------------------------|
| | | | [31:29] | [28:26] | [25:20] | [19:10] | [9:5] | [4:0] |
| 0 | 0 | 0 | 0b000 | QACR0[4:2] | Rn[25:5] | | | All 32 bytes are transferred |
| | | 1 | 0b000 | QACR1[4:2] | Rn[25:5] | | | |
| | 1 | 0 | QACR0[7:2] | | | Rn[25:5] | | |
| | | 1 | QACR1[7:2] | | | Rn[25:5] | | |
| 1 | 0 | 0 | 0b000 | UTLB[i].PPN[28:20] | | UTLB[i].PPN combined with Rn[19:10], depending on page size | Rn[9:5] | |
| | | 1 | 0b000 | UTLB[i].PPN[28:20] | | | Rn[9:5] | |
| | 1 | 0 | UTLB[i].PPN[31:20] | | | | Rn[9:5] | |
| | | 1 | UTLB[i].PPN[31:20] | | | | Rn[9:5] | |

4.6.7 Access rights for store queue operations

[Table 79](#) defines the behavior when using **MOV.L** (store) or **PREF** to access either of the store queues. A grey box indicates a wildcard bit. The *Outcome* column indicates when the operation will succeed, or the type of exception when the operation will fail.

Table 79. Access rights and exception types for operations on the store queues

| Operation | SR.MD | MMUCR.SQMD | MMUCR.AT | UTLB entry | | | Outcome |
|---|-------|------------|-------------------|-------------------|------------|------------|------------|
| | | | | PR[1] | PR[0] | D | |
| STORE <i>Section 4.6.3</i> | 0 | 0 | 0 | | | | successful |
| | | | 1 | no matching entry | | | WTLBMISS |
| | | | | 1 | 1 | 1 | successful |
| | | | | | | 0 | FIRSTWRITE |
| | | | | 0 | | | WRITEPROT |
| | | | | 0 | | | |
| | 1 | | | | WADDERR | | |
| | 1 | | 0 | | | | successful |
| | | | 1 | no matching entry | | | WTLBMISS |
| | | | | | 1 | 1 | successful |
| | | | | | | 0 | FIRSTWRITE |
| | | | | 0 | | | WRITEPROT |
| 0 | | | | | | | |
| LOAD <i>Section 4.6.4</i> | 0 | | | | RADDERR | | |
| | 1 | | | | successful | | |
| External transfer <i>Section 4.6.5</i> | 0 | 0 | 0 | | | | successful |
| | | | 1 | no matching entry | | | RTLBMIS |
| | | | | 1 | | | successful |
| | | | | | | | READPROT |
| | | | | 0 | | | |
| | | | 0 | | | | |
| | 1 | | | | RADDERR | | |
| | 1 | | 0 | | | | successful |
| 1 | | | no matching entry | | | RTLBMIS | |
| | | | | | | successful | |

4.7 Cache state coherency

Software must ensure coherence between the instruction stream and access to the cache state by adhering to restrictions regarding the address region and cacheability of the instruction address of the **MOV.L** instruction used to access the cache state. If these restrictions are not followed:

- when the cache state is changed by software, some subsequent instructions may be executed without taking account of the state change
- when the cache state is read by software, an inconsistent value may be read
- on the 100-, 200-, 400- and 500- series cores, it is also possible for the **MOV.L** instruction which reads or writes the cache state to interact with nearby instructions so as to put the cache into an undefined state

The restrictions define the virtual address region in which the PC must be at the time of the **MOV.L**, the instruction cacheability of that region, and the precautions that must be taken afterwards before making an instruction or data access to another region. [Table 80](#) shows these restrictions on PC placement and later access to other regions.

Table 80. Coherency requirements for access to cache state

| Resource | Operation | PC of accessing instruction must be in | | Cohere later instruction fetch from | | Cohere later data access to | |
|-------------------|--|--|-----------------------------------|-------------------------------------|-------------------|-------------------------------|-------------------|
| | | 29-bit (SE=0) | 32-bit (SE=1) | U0/P0 & P3 | P1 ⁽¹⁾ | U0/P0, P3 & SQ ⁽²⁾ | P1 ⁽¹⁾ |
| CCR, RAMCR | Read | No restriction | No restriction | N/A | N/A | N/A | N/A |
| | Write | P2 or P4 | P1 or P2 (uncacheable page) or P4 | Yes | Yes | Yes | Yes |
| Instruction cache | Memory-mapped read or write (see Section 4.5.1 and Section 4.5.2) | P2 or P4 | P1 or P2 (uncacheable page) or P4 | Yes | Yes | Yes | Yes |
| Operand cache | Memory-mapped read or write (see Section 4.5.3 and Section 4.5.4) | P1, P2 or P4 | P1, P2 or P4 | Yes | No | Yes | No |

1. Or a cacheable page in P2 when in space-enhancement mode.

2. SQ = store-queues

The methods for achieving coherency are detailed in [Section 3.15.2: Achieving coherency \(ST40-100, ST40-200, ST40-400 and ST40-500 series cores\) on page 106](#) and [Section 3.15.3: Achieving coherency \(ST40-300 series cores\) on page 106](#).

On the ST40-300 series, any write to CCR that might change CCR.OCE from 1 to 0 should be preceded by a **SYNCO** instruction.

5 Exceptions

The process of responding to an extraordinary event such as a reset, a general exception (trap) or an interrupt, is called exception handling. Exception handling is performed by user supplied special routines, that are executed by the CPU when one of these extraordinary events is encountered.

5.1 Register descriptions

There are three registers related to exception handling. These are allocated to memory, and can be accessed by specifying the P4 address shown in [Table 81](#) or through an address translation from another virtual address region as described in [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#).

Table 81. Exception-related registers

| Name | Abbreviation | R/W | Initial value | P4 address ⁽¹⁾ | Access size |
|--------------------------|--------------|-----|--|---------------------------|-------------|
| TRAPA exception register | TRA | R/W | Undefined | 0xFF00 0020 | 32 |
| Exception event register | EXPEVT | R/W | 0x0000 0000/ 0x0000 0020 ⁽²⁾ | 0xFF00 0024 | 32 |
| Interrupt event register | INTEVT | R/W | Undefined | 0xFF00 0028 | 32 |

1. This is the P4 virtual address at which the register may be accessed.
2. 0x0000 0000 is set in a power-on reset, and 0x0000 0020 in a manual reset.

5.1.1 Exception event register (EXPEVT)

The exception event register (EXPEVT) resides at P4 address 0xFF00 0024, and contains a 14-bit exception code. The exception code set in EXPEVT is that for a reset or general exception event. The exception code is set automatically by hardware when an exception occurs. EXPEVT can also be modified by software.

Table 82. EXPEVT register description

| EXPEVT | | | | |
|----------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Exception code | [11:0] | 12 | Exception code | RW |
| | Operation | | Exception code set automatically by hardware when exception occurs. | |
| | Reset | | 0x000 at power-on reset, 0x020 at manual reset. | |
| RES | [31:12] | 20 | Bits reserved | RW |
| | Reset | | 0 | |

5.1.2 Interrupt event register (INTEVT)

The interrupt event register (INTEVT) resides at P4 address 0xFF00 0028, and contains a 14-bit exception code. The exception code set in INTEVT is that for an interrupt request. The exception code is set automatically by hardware when an exception occurs. INTEVT can also be modified by software.

Table 83. INTEVT register description

| INTEVT | | | | |
|----------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Exception code | [13:0] | 14 | Exception code | RW |
| | Operation | | Exception code set automatically by hardware when exception occurs. | |
| | Reset | | Undefined | |
| RES | [31:14] | 18 | Bits reserved | RW |
| | Reset | | Undefined | |

5.1.3 TRAPA exception register (TRA)

The TRAPA exception register (TRA) resides at P4 address 0xFF00 0020. TRA is set automatically by hardware when a TRAPA instruction is executed. TRA can also be modified by software.

Table 84. TRA register description

| TRA | | | | |
|-------|----------------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Imm | [9:2] | 8 | 8-bit immediate data for the TRAPA instruction. | RW |
| | Operation | | | |
| | Reset | | Undefined | |
| RES | [1:0], [31:10] | 24 | Bits reserved | RW |
| | Reset | | Undefined | |

5.2 Exception handling functions

5.2.1 Exception handling flow

In exception handling, the contents of the program counter (PC), status register (SR) and R15 are saved in the saved program counter (SPC), saved status register (SSR) and saved general register (SGR). The CPU starts execution of the appropriate exception handling routine according to the vector address. An exception handling routine is a program the user writes to handle a specific exception. The exception handling routine is terminated and control returned to the original program, by executing a return-from-exception instruction (RTE). This instruction restores the PC and SR contents, and returns control to the normal processing routine at the point at which the exception occurred. The SGR contents are not written back to R15 by an RTE instruction.

The basic processing flow is as follows. See [Table 6: SR register description on page 25](#), for the meaning of the individual SR bits.

1. The PC, SR and R15 contents are saved in SPC, SSR and SGR.
2. The block bit (BL) in SR is set to 1.
3. The mode bit (MD) in SR is set to 1.
4. The register bank bit (RB) in SR is set to 1.
5. In a reset, the FPU disable bit (FD) in SR is cleared to 0.
6. The exception code is written to bits 11 to 0 of the exception event register (EXPEVT), or to bits 13 to 0 of the interrupt event register (INTEVT).
7. The CPU branches to the determined exception handling vector address, and the exception handling routine begins.

5.2.2 Exception handling vector addresses

Exception and interrupt vector addresses are determined by adding the offset for the specific event, to the vector base address, which is set by software in the vector base register (VBR). In the case of the TLB miss exception, for example, the offset is 0x0000 0400, so if 0x9C08 0000 is set in VBR, the exception handling vector address will be 0x9C08 0400.

If a further exception occurs at the exception handling vector address, a manual reset results because SR.BL = 1 on entry to an exception. Recovery from manual reset is difficult. Therefore, software must ensure that exceptions do not arise in exception handling routines. In particular, care must be taken to ensure a TLBMISS cannot arise when executing the exception handler. This is usually achieved by one of the following methods:

- for 29-bit addressing mode (SE = 0), the handler is placed at an untranslated virtual address, that is, in the P1 or P2 region
- for space-enhancement (32-bit) mode (SE = 1), the handler is placed in P1 or P2 and a translation is kept permanently in the PMB for it
- with MMUCR.AT = 1, the handler may be placed in P0 or P3, and software can ensure that the address of the handler is permanently resident in the UTLB using the MMUCR.URB facility
- with MMUCR.AT = 0, the handler may be placed in P0 or P3

5.3 Exception types and priorities

[Table 85](#) shows the types of exceptions, with their relative priorities, vector addresses, and exception/interrupt codes.

Table 85. Exceptions

| Exception category | Execution mode | Exception | Priority level | Priority order | Vector address | Offset | Exception code |
|--------------------|-------------------|--------------------------|----------------|----------------|--|---------|-----------------------------|
| Reset | Abort type | POWERON | 1 | 1 | See Section 2.8.2 on page 29 | - | 0x000 |
| | | MANRESET | 1 | 2 | | - | 0x020 |
| | | HUDIRESET | 1 | 1 | | - | 0x000 |
| | | ITLBMULTIHIT | 1 | 3 | | - | 0x140 |
| | | OTLBMULTIHIT | 1 | 4 | | - | 0x140 |
| General exception | Re-execution type | UBRKBEORE ⁽¹⁾ | 2 | 0 | (VBR/DBR) | 0x100/- | 0x1E0 |
| | | IADDERR | 2 | 1 | (VBR) | 0x100 | 0x0E0 |
| | | ITLBMISS | 2 | 2 | (VBR) | 0x400 | 0x040 |
| | | EXECROT | 2 | 3 | (VBR) | 0x100 | 0x0A0 |
| | | RESINST | 2 | 4 | (VBR) | 0x100 | 0x180 |
| | | ILLSLOT | 2 | 4 | (VBR) | 0x100 | 0x1A0 |
| | | FPUDIS | 2 | 4 | (VBR) | 0x100 | 0x800 |
| | | SLOTFPUDIS | 2 | 4 | (VBR) | 0x100 | 0x820 |
| | | RADDERR | 2 | 5 | (VBR) | 0x100 | 0x0E0 |
| | | WADDERR | 2 | 5 | (VBR) | 0x100 | 0x100 |
| | | RTLBMISS | 2 | 6 | (VBR) | 0x400 | 0x040 |
| | | WTLBMISS | 2 | 6 | (VBR) | 0x400 | 0x060 |
| | | READROT | 2 | 7 | (VBR) | 0x100 | 0x0A0 |
| | | WRITEROT | 2 | 7 | (VBR) | 0x100 | 0x0C0 |
| | | FPUExc | 2 | 8 | (VBR) | 0x100 | 0x120 |
| | | FIRSTWRITE | 2 | 9 | (VBR) | 0x100 | 0x080 |
| | Completion type | TRAP | 2 | 4 | (VBR) | 0x100 | 0x160 |
| | | UBRKAFTEr ⁽¹⁾ | 2 | 10 | (VBR/DBR) | 0x100/- | 0x1E0 |
| Interrupt | Completion type | NMI | 3 | - | (VBR) | 0x600 | 0x1C0 |
| | | IRLINT ⁽²⁾ | 4 | (2) | (VBR) | 0x600 | See footnote ⁽²⁾ |
| | | PERIPHINT ⁽²⁾ | 4 | (2) | (VBR) | 0x600 | |

1. When BRcR.UBDE = 1, PC = DBR. In other cases, PC = VBR + 0x100.

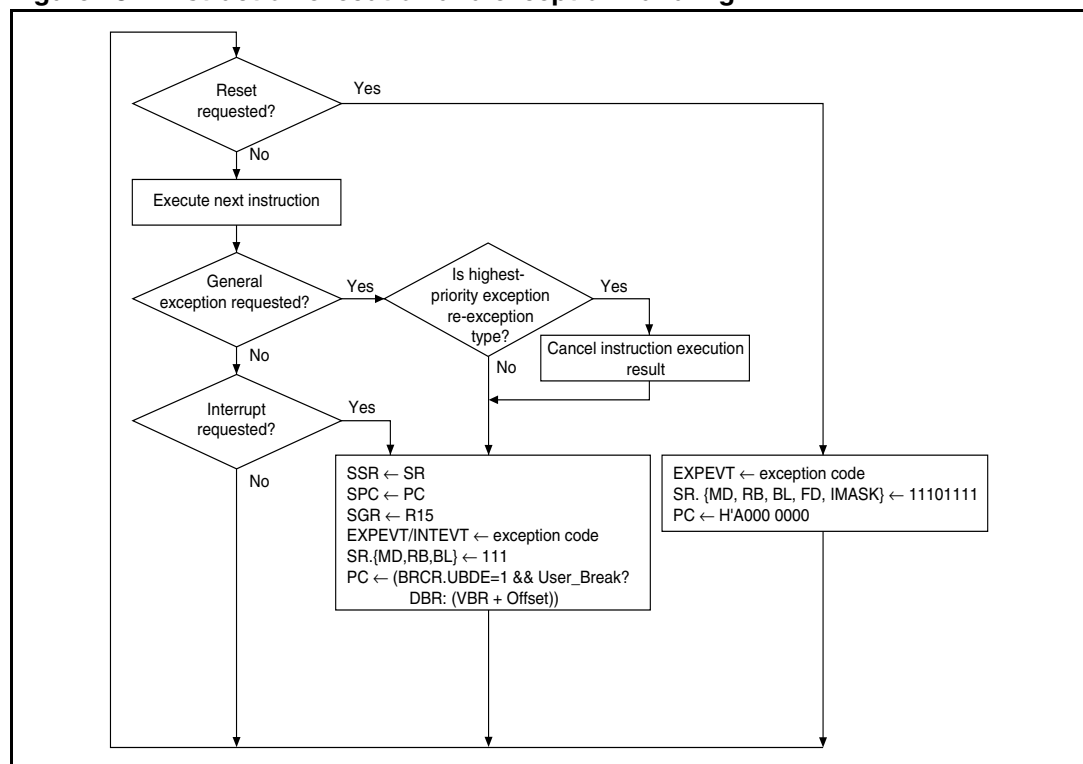
2. The priority order of external interrupts and peripheral module interrupts can be set by software, in addition, the set of peripheral interrupts is system-dependent. See the Interrupt Controller chapter in the [Core Support Peripherals Architecture Manual](#) for the list of peripheral interrupts and their corresponding INTEVT codes.

| | |
|---|---|
| Priority | Priority is first assigned by priority level, then by priority order within each level (the lowest number represents the highest priority). |
| Exception transition destination | Control passes to the reset address (Section 2.8.2: Reset address on page 29) in a reset, and to [VBR + offset] in other cases. |
| Exception code | Stored in EXPEVT for a reset or general exception, and in INTEVT for an interrupt. |
| IRL | Interrupt request level (as indicated by the IRL3-IRL0 signals). |
| Module/source | See the sections on the relevant peripheral modules. |

5.4 Exception flow

[Figure 18](#) shows an outline flowchart of the basic operations in instruction execution and exception handling. For the sake of clarity, the following description assumes that instructions are executed sequentially and atomically. Register settings in the event of an exception are shown only for SSR, SPC, EXPEVT/INTEVT, SR, and PC, but other registers may be set automatically by hardware, depending on the exception, for details, see [Section 5.5: Description of exceptions on page 153](#).

Figure 18. Instruction execution and exception handling

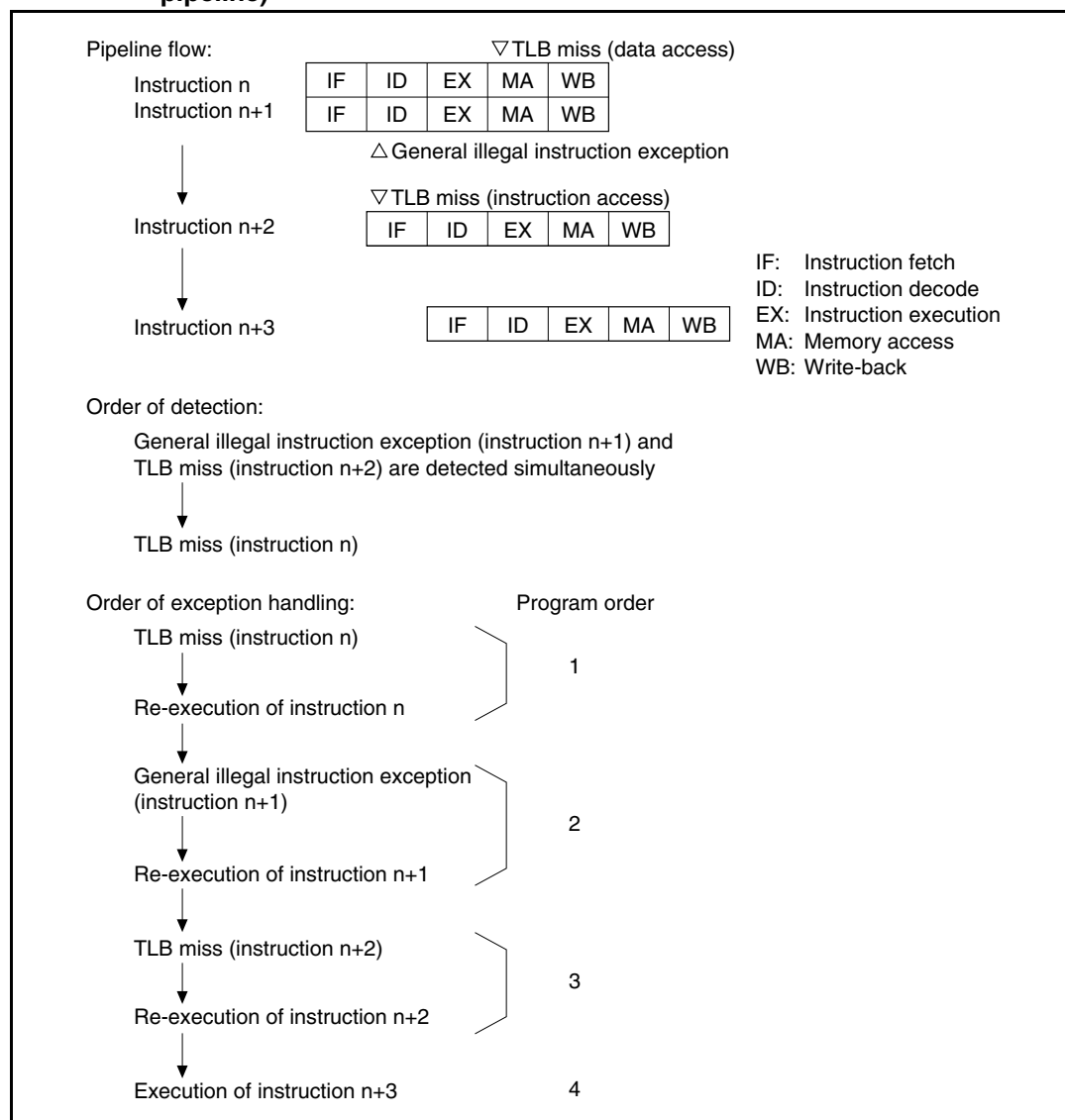


5.4.1 Exception source acceptance

A priority ranking is provided for all exceptions, for use in determining which of two or more simultaneously generated exceptions should be accepted. Five of the general exceptions:

- general illegal instruction exception
- slot illegal instruction exception
- general FPU disable exception
- slot FPU disable exception
- unconditional trap exception

are detected in the process of instruction decoding, and do not occur simultaneously in the instruction pipeline. Therefore, these exceptions all have the same priority. General exceptions are detected in the order of instruction execution. However, exception handling is performed in the order of instruction flow (program order). Thus, an exception for an earlier instruction is accepted before that for a later instruction. An example of the order of acceptance for general exceptions is shown in [Figure 19](#).

Figure 19. Example of general exception acceptance order (illustrated for 5-stage pipeline)

5.4.2 Exception requests and BL bit

When the BL bit in SR is 0, exceptions and interrupts are accepted.

When the BL bit in SR is 1 and an exception other than a user break is generated, the CPU's internal registers are set to their post-reset state, the registers of the other modules retain their contents prior to the exception, and the CPU branches to the same address as in a reset (see [Section 2.8.2: Reset address on page 29](#)). For the operation in the event of a user break see [Chapter 12: User break controller \(UBC\) on page 510](#). If an ordinary interrupt occurs, the interrupt request is held pending, and is accepted after the BL bit has been cleared to 0 by software. If a non-maskable interrupt (NMI) occurs, it can be held pending or accepted, according to the setting made by software.

Thus, normally, SPC and SSR are saved and then the BL bit in SR is cleared to 0, to enable multiple exception state acceptance.

5.4.3 Return from exception handling

The RTE instruction is used to return from exception handling. When the RTE instruction is executed, the SPC contents are restored to PC, and the SSR contents to SR. The CPU returns from the exception handling routine by branching to the SPC address. If SPC and SSR were saved to external memory, set the BL bit in SR to 1 before restoring the SPC and SSR contents and issuing the RTE instruction.

5.5 Description of exceptions

The various exception handling operations are described here, covering exception sources, transition addresses, and processor operation, when a transition is made.

5.5.1 Resets

1. POWERON - power-on reset

- Sources:
For details of how the core can be driven to the power-on reset state, refer to the datasheet, *Core Support Peripherals Architecture Manual* and *Emulation Support Peripheral Architecture Manual* of the appropriate product.
- Transition address: see [Section 2.8.2: Reset address on page 29](#).
- Transition operations:
Exception code 0x000 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to the reset address ([Section 2.8.2: Reset address on page 29](#)). In the initialization processing, the VBR register is set to 0x0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3-I0) are set to 0xF.
CPU initialization is performed.
Refer to [Appendix A: Address list on page 535](#) for power-on reset values for the various CPU core modules set by the `Initialize_Module` function.

```
POWERON()
{
    Initialize_Module(PowerOn);
    EXPEVT = 0x00000000;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD=0;
    PC = (reset address);
}
```

2. MANRESET - manual reset

- Sources:
When a general exception other than a user break occurs while the BL bit is set to 1 in SR. It is also possible for the system in which the core is integrated to drive the processor in to this reset state. For details refer to the datasheets, *Core Support Peripherals Architecture Manual* and *Emulation Support Peripheral Architecture Manual* of the appropriate product.
- Transition address: [Section 2.8.2: Reset address on page 29](#)
- Transition operations:
Exception code 0x020 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to the reset address ([Section 2.8.2: Reset address on page 29](#)). In the initialization processing, the VBR register is set to 0x0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3-I0) are set to 0xF. CPU and system initialization are performed.
Refer to [Appendix A: Address list on page 535](#) for the manual reset values for the various CPU core modules set by the `Initialize_Module` function.

```
MANRESET()
{
    Initialize_Module(Manual);
    EXPEVT = 0x00000020;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = (reset address);
}
```

3. HUDIRESET - H-UDI reset

- Source:
Refer to the *Emulation Support Peripheral Architecture Manual* for a description of how the core is placed in the H-UDI reset state.
- Transition address: [Section 2.8.2: Reset address on page 29](#).
- Transition operations:
Exception code 0x000 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to the reset address ([Section 2.8.2: Reset address on page 29](#)). In the initialization processing, the VBR register is set to 0x0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3-I0) are set to 0xF. CPU and system initialization are performed.
Refer to [Appendix A: Address list on page 535](#) for the power-on reset values for the various CPU core modules set by the `Initialize_Module` function.

```

HUIDRESET(ResetType)
{
    Initialize_Module(PowerOn);
    EXPEVT = 0x00000000;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = (reset address);
}

```

4. ITLBMULTIHIT - Instruction TLB Multiple-Hit Exception

- Source: Multiple ITLB address matches.
- Transition address: [Section 2.8.2: Reset address on page 29](#).
- Transition operations:
 The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.
 Exception code 0x140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to the reset address ([Section 2.8.2: Reset address on page 29](#)).
 In the initialization processing, the VBR register is set to 0x0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3-I0) are set to 0xF.
 CPU and system initialization are performed in the same way as in a manual reset.
 Refer to [Appendix A: Address list on page 535](#) for the manual reset values for the various CPU core modules set by the Initialize_Module function.

```

ITLBMULTIHIT()
{
    Initialize_Module(Manual);
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = 0x00000140;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = (reset address);
}

```

5. OTLBMULTIHIT - operand TLB multiple-hit exception

- Source: Multiple UTLB address matches.
- Transition address: [Section 2.8.2: Reset address on page 29](#).
- Transition operations:
 The virtual address (32 bits) at which this exception occurred is set in TEA, and the corresponding virtual page number (22 bits) is set in PTEH [31:10]. ASID in PTEH indicates the ASID when this exception occurred.
 Exception code 0x140 is set in EXPEVT, initialization of VBR and SR is performed, and a branch is made to the reset address ([Section 2.8.2: Reset address on page 29](#)).
 In the initialization processing, the VBR register is set to 0x0000 0000, and in SR, the MD, RB, and BL bits are set to 1, the FD bit is cleared to 0, and the interrupt mask bits (I3-I0) are set to 0xF.
 CPU and system initialization are performed in the same way as in a manual reset.
 Refer to [Appendix A: Address list on page 535](#) for the manual reset values for the various CPU core modules set by the `Initialize_Module` function.

```
OTLBMULTIHIT()
{
    Initialize_Module(Manual);
    TEA = EXCEPTION_ADDRESS;
    PTEH.VPN = PAGE_NUMBER;
    EXPEVT = 0x00000140;
    VBR = 0x00000000;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    SR.(I0-I3) = 0xF;
    SR.FD = 0;
    PC = (reset address);
}
```

5.5.2 General exceptions

All general exceptions listed in [Table 86](#) make the following updates to the machine state:

- SR is saved to SSR
- R15 is saved to SGR
- EXPEVT is set to the exception code shown in [Table 85](#)
- PC is set to the vector address + offset shown in [Table 85](#)
- SR.MD, SR.RB and SR.BL are all set to 1

[Table 86](#) defines the updates to SPC, TEA and PTEH[31:10] for each type of general exception.

Table 86. Updating of registers by general exceptions

| Exception | SPC ⁽¹⁾ | TEA ⁽²⁾ | PTEH ⁽²⁾ [31:10] | Source |
|------------|--------------------|--------------------|--------------------------------|---|
| UBRKBEFORE | [E] | Unmodified | | Matching a break condition set in the user break controller. |
| IADDERR | [A] | IA | IA[31:10] | 1. Instruction fetch from other than a word boundary. 2. Instruction fetch from the area 0x8000 0000 to 0xFFFF FFFF in user mode. |
| ITLBMISS | [A] | IA | IA[31:10] | Address mismatch in ITLB address comparison. |
| EXECPROT | [A] | IA | IA[31:10] | The access does not agree with the ITLB protection information (PR bits), see Table 29 . |
| RESINST | [C] | Unmodified | | 1. Decoding of an undefined instruction, not in a branch delay slot. The opcode 0xFFFFD is guaranteed to be an undefined instruction in any ST40 architecture revision. Other unused opcodes may be treated as reserved in any particular ST40 implementation. 2. The decoding in user mode of a privileged instruction not in a branch delay slot. The privileged instructions are: LDC , STC , LDC.L , STC.L , RTE , LDTLB and SLEEP , but excluding LDC[L] and STC[L] that access GBR. |
| ILLSLOT | [B] | Unmodified | | 1. Decoding of an undefined instruction, in a branch delay slot. Undefined instructions are defined in RESINST(1). 2. Decoding of a privileged instruction in a branch delay slot in user mode. Privileged instructions are defined in RESINST(2). 3. Decoding of the following instructions in a branch delay slot: JMP , JSR , BRA , BRAF , BSR , BSRF , RTS , RTE , BT , BF , BT/S , BF/S , TRAPA , LDC Rm , SR , LDC.L @Rm+ , SR , MOVA , MOV.W @(disp,PC) , Rn , MOV.L @(disp,PC) , Rn . |
| FPUDIS | [C] | Unmodified | | Decoding of an FPU instruction not in a branch delay slot when SR.FD = 1 ⁽³⁾ . FPU instructions are those whose first four bits are 0xF (excluding 0xFFFFD), and the LDS , STS , LDS.L and STS.L instructions that access FPSCR and FPUL. |
| SLOTFPUDIS | [B] | Unmodified | | Decoding of an FPU instruction in a branch delay slot when SR.FD = 1 ⁽³⁾ . FPU functions are described in FPUDIS above. |
| RADDERR | [A] | OA | OA[31:10] | 1. Word access from other than a word boundary (2n+1). 2. Longword access from other than a longword boundary (4n+1, 4n+2 or 4n+3). 3. Quadword access from other than a quadword boundary (8n+1, 8n+2, 8n+3, 8n+4, 8n+5, 8n+6 or 8n+7). 4. Access to the area 0x8000 0000 to 0xFFFF FFFF in user mode. |
| WADDERR | [A] | OA | OA[31:10] | 1. Word access from other than a word boundary (2n+1). 2. Longword access from other than a longword boundary (4n+1, 4n+2 or 4n+3). 3. Quadword access from other than a quadword boundary (8n+1, 8n+2, 8n+3, 8n+4, 8n+5, 8n+6 or 8n+7). 4. Access to the area 0x8000 0000 to 0xFFFF FFFF (except for the store queue area 0xE000 0000 to 0xE3FF FFFF) in user mode. |
| RTLBMIS | [A] | OA | OA[31:10] | Address mismatch in UTLB address comparison. |
| WTLBMIS | [A] | OA | OA[31:10] | Address mismatch in UTLB address comparison. |

Table 86. Updating of registers by general exceptions (continued)

| Exception | SPC ⁽¹⁾ | TEA ⁽²⁾ | PTEH ⁽²⁾ [31:10] | Source |
|------------|--------------------|--------------------|--------------------------------|--|
| READPROT | [A] | OA | OA[31:10] | The access does not agree with the UTLB protection information (PR bits), see Table 29 . |
| WRITEPROT | [A] | OA | OA[31:10] | The access does not agree with the UTLB protection information (PR bits), see Table 29 . |
| FPUExc | [A] | Unmodified | | Exception arising due to the execution of a floating point operation. |
| FIRSTWRITE | [A] | OA | OA[31:10] | UTLB match is found for a store access, but the page's D-bit is clear, see Table 29 . |
| TRAP | [D] | Unmodified | | Execution of a TRAPA instruction. |
| UBRKAFter | [F] | Unmodified | | Matching a break condition set in the user break controller. |

1. See [Table 87](#).

2. IA = instruction address (virtual), OA = operand address (virtual).

3. Any ST40 without an FPU always has SR.FD = 1.

[Table 87](#) shows how SPC is updated for each exception type depending on whether the exception happens on a branch, a branch delay slot or some other non-branch instruction.

If a delay slot instruction is executed separately from the branch before it, it is treated as non-branch in [Table 87](#). This can arise if:

- a branch is executed whose target is that delay slot instruction
- an RTE instruction causes control to jump to the delay slot instruction

Table 87. Updating of SPC by general exceptions

| Scenario | Branch PC | Delay slot PC | Branch target PC | Exception group ⁽¹⁾ | | | | | |
|------------------|--------------------|---------------|------------------|--------------------------------|-----|-----|-----|-----|-----|
| | | | | [A] | [B] | [C] | [D] | [E] | [F] |
| Taken branch | B | | T | B | | | | B | T |
| | | B+2 | T | B | B | | | B | T |
| Not taken branch | B | | T | B | | | | B | B+4 |
| | | B+2 | T | B | B | | | B | B+4 |
| Non-branch | Instruction PC = P | | | P | | P | P+2 | P | P+2 |

1. From [Table 86](#).

5.5.3 Interrupts

1. NMI - non-maskable interrupt

- Source: Refer to relevant *Core Support Peripherals Architecture Manual* for details of non-maskable interrupt generation (NMI).
- Transition address: VBR + 0x0000 0600.
- Transition operations:
 The PC and SR contents for the instruction at which this exception is accepted are saved in SPC and SSR. The R15 contents at this time are saved in SGR.
 Exception code 0x1C0 is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to PC = VBR + 0x0600.
 When the BL bit in SR is 0, this interrupt is not masked by the interrupt mask bits in SR, and is accepted at the highest priority level. When the BL bit in SR is 1, a software setting can specify whether this interrupt is to be masked or accepted. For details refer to the description of interrupt programming in the appropriate *Core Support Peripherals Architecture Manual*.

```
NMI ()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x000001C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}
```

2. IRLINT - IRL interrupts

- Source: The interrupt mask bit setting in SR is smaller than the IRL (3-0) level, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + 0x0000 0600.
- Transition operations:
 The PC contents immediately after the instruction at which the interrupt is accepted are set in SPC. The SR and R15 contents at the time of acceptance are set in SSR and SGR.
 The code corresponding to the IRL (3-0) level is set in INTEVT. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + 0x0600. The acceptance level is not set in the interrupt mask bits in SR. When the BL bit in SR is 1, the interrupt is masked. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller.

```

IRLINT()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x00000200 ~ 0x000003C0;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}

```

3. PERIPHINT - peripheral module interrupts

- Source: The interrupt mask bit setting in SR is smaller than the peripheral module interrupt level, and the BL bit in SR is 0 (accepted at instruction boundary).
- Transition address: VBR + 0x0000 0600 .
- Transition operations:
 The PC contents immediately after the instruction at which the interrupt is accepted are set in SPC. The SR and R15 contents at the time of acceptance are set in SSR and SGR.
 The code corresponding to the interrupt source is set in INTEVT. The BL, MD, and RB bits are set to 1 in SR, and a branch is made to VBR + 0x0600. The module interrupt levels should be set as values between 0x0 and 0xF in the interrupt priority registers (IPRA-IPRC) in the interrupt controller. For further details of the interrupt handling behavior, refer to the product level documentation of the interrupt controller.

```

Module_interruption()
{
    SPC = PC;
    SSR = SR;
    SGR = R15;
    INTEVT = 0x00000400 ~ 0x00000B80;
    SR.MD = 1;
    SR.RB = 1;
    SR.BL = 1;
    PC = VBR + 0x00000600;
}

```

5.5.4 Priority order with multiple exceptions

With some instructions, such as instructions that make two accesses to memory, and the indivisible pair comprising a delayed branch instruction and delay slot instruction, multiple exceptions occur. Care is required in these cases, as the exception priority order differs from the normal order.

1. Instructions that make two accesses to memory.

With MAC instructions, memory-to-memory arithmetic/logic instructions, and TAS instructions, two data transfers are performed by a single instruction, and an exception

will be detected for each of these data transfers. In these cases, therefore, the following order is used to determine priority.

- a) Data address error in first data transfer.
 - b) TLB miss in first data transfer.
 - c) TLB protection violation in first data transfer.
 - d) Data address error in second data transfer.
 - e) TLB miss in second data transfer.
 - f) TLB protection violation in second data transfer.
 - g) Initial page write exception in second data transfer.
2. Indivisible delayed branch instruction and delay slot instruction.

As a delayed branch instruction and its associated delay slot instruction are indivisible, they are treated as a single instruction. Consequently, the priority order for exceptions that occur in these instructions differs from the usual priority order. The priority order shown below is for the case where the delay slot instruction has only one data transfer.

- a) The delayed branch instruction is checked for priority levels 1 and 2.
- b) The delay slot instruction is checked for priority levels 1 and 2.
- c) A check is performed for priority level 3 in the delayed branch instruction and priority level 3 in the delay slot instruction. (There is no priority ranking between these two.)
- d) A check is performed for priority level 4 in the delayed branch instruction and priority level 4 in the delay slot instruction. (There is no priority ranking between these two.)

If the delay slot instruction has a second data transfer, two checks are performed in step b, as in 1 above.

If the accepted exception (the highest-priority exception) is a delay slot instruction re-execution type exception, the branch instruction PR register write operation (PC PR operation performed in BSR, BSRF, JSR) is inhibited.

5.6 Usage notes

1. Return from exception handling
 - a) Check the BL bit in SR with software.
If SPC and SSR have been saved to external memory, set the BL bit in SR to 1 before restoring them.
 - b) Issue an RTE instruction.
When RTE is executed, the SPC contents are set in PC, the SSR contents are set in SR, and branch is made to the SPC address to return from the exception handling routine.
2. If an exception or interrupt occurs when SR.BL = 1
 - a) Exception
When an exception other than a user break occurs, the CPU's internal registers are set to their post-reset state, the registers of the other modules retain their contents prior to the exception, and the CPU branches to the same address as in a reset. The reset address is defined in [Section 2.8.2: Reset address on page 29](#).

The value in EXPEVT at this time is 0x0000 0020. The value of the SPC and SSR registers is undefined.

b) Interrupt

If an ordinary interrupt occurs, the interrupt request is held pending and is accepted after the BL bit in SR has been cleared to 0 by software. If a nonmaskable interrupt (NMI) occurs, it can be held pending or accepted according to the setting made by software. In the sleep or standby state, an interrupt is accepted even if the BL bit in SR is set to 1.

3. SPC when an exception occurs

a) Re-execution type exception

The PC value for the instruction in which the exception occurred is set in SPC, and the instruction is re-executed after returning from exception handling. If an exception occurs in a delay slot instruction, the PC value for the delay slot instruction is saved in SPC, regardless of whether or not the preceding delay slot instruction condition is satisfied.

b) Completion type exception or interrupt

The PC value for the instruction following that in which the exception occurred is set in SPC. If an exception occurs in a branch instruction with delay slot, the PC value for the branch destination is saved in SPC.

4. An exception must not be generated in an RTE instruction delay slot, as the operation will be undefined in this case.

6 Floating-point unit

Note: This chapter is only relevant to ST40 products which incorporate an FPU.

For products where no FPU is present, the FD bit in the SR is permanently set and any attempt to execute an FPU instruction will cause an FPU disable exception. The recommended technique for checking if an FPU is present is to clear this bit and then check whether the write was successful by reading back the new value.

The floating-point unit (FPU) has the following features:

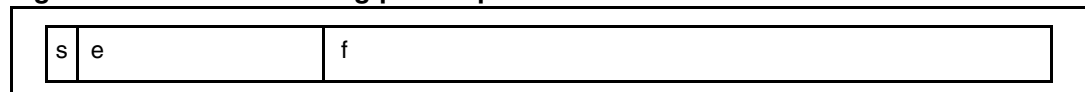
- conforms to IEEE754 standard
- 32 single-precision floating-point registers (can also be referenced as 16 double-precision registers)
- two rounding modes: round to nearest and round to zero
- two denormalization modes: flush to zero and treat denormalized number
- six exception sources: FPU error, invalid operation, divide by zero, overflow, underflow, and inexact
- comprehensive instructions: single-precision, double-precision, graphics support, system control

When the FD bit in SR is set to 1, the FPU cannot be used, and an attempt to execute an FPU instruction will cause a FPUDIS or SLOTFPUDIS exception.

6.1 Floating-point format

An IEEE754 floating-point number contains three fields: a sign (s), an exponent (e) and a fraction (f) in the format given in [Figure 20](#).

Figure 20. IEEE754 floating-point representations



The sign, s, is the sign of the represented number. If s is 0, the number is positive. If s is 1, the number is negative.

The exponent, e, is held as a biased value. The relationship between the biased exponent, e, and the unbiased exponent, E, is given by $e = E + \text{bias}$, where bias is a fixed positive number. The unbiased exponent, E, takes any value in the range $[E_{\min}-1, E_{\max}+1]$. The minimum and maximum values in that range, $E_{\min}-1$ and $E_{\max}+1$, designate special values such as positive zero, negative zero, positive infinity, negative infinity, denormalized numbers and “not a number” (NaN).

The fraction, f, specifies the binary digits that lie to the right of the binary point. A normalized floating-point number has a leading bit of 1 which lies to the left of the binary point. A denormalized floating-point number has a leading bit of 0 which lies to the left of the binary point. The leading bit is implicitly represented; it is determined by whether the number is normalized or denormalized, and is not explicitly encoded. The implicit leading bit and the explicit fraction bits together form the significance of the floating-point number.

Floating-point number value v is determined as follows:

The value, v , of a floating-point number is determined as follows:

NaN: if $E = E_{\max} + 1$ and $f \neq 0$, then v is Not a Number irrespective of the sign s

Positive or negative infinity: if $E = E_{\max} + 1$ and $f = 0$, then $v = (-1)^s (\infty)$

Normalized number: if $E_{\min} \leq E \leq E_{\max}$, then $v = (-1)^s 2^E(1.f)$

Denormalized number: if $E = E_{\min} - 1$ and $f \neq 0$, then $v = (-1)^s 2^{E_{\min}}(0.f)$

Positive or negative zero: if $E = E_{\min} - 1$ and $f = 0$, then $v = (-1)^s 0$

The architecture supports two IEEE754 basic floating-point number formats: single-precision and double-precision.

Table 88. Floating-point number formats and parameters

| Parameter | Single-precision | Double-precision |
|-----------------|------------------|------------------|
| Total bit width | 32 bits | 64 bits |
| Sign bit | 1 bit | 1 bit |
| Exponent field | 8 bits | 11 bits |
| Fraction field | 23 bits | 52 bits |
| Precision | 24 bits | 53 bits |
| Bias | +127 | +1023 |
| E_{\max} | +127 | +1023 |
| E_{\min} | -126 | -1022 |

[Table 89](#) shows the ranges of the various numbers in hexadecimal notation.

Table 89. Floating-point ranges

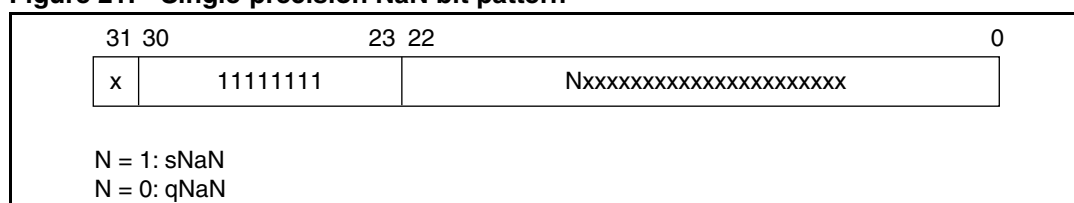
| Type | Single-precision | Double-precision |
|---------------------------------------|---|---|
| sNaN (Signaling not-a-number) | 0x7FFFFFFF to 0x7FC00000 and 0xFFC00000 to 0xFFFFFFFF | 0x7FFFFFFF 0xFFFFFFFF to 0x7FF80000 0x00000000 and 0xFF800000 0x00000000 to 0xFFFFFFFF 0xFFFFFFFF |
| qNaN (Quiet not-a-number) | 0x7FBFFFFF to 0x7F800001 and 0xFF800001 to 0xFFBFFFFF | 0x7FF7FFFF 0xFFFFFFFF to 0x7FF00000 0x00000001 and 0xFF000000 0x00000001 to 0xFF7FFFFF 0xFFFFFFFF |
| +INF (Positive infinity) | 0x7F800000 | 0x7FF00000 0x000000 |
| +NORM (Positive normalized number) | 0x7F7FFFFF to 0x00800000 | 0x7FEFFFFFFF 0xFFFFFFFF to 0x00100000 0x00000000 |
| +DNORM (Positive denormalized number) | 0x007FFFFF to 0x00000001 | 0x000FFFFFFF 0xFFFFFFFF to 0x00000000 0x00000001 |
| +0.0 (Positive zero) | 0x00000000 | 0x00000000 0x00000000 |
| - 0.0 (Negative zero) | 0x80000000 | 0x80000000 0x00000000 |

Table 89. Floating-point ranges (continued)

| Type | Single-precision | Double-precision |
|---------------------------------------|--------------------------|--|
| -DNORM (Negative denormalized number) | 0x80000001 to 0x807FFFFF | 0x80000000 0x00000001 to 0x800FFFFF 0xFFFFFFFF |
| -NORM (Negative normalized number) | 0x80800000 to 0xFF7FFFFF | 0x80100000 0x00000000 to 0xFFEFFFFF 0xFFFFFFFF |
| -INF (Negative infinity) | 0xFF800000 | 0xFFF00000 0x00000000 |

6.1.1 Non-numbers (NaN)

Figure 21 shows the bit pattern of a non-number (NaN).

Figure 21. Single-precision NaN bit pattern

A floating-point number is a NaN if the exponent field contains the maximum representable value and the fraction is non-zero, regardless of the value of the sign. In the figure above, x can have a value of 0 or 1. If the most significant bit of the fraction (N, in the figure above) is 1, the value is a signaling NaN (sNaN), otherwise the value is a quiet NaN (qNaN).

An sNaN is input in an operation, except copy, FABS, and FNEG, that generates a floating-point value.

- When the EN.V bit in the FPSCR register is 0, the operation result (output) is a qNaN.
- When the EN.V bit in the FPSCR register is 1, an invalid operation exception will be generated. In this case, the contents of the operation destination register are unchanged.

If a qNaN is input in an operation that generates a floating-point value, and an sNaN has not been input in that operation, the output will always be a qNaN irrespective of the setting of the EN.V bit in the FPSCR register. An exception will not be generated in this case.

See the individual instruction descriptions for details of floating-point operations when a non-number (NaN) is input.

6.1.2 Denormalized numbers

For a denormalized number floating-point value, the exponent field is expressed as 0, and the fraction field as a non-zero value.

When the DN bit in the FPU's status register FPSCR is 1, a denormalized number (source operand or operation result) is always flushed to 0 in a floating-point operation that generates a value (an operation other than **FMOV**, **FLDS**, **FSTS**, **FNEG**, or **FABS**).

When the DN bit in FPSCR is 0, a denormalized number (source operand or operation result) is processed as it is. See the individual instruction descriptions for details of floating-point operations when a denormalized number is input.

6.2 Rounding

In a floating-point instruction, rounding is performed when generating the final operation result from the intermediate result. Therefore, the result of combination instructions such as FMAC, FTRV, and FIPR will differ from the result when using a basic instruction such as FADD, FSUB, or FMUL. Rounding is performed once in FMAC, but twice in FADD, FSUB, and FMUL.

There are two rounding methods, the method to be used being determined by the RM field in FPSCR.

- RM = 0: round to nearest.
The value is rounded to the nearest expressible value. If there are two nearest expressible values, the one with an LSB of 0 is selected.
If the unrounded value is $2^{E_{\max}} (2 \cdot 2^{-P})$ or more, the result will be infinity with the same sign as the unrounded value. The values of E_{\max} and P , respectively, are 127 and 24 for single-precision, and 1023 and 53 for double-precision.
- RM = 1: round to zero.
The digits below the round bit of the unrounded value are discarded.
If the unrounded value is larger than the maximum expressible absolute value, the value will be the maximum expressible absolute value.

6.3 Floating-point exceptions

FPU-related exceptions are as given below.

- FPU disabled (FPUDIS) / slot FPU disabled (SLOTFPUDIS) exception.
The exception occurs if an FPU instruction is executed when SR.FD = 1.
- FPU exceptions (FPUEXC).
The exception sources are as follows:
 - FPU error (E): when FPSCR.DN = 0 and a denormalized number is input
 - invalid operation (V): in case of an invalid operation, such as NaN input
 - division by zero (Z): division with a zero divisor
 - overflow (O): when the operation result overflows
 - underflow (U): when the operation result underflows
 - inexact exception (I): when overflow, underflow, or rounding occurs

The FPSCR cause field contains bits corresponding to all of above sources E, V, Z, O, U, and I, and the FPSCR flag and enable fields contain bits corresponding to sources V, Z, O, U, and I, but not E. Thus, FPU errors cannot be disabled.

When an exception source occurs, the corresponding bits in the cause and flag fields are set to 1. When an exception source does not occur, the corresponding bit in the cause field is cleared to 0, but the corresponding bit in the flag field remains unchanged.

- Enable/disable exception handling.

The ST40 CPU core supports enable exception handling and disable exception handling.

Enable exception handling is initiated in the following cases:

- FPU error (E): FPSCR.DN = 0 and a denormalized number is input
- invalid operation (V): FPSCR.EN.V = 1 and (instruction = FTRV or invalid operation)
- division by zero (Z): FPSCR.EN.Z = 1 and division with a zero divisor
- overflow (O): FPSCR.EN.O = 1 and instruction with any possibility of the operation result overflowing
- underflow (U): FPSCR.EN.U = 1 and instruction with any possibility of the operation result underflowing
- inexact exception (I): FPSCR.EN.I = 1 and instruction with any possibility of an inexact operation result

These possibilities are shown in the individual instruction descriptions. All exception events that originate in the FPU are assigned as the same exception event. The meaning of an exception is determined by software by reading the system register FPSCR and interpreting the information it contains. If no bits are set in the cause field of FPSCR when one or more of bits O, U, I, and V (in case of FTRV only) are set in the enable field, this indicates that an actual exception source has not been generated. Also, the destination register is not changed by any enable exception handling operation.

Except for the above, the FPU disables exception handling. In all processing, the cause and flag bits corresponding to source V, Z, O, U, or I is set to 1, and disable exception handling is provided for each exception.

- Invalid operation (V): qNAN is generated as the result.
- Division by zero (Z): Infinity with the same sign as the unrounded value is generated.
- Overflow (O):
when rounding mode = RZ, the maximum normalized number, with the same sign as the unrounded value, is generated
when rounding mode = RN, infinity with the same sign as the unrounded value is generated
- Underflow (U):
when FPSCR.DN = 0, a denormalized number with the same sign as the unrounded value, or zero with the same sign as the unrounded value, is generated
when FPSCR.DN = 1, zero with the same sign as the unrounded value, is generated
- Inexact exception (I): An inexact result is generated.

6.4 Graphics support functions

The ST40 CPU core supports two kinds of graphics functions: instructions for geometric operations, and pair single-precision transfer instructions that enable high-speed data transfer.

6.4.1 Geometric operation instructions

The geometric operation instructions perform approximate-value computations. To enable high-speed computation with a minimum of hardware, the ST40 CPU core ignores comparatively small values in the partial computation results of four multiplications. Consequently, the error shown below is produced in the result of the computation:

Maximum error = MAX (individual multiplication result $\times 2^{-\text{MIN (number of multiplier significant digits1, number of multiplicand significant digits1)}}$) + MAX (result value $\times 2^{-23}, 2^{-149}$)

The number of significant digits is 24 for a normalized number and 23 for a denormalized number (number of leading zeros in the fractional part).

In future versions of the ST40 series, the above error bound is guaranteed to be met. However, the actual result value is not guaranteed to be identical to that produced by an earlier ST40 core.

FIPR FVm, FVn (m, n: 0, 4, 8, 12): This instruction may be used for the following (or similar) purposes:

- Inner product (m does not= n)
This operation is generally used for surface/rear surface determination for polygon surfaces.
- Sum of square of elements (m = n)
This operation is generally used to find the length of a vector.
- 4-point multiply accumulate
This is generally used for FIR filtering operations.

Since approximate-value computations are performed to enable high-speed computation, the inexact exception (I) bit in the cause field and flag field is always set to 1 when an FIPR instruction is executed. Therefore, if the corresponding bit is set in the enable field, enable exception handling will be executed.

FTRV XMTRX, FVn (n: 0, 4, 8, 12): This instruction may be used for the following (or similar) purposes:

- Matrix (4 x 4). vector (4)
This operation is generally used for viewpoint changes, angle changes, or movements called vector transformations (4-dimensional). Since affine transformation processing for angle + parallel movement requires a 4 x 4 matrix, the ST40 CPU core supports 4-dimensional operations.
- Matrix (4 x 4) x matrix (4 x 4)
This operation requires the execution of four FTRV instructions.
This operation may also be used in the implementation of optimized transform kernels for digital signal processing and similar applications.

Since approximate-value computations are performed to enable high-speed computation, the inexact exception (I) bit in the cause field and flag field is always set to 1 when an FTRV instruction is executed. Therefore, if the corresponding bit is set in the enable field, enable

exception handling will be executed. For the same reason, it is not possible to check all data types in the registers beforehand when executing an FTRV instruction. If the V bit is set in the enable field, enable exception handling will be executed.

FRCHG: This instruction modifies banked registers. For example, when the FTRV instruction is executed, matrix elements must be set in an array in the background bank. However, to create the actual elements of a translation matrix, it is easier to use registers in the foreground bank. When the LDC instruction is used on FPSCR, this instruction expends 4 to 5 cycles in order to maintain the FPU state. With the FRCHG instruction, an FPSCR.FR bit modification can be performed in one cycle.

FSCA: This floating-point instruction computes the sine and cosine of an angle and returns the results in a pair of single-precision floating-point registers. This is an approximate computation and thus an inexact operation is always signaled. The specified error in the result value is $\pm 2^{-21}$.

FSRRA: This floating-point instruction computes the reciprocal of the square root of a value. Source and result are both in single precision format. This is an approximate computation and thus an inexact operation is always signaled. The specified error in the result value is $\pm 2^{E-21}$, where E = unbiased exponent value of the result.

6.5 64-bit data transfer

The ST40 FPU can support 64-bit data transfers. For register-to-register moves, this allows a pair of registers to be copied to another pair of registers in a single instruction, to support motion of double-precision data. For register-to-memory and memory-to-register moves, this allows 64-bits to be transferred by one instruction.

64-bit data transfer is active whenever FPSCR.SZ=1. When FPSCR.SZ=0, 32-bit data transfer is used. The SZ bit can be changed either:

- with a FSCHG instruction, or
- with a LDS instruction to update the FPSCR

The FSCHG instruction is usually preferred because it executes much more quickly (see [Chapter 10: ST40-100, 200, 400, 500 performance characteristics on page 474](#) and [Chapter 11: ST40-300 performance characteristics on page 497](#)).

6.5.1 Register-to-register transfers

[Table 90](#) shows how the single encoding `1111nnnnmmmm1100` is interpreted depending on the current setting of FPSCR.SZ.

The decoding when FPSCR.SZ=1 limits the 64-bit transfers to operate on DR0, DR2, DR4, DR6, DR8, DR10, DR12, DR14, XD0, XD2, XD4, XD6, XD8, XD10, XD12 and XD14 only. It is not possible to use a 64-bit transfer where the lower register number of the pair is odd.

Table 90. Floating-point register-to-register move instructions

| FPSCR.SZ | Encoding: 1111nnnnmmmm1100 | | | | | | Instruction |
|----------|----------------------------|-----|---|------|---|------|---------------|
| | 1111 | nnn | n | mmmm | m | 1100 | |
| 0 | | FRn | | FRm | | | FMOV FRm, FRn |
| 1 | | DRn | 0 | DRm | 0 | | FMOV DRm, DRn |
| | | DRn | 0 | XDm | 1 | | FMOV XDm, DRn |
| | | XDn | 1 | DRm | 0 | | FMOV DRm, XDn |
| | | XDn | 1 | XDm | 1 | | FMOV XDm, XDn |

6.5.2 Memory transfers

The behavior of transfers between registers and memory is determined by the FPSCR.SZ and FPSCR.PR bits and the endianness under which the ST40 is operating.

[Table 91](#) shows the relationship between the bits in the floating point registers and the bytes in memory for the available modes of operation. The table is applicable for stores as well as loads, if FR_n is replaced by FR_m. In each case, the value VA is the virtual address operand of the load or store operation.

The capability to do true double precision loads and stores (FPSCR.SZ=1 and FPSCR.PR=1 at the same time) is a new feature on the ST40-300 series. In previous ST40 cores, this mode of operation gave undefined behavior.

Table 91. Association of FPU register bit fields with bytes in memory for transfers between registers and memory

| Endianness | FPSCR | | ST40 Series | FPU register state | Memory byte address ⁽¹⁾ | Purpose |
|------------|-------|----|---------------|--|--|---------------------------------------|
| | SZ | PR | | | | |
| Big | 0 | 0 | 100, 200, 300 | FR _n [31:24] FR _n [23:16] FR _n [15:8] FR _n [7:0] | (VA) (VA+1) (VA+2) (VA+3) | Single-precision load or store |
| Big | 1 | 0 | 100, 200, 300 | FR _{2n} [31:24] FR _{2n} [23:16] FR _{2n} [15:8] FR _{2n} [7:0] FR _{2n+1} [31:24] FR _{2n+1} [23:16] FR _{2n+1} [15:8] FR _{2n+1} [7:0] | (VA) (VA+1) (VA+2) (VA+3) (VA+4) (VA+5) (VA+6) (VA+7) | Paired single-precision load or store |
| Big | 1 | 1 | 100, 200 | Undefined behavior | | |

Table 91. Association of FPU register bit fields with bytes in memory for transfers between registers and memory (continued)

| Endianness | FPSCR | | ST40 Series | FPU register state | Memory byte address ⁽¹⁾ | Purpose |
|------------|-------|----|---------------|--|--|---------------------------------------|
| | SZ | PR | | | | |
| Big | 1 | 1 | 300 | FR _{2n} [31:24] FR _{2n} [23:16] FR _{2n} [15:8] FR _{2n} [7:0] FR _{2n+1} [31:24] FR _{2n+1} [23:16] FR _{2n+1} [15:8] FR _{2n+1} [7:0] | (VA) (VA+1) (VA+2) (VA+3) (VA+4) (VA+5) (VA+6) (VA+7) | Double-precision load or store |
| Little | 0 | 0 | 100, 200, 300 | FR _n [31:24] FR _n [23:16] FR _n [15:8] FR _n [7:0] | (VA+3) (VA+2) (VA+1) (VA) | Single-precision load or store |
| Little | 1 | 0 | 100, 200, 300 | FR _{2n} [31:24] FR _{2n} [23:16] FR _{2n} [15:8] FR _{2n} [7:0] FR _{2n+1} [31:24] FR _{2n+1} [23:16] FR _{2n+1} [15:8] FR _{2n+1} [7:0] | (VA+3) (VA+2) (VA+1) (VA) (VA+7) (VA+6) (VA+5) (VA+4) | Paired single-precision load or store |
| Little | 1 | 1 | 100, 200 | Undefined behavior | | |
| Little | 1 | 1 | 300 | FR _{2n} [31:24] FR _{2n} [23:16] FR _{2n} [15:8] FR _{2n} [7:0] FR _{2n+1} [31:24] FR _{2n+1} [23:16] FR _{2n+1} [15:8] FR _{2n+1} [7:0] | (VA+7) (VA+6) (VA+5) (VA+4) (VA+3) (VA+2) (VA+1) (VA) | Double-precision load or store |

1. VA is the virtual address operand of the load or store operation

7 Instruction set

This chapter describes the ST40 instruction set.

7.1 Execution environment

PC

The PC indicates the address of the instruction currently executing.

Data sizes and data types

The ST40 instruction set is implemented with 16-bit fixed-length instructions. The ST40 CPU core can use byte (8-bit), word (16-bit), long-word (32-bit), and quad-word (64-bit) data sizes for memory access. Single-precision floating-point data (32 bits) can be moved to and from memory using long-word or quad-word size. Double-precision floating-point data (64 bits) can be moved to and from memory using long-word size (100, 200 and 300 series) and quad-word size (300 series only). Quad-word data transfers are described in [Section 6.5: 64-bit data transfer on page 169](#).

When the ST40 CPU core moves byte-size or word-size data from memory to a register, the data is sign-extended.

Load-store architecture

The ST40 CPU core features a load-store architecture in which data is read from memory into registers, manipulated within the registers and then written back to memory.

Delayed branches

Except for the two branch instructions BF and BT, the ST40's branch instructions and RTE are delayed branches. In a delayed branch, the instruction following the branch is executed before the branch destination instruction. This execution slot following a delayed branch is called a delay slot. For example, the BRA execution sequence is as follows:

| Static sequence | | Dynamic sequence | | |
|-----------------|--------|------------------|--------|--|
| BRA | TARGET | BRA | TARGET | |
| ADD next_2 | R1, R0 | ADD target_instr | R1, R0 | ADD in delay slot is executed before branching to TARGET |

Delay slot

An illegal instruction exception may occur when a specific instruction is executed in a delay slot, for more details see [Chapter 5: Exceptions on page 146](#). The instruction following BF/S or BT/S for which the branch is not taken is also a delay slot instruction.

T bit

The T bit in the status register (SR) is used to show the result of a compare operation, and is referenced by a conditional branch instruction. An example of the use of a conditional branch instruction is shown below.

ADD #1, R0 T bit is not changed by ADD operation
 CMP/EQ R1, R0 If R0 = R1, T bit is set to 1
 BT TARGET Branches to TARGET if T bit = 1 (R0 = R1)

In an RTE delay slot the Status Register (SR) is updated with the contents of the Saved Status Register (SSR). The ordering of this update with respect to the instruction executing in the delay slot varies as follows. During instruction fetch, the value of the MD bit before the update is used, however, any data access performed uses the MD after the update. The other bits: S, T, M, Q, FD, BL, and RB are all updated prior to execution of the instruction in the delay slot. The STC and STC.L SR instructions read the SR bits after they have been updated.

Constant values

An 8-bit constant value can be specified by the instruction code and an immediate value. 16-bit and 32-bit constant values can be defined as literal constant values in memory, and can be referenced by a PC-relative load instruction.

MOV.W @(disp, PC), Rn
 MOV.L @(disp, PC), Rn

There are no PC-relative load instructions for floating-point operations. However, it is possible to set 0.0 or 1.0 by using the FLDI0 or FLDI1 instruction on a single-precision floating-point register.

7.2 Addressing modes

Addressing modes and virtual address calculation methods are shown in [Table 92](#). When a location in virtual memory space is accessed (MMUCR.AT = 1), the virtual address is translated into a physical memory address. If multiple virtual memory space systems are selected (MMUCR.SV = 0), the least significant bits of PTEH are also referenced as the access ASID. See [Chapter 3: Memory management unit \(MMU\) on page 32](#).

Table 92. Addressing modes and virtual addresses


| Addressing mode | Instruction format | Virtual address calculation method | Calculation formula |
|-------------------|--------------------|--|----------------------------------|
| Register direct | Rn | Virtual address is register Rn. (Operand is register Rn contents.) | — |
| Register indirect | @Rn | Virtual address is register Rn contents.  | Rn → EA (EA: virtual address) |

Table 92. Addressing modes and virtual addresses (continued)

| Addressing mode | Instruction format | Virtual address calculation method | Calculation formula |
|---------------------------------------|--------------------|---|--|
| Register indirect with post-increment | @Rn+ | <p>Virtual address is register Rn contents. A constant is added to Rn after instruction execution: 1 for a byte operand, 2 for a word operand, 4 for a longword operand, 8 for a quadword operand.</p> | <p>Rn →EA After instruction execution Byte: Rn + 1 →Rn Word: Rn + 2 →Rn Longword: Rn + 4 →Rn Quadword: Rn + 8 →Rn</p> |
| Register indirect with pre-decrement | @-Rn | <p>Virtual address is register Rn contents, decremented by a constant beforehand: 1 for a byte operand, 2 for a word operand, 4 for a longword operand, 8 for a quadword operand.</p> | <p>Byte: Rn - 1 →Rn Word: Rn - 2 →Rn Longword: Rn - 4 →Rn Quadword: Rn - 8 →Rn Rn →EA (Instruction executed with Rn after calculation)</p> |
| Register indirect with displacement | @(disp:4, Rn) | <p>Virtual address is register Rn contents with 4-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.</p> | <p>Byte: Rn + disp →EA Word: Rn + disp × 2 →EA Longword: Rn + disp × 4 →EA</p> |
| Indexed register indirect | @(R0, Rn) | <p>Virtual address is sum of register Rn and R0 contents.</p> | Rn + R0 →EA |

Table 92. Addressing modes and virtual addresses (continued)

| Addressing mode | Instruction format | Virtual address calculation method | Calculation formula |
|--------------------------------|--------------------|--|--|
| GBR indirect with displacement | @(disp:8, GBR) | <p>Virtual address is register GBR contents with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 1 (byte), 2 (word), or 4 (longword), according to the operand size.</p> | <p>Byte: $\text{GBR} + \text{disp} \rightarrow \text{EA}$ Word: $\text{GBR} + \text{disp} \times 2 \rightarrow \text{EA}$ Longword: $\text{GBR} + \text{disp} \times 4 \rightarrow \text{EA}$</p> |
| Indexed GBR indirect | @(R0, GBR) | <p>Virtual address is sum of register GBR and R0 contents.</p> | $\text{GBR} + \text{R0} \rightarrow \text{EA}$ |
| PC-relative with displacement | @(disp:8, PC) | <p>Virtual address is PC+4 with 8-bit displacement disp added. After disp is zero-extended, it is multiplied by 2 (word), or 4 (longword), according to the operand size. With a longword operand, the lower 2 bits of PC are masked.</p> <p>* With longword operand</p> | <p>Word: $\text{PC} + 4 + \text{disp} \times 2 \rightarrow \text{EA}$ Long-word: $\text{PC} \wedge 0\text{xFFFFFFFFC} + 4 + \text{disp} \times 4 \rightarrow \text{EA}$</p> |

Table 92. Addressing modes and virtual addresses (continued)

| Addressing mode | Instruction format | Virtual address calculation method | Calculation formula |
|-----------------|--------------------|---|--|
| PC-relative | disp:8 | <p>Virtual address is PC+4 with 8-bit displacement disp added after being sign-extended and multiplied by 2.</p> <pre> graph TD PC[PC] --> A((+)) 4[4] --> A A --> B((+)) disp[disp sign-extended] --> C((x)) 2[2] --> C C --> B B --> Result[PC + 4 + disp * 2] </pre> | $PC + 4 + \text{disp} \times 2 \rightarrow \text{Branch-Target}$ |
| PC-relative | disp:12 | <p>Virtual address is PC+4 with 12-bit displacement disp added after being sign-extended and multiplied by 2.</p> <pre> graph TD PC[PC] --> A((+)) 4[4] --> A A --> B((+)) disp[disp sign-extended] --> C((x)) 2[2] --> C C --> B B --> Result[PC + 4 + disp * 2] </pre> | $PC + 4 + \text{disp} \times 2 \rightarrow \text{Branch-Target}$ |
| | Rn | <p>Virtual address is sum of PC+4 and Rn.</p> <pre> graph TD PC[PC] --> A((+)) 4[4] --> A A --> B((+)) Rn[Rn] --> B B --> Result[PC + 4 + Rn] </pre> | $PC + 4 + Rn \rightarrow \text{Branch-Target}$ |
| Immediate | #imm:8 | 8-bit immediate data imm of TST, AND, OR, or XOR instruction is zero-extended. | — |
| | #imm:8 | 8-bit immediate data imm of MOV, ADD, or CMP/EQ instruction is sign-extended. | — |

Note: For the addressing modes below that use a displacement (*disp*), the assembler descriptions in this manual show the value before scaling ($\times 1$, $\times 2$, or $\times 4$) is performed according to the operand size. This is done to clarify the operation of the chip. Refer to the relevant assembler notation rules for the actual assembler descriptions.

| | |
|-----------------|-------------------------------------|
| @ (disp:4, Rn) | Register indirect with displacement |
| @ (disp:8, GBR) | GBR indirect with displacement |
| @ (disp:8, PC) | PC-relative with displacement |
| disp:8, disp:12 | PC-relative |

7.3 Instruction set summary

[Table 93](#) shows the notation used in the following SH instruction list.

Table 93. Notation used in instruction list

| Item | Format | Description |
|----------------------|-----------------|--|
| Instruction mnemonic | OP.Sz SRC, DEST | OP: Operation code Sz: Size SRC: Source DEST: Source and/or destination operand |
| Summary of operation | | → ← Transfer direction (xx) Memory operand M/Q/T SR flag bits ^ Logical AND of individual bits v Logical OR of individual bits ⊕ Logical exclusive-OR of individual bits ~ Logical NOT of individual bits <<n, >>n n-bit shift |
| Instruction code | MSB ↔ LSB | mmmm: Register number (Rm, FRm) nnnn: Register number (Rn, FRn) 0000: R0, FR0 0001: R1, FR1 : 1111: R15, FR15 mmm: Register number (DRm, XDm, Rm_BANK) nnn: Register number (DRm, XDm, Rn_BANK) 000: DR0, XD0, R0_BANK 001: DR2, XD2, R1_BANK : 111: DR14, XD14, R7_BANK mm: Register number (FVm) nn: Register number (FVn) 00: FV0 01: FV4 10: FV8 11: FV12 iiii: Immediate data dddd: Displacement |

Table 93. Notation used in instruction list (continued)

| Item | Format | Description |
|-----------------|--|---|
| Privileged mode | | “Privileged” means the instruction can only be executed in privileged mode. |
| T bit | Value of T bit after instruction execution | —: No change |

Note: Scaling ($\times 1$, $\times 2$, $\times 4$, or $\times 8$) is executed according to the size of the instruction operand(s).

Table 94. Fixed-point transfer instructions

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|---------------|---|-------------------|------------|-------|
| MOV | #imm,Rn | imm \rightarrow sign extension \rightarrow Rn | 1110nnnniiiiiii | — | — |
| MOV.W | @(disp,PC),Rn | (disp $\times 2$ + PC + 4) \rightarrow sign extension \rightarrow Rn | 1001nnnnnddddddd | — | — |
| MOV.L | @(disp,PC),Rn | (disp $\times 4$ + PC ^ 0xFFFFFC + 4) \rightarrow Rn | 1101nnnnnddddddd | — | — |
| MOV | Rm,Rn | Rm \rightarrow Rn | 0110nnnnmmmm0011 | — | — |
| MOV.B | Rm,@Rn | Rm \rightarrow (Rn) | 0010nnnnmmmm0000 | — | — |
| MOV.W | Rm,@Rn | Rm \rightarrow (Rn) | 0010nnnnmmmm0001 | — | — |
| MOV.L | Rm,@Rn | Rm \rightarrow (Rn) | 0010nnnnmmmm0010 | — | — |
| MOV.B | @Rm,Rn | (Rm) \rightarrow sign extension \rightarrow Rn | 0110nnnnmmmm0000 | — | — |
| MOV.W | @Rm,Rn | (Rm) \rightarrow sign extension \rightarrow Rn | 0110nnnnmmmm0001 | — | — |
| MOV.L | @Rm,Rn | (Rm) \rightarrow Rn | 0110nnnnmmmm0010 | — | — |
| MOV.B | Rm,@-Rn | Rn-1 \rightarrow Rn, Rm \rightarrow (Rn) | 0010nnnnmmmm0100 | — | — |
| MOV.W | Rm,@-Rn | Rn-2 \rightarrow Rn, Rm \rightarrow (Rn) | 0010nnnnmmmm0101 | — | — |
| MOV.L | Rm,@-Rn | Rn-4 \rightarrow Rn, Rm \rightarrow (Rn) | 0010nnnnmmmm0110 | — | — |
| MOV.B | @Rm+,Rn | (Rm) \rightarrow sign extension \rightarrow Rn, Rm + 1 \rightarrow Rm | 0110nnnnmmmm0100 | — | — |
| MOV.W | @Rm+,Rn | (Rm) \rightarrow sign extension \rightarrow Rn, Rm + 2 \rightarrow Rm | 0110nnnnmmmm0101 | — | — |
| MOV.L | @Rm+,Rn | (Rm) \rightarrow Rn, Rm + 4 \rightarrow Rm | 0110nnnnmmmm0110 | — | — |
| MOV.B | R0,@(disp,Rn) | R0 \rightarrow (disp + Rn) | 10000000nnnnndddd | — | — |
| MOV.W | R0,@(disp,Rn) | R0 \rightarrow (disp $\times 2$ + Rn) | 10000001nnnnndddd | — | — |
| MOV.L | Rm,@(disp,Rn) | Rm \rightarrow (disp $\times 4$ + Rn) | 0001nnnnmmmmndddd | — | — |
| MOV.B | @(disp,Rm),R0 | (disp + Rm) \rightarrow sign extension \rightarrow R0 | 10000100mmmmndddd | — | — |
| MOV.W | @(disp,Rm),R0 | (disp $\times 2$ + Rm) \rightarrow sign extension \rightarrow R0 | 10000101mmmmndddd | — | — |
| MOV.L | @(disp,Rm),Rn | (disp $\times 4$ + Rm) \rightarrow Rn | 0101nnnnmmmmndddd | — | — |
| MOV.B | Rm,@(R0,Rn) | Rm \rightarrow (R0 + Rn) | 0000nnnnmmmm0100 | — | — |
| MOV.W | Rm,@(R0,Rn) | Rm \rightarrow (R0 + Rn) | 0000nnnnmmmm0101 | — | — |

Table 94. Fixed-point transfer instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-----------------|---|--------------------|------------|-------|
| MOV.L | Rm, @(R0,Rn) | Rm \rightarrow (R0 + Rn) | 0000nnnnnnmmmm0110 | — | — |
| MOV.B | @(R0,Rm),Rn | (R0 + Rm) \rightarrow sign extension \rightarrow Rn | 0000nnnnnnmmmm1100 | — | — |
| MOV.W | @(R0,Rm),Rn | (R0 + Rm) \rightarrow sign extension \rightarrow Rn | 0000nnnnnnmmmm1101 | — | — |
| MOV.L | @(R0,Rm),Rn | (R0 + Rm) \rightarrow Rn | 0000nnnnnnmmmm1110 | — | — |
| MOV.B | R0, @(disp,GBR) | R0 \rightarrow (disp + GBR) | 11000000ddddddddd | — | — |
| MOV.W | R0, @(disp,GBR) | R0 \rightarrow (disp \times 2 + GBR) | 11000001ddddddddd | — | — |
| MOV.L | R0, @(disp,GBR) | R0 \rightarrow (disp \times 4 + GBR) | 11000010ddddddddd | — | — |
| MOV.B | @(disp,GBR),R0 | (disp + GBR) \rightarrow sign extension \rightarrow R0 | 11000100ddddddddd | — | — |
| MOV.W | @(disp,GBR),R0 | (disp \times 2 + GBR) \rightarrow sign extension \rightarrow R0 | 11000101ddddddddd | — | — |
| MOV.L | @(disp,GBR),R0 | (disp \times 4 + GBR) \rightarrow R0 | 11000110ddddddddd | — | — |
| MOVA | @(disp,PC),R0 | disp \times 4 + PC \wedge 0xFFFFF0 + 4 \rightarrow R0 | 11000111ddddddddd | — | — |
| MOVT | Rn | T \rightarrow Rn | 0000nnnn00101001 | — | — |
| SWAP.B | Rm,Rn | Rm \rightarrow swap lower 2 bytes \rightarrow REG | 0110nnnnnnmmmm1000 | — | — |
| SWAP.W | Rm,Rn | Rm \rightarrow swap upper/lower words \rightarrow Rn | 0110nnnnnnmmmm1001 | — | — |
| XTRCT | Rm,Rn | Rm:Rn middle 32 bits \rightarrow Rn | 0010nnnnnnmmmm1101 | — | — |

Table 95. Arithmetic operation instructions

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|---------|---|--------------------|------------|-------------------|
| ADD | Rm,Rn | Rn + Rm \rightarrow Rn | 0011nnnnnnmmmm1100 | — | — |
| ADD | #imm,Rn | Rn + imm \rightarrow Rn | 0111nnnniiiiiiii | — | — |
| ADDC | Rm,Rn | Rn + Rm + T \rightarrow Rn, carry \rightarrow T | 0011nnnnnnmmmm1110 | — | Carry |
| ADDV | Rm,Rn | Rn + Rm \rightarrow Rn, overflow \rightarrow T | 0011nnnnnnmmmm1111 | — | Overflow |
| CMP/EQ | #imm,R0 | When R0 = imm, 1 \rightarrow T Otherwise, 0 \rightarrow T | 10001000iiiiiiii | — | Comparison result |
| CMP/EQ | Rm,Rn | When Rn = Rm, 1 \rightarrow T Otherwise, 0 \rightarrow T | 0011nnnnnnmmmm0000 | — | Comparison result |
| CMP/HS | Rm,Rn | When Rn \geq Rm (unsigned), 1 \rightarrow T Otherwise, 0 \rightarrow T | 0011nnnnnnmmmm0010 | — | Comparison result |
| CMP/GE | Rm,Rn | When Rn \geq Rm (signed), 1 \rightarrow T Otherwise, 0 \rightarrow T | 0011nnnnnnmmmm0011 | — | Comparison result |

Table 95. Arithmetic operation instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|------------|---|--------------------|------------|-----------------------|
| CMP/Hi | Rm,Rn | When $Rn > Rm$ (unsigned), 1 \rightarrow T Otherwise, 0 \rightarrow T | 0011nnnnnnnnnn0110 | — | Comparison result |
| CMP/GT | Rm,Rn | When $Rn > Rm$ (signed), 1 \rightarrow T Otherwise, 0 \rightarrow T | 0011nnnnnnnnnn0111 | — | Comparison result |
| CMP/PZ | Rn | When $Rn \geq 0$, 1 \rightarrow T Otherwise, 0 \rightarrow T | 0100nnnn00010001 | — | Comparison result |
| CMP/PL | Rn | When $Rn > 0$, 1 \rightarrow T Otherwise, 0 \rightarrow T | 0100nnnn00010101 | — | Comparison result |
| CMP/STR | Rm,Rn | When any bytes are equal, 1 \rightarrow T Otherwise, 0 \rightarrow T | 0010nnnnnnnnnn1100 | — | Comparison result |
| DIV1 | Rm,Rn | 1-step division ($Rn \div Rm$) | 0011nnnnnnnnnn0100 | — | Calculation result |
| DIV0S | Rm,Rn | MSB of $Rn \rightarrow Q$, MSB of $Rm \rightarrow M$, $M \oplus Q \rightarrow T$ | 0010nnnnnnnnnn0111 | — | Calculation result |
| DIV0U | | 0 $\rightarrow M/Q/T$ | 0000000000011001 | — | 0 |
| DMULS.L | Rm,Rn | Signed, $Rn \times Rm \rightarrow MAC$, $32 \times 32 \rightarrow 64$ bits | 0011nnnnnnnnnn1101 | — | — |
| DMULU.L | Rm,Rn | Unsigned, $Rn \times Rm \rightarrow MAC$, $32 \times 32 \rightarrow 64$ bits | 0011nnnnnnnnnn0101 | — | — |
| DT | Rn | $Rn - 1 \rightarrow Rn$; when $Rn = 0$, 1 \rightarrow T When $Rn \neq 0$, 0 \rightarrow T | 0100nnnn00010000 | — | Comparison result |
| EXTS.B | Rm,Rn | Rm sign-extended from byte $\rightarrow Rn$ | 0110nnnnnnnnnn1110 | — | — |
| EXTS.W | Rm,Rn | Rm sign-extended from word $\rightarrow Rn$ | 0110nnnnnnnnnn1111 | — | — |
| EXTU.B | Rm,Rn | Rm zero-extended from byte $\rightarrow Rn$ | 0110nnnnnnnnnn1100 | — | — |
| EXTU.W | Rm,Rn | Rm zero-extended from word $\rightarrow Rn$ | 0110nnnnnnnnnn1101 | — | — |
| MAC.L | @Rm+, @Rn+ | Signed, $(Rn) \times (Rm) + MAC \rightarrow$ MAC $Rn + 4 \rightarrow Rn$, $Rm + 4 \rightarrow Rm$ $32 \times 32 + 64 \rightarrow 64$ bits | 0000nnnnnnnnnn1111 | — | — |
| MAC.W | @Rm+, @Rn+ | Signed, $(Rn) \times (Rm) + MAC \rightarrow$ MAC $Rn + 2 \rightarrow Rn$, $Rm + 2 \rightarrow Rm$ $16 \times 16 + 64 \rightarrow 64$ bits | 0100nnnnnnnnnn1111 | — | — |
| MUL.L | Rm,Rn | $Rn \times Rm \rightarrow MACL$ $32 \times 32 \rightarrow 32$ bits | 0000nnnnnnnnnn0111 | — | — |
| MULS.W | Rm,Rn | Signed, $Rn \times Rm \rightarrow MACL$ $16 \times 16 \rightarrow 32$ bits | 0010nnnnnnnnnn1111 | — | — |

Table 95. Arithmetic operation instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-------|--|--------------------|------------|-----------|
| MULU.W | Rm,Rn | Unsigned, $Rn \times Rm \rightarrow \text{MACL}$ $16 \times 16 \rightarrow 32$ bits | 0010nnnnnnmmmm1110 | — | — |
| NEG | Rm,Rn | $0 - Rm \rightarrow Rn$ | 0110nnnnnnmmmm1011 | — | — |
| NEGC | Rm,Rn | $0 - Rm - T \rightarrow Rn$, borrow $\rightarrow T$ | 0110nnnnnnmmmm1010 | — | Borrow |
| SUB | Rm,Rn | $Rn - Rm \rightarrow Rn$ | 0011nnnnnnmmmm1000 | — | — |
| SUBC | Rm,Rn | $Rn - Rm - T \rightarrow Rn$, borrow $\rightarrow T$ | 0011nnnnnnmmmm1010 | — | Borrow |
| SUBV | Rm,Rn | $Rn - Rm \rightarrow Rn$, underflow $\rightarrow T$ | 0011nnnnnnmmmm1011 | — | Underflow |

Table 96. Logic operation instructions

| Instruction | | Operation | Instruction code | Privileged | T Bit |
|-------------|----------------|---|--------------------|------------|-------------|
| AND | Rm,Rn | $Rn \wedge Rm \rightarrow Rn$ | 0010nnnnnnmmmm1001 | — | — |
| AND | #imm,R0 | $R0 \wedge \text{imm} \rightarrow R0$ | 11001001iiiiiiii | — | — |
| AND.B | #imm,@(R0,GBR) | $(R0 + \text{GBR}) \wedge \text{imm} \rightarrow (R0 + \text{GBR})$ | 11001101iiiiiiii | — | — |
| NOT | Rm,Rn | $\sim Rm \rightarrow Rn$ | 0110nnnnnnmmmm0111 | — | — |
| OR | Rm,Rn | $Rn \vee Rm \rightarrow Rn$ | 0010nnnnnnmmmm1011 | — | — |
| OR | #imm,R0 | $R0 \vee \text{imm} \rightarrow R0$ | 11001011iiiiiiii | — | — |
| OR.B | #imm,@(R0,GBR) | $(R0 + \text{GBR}) \vee \text{imm} \rightarrow (R0 + \text{GBR})$ | 11001111iiiiiiii | — | — |
| TAS.B | @Rn | When $(Rn) = 0$, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ In both cases, $1 \rightarrow \text{MSB of } (Rn)$ | 0100nnnn00011011 | — | Test result |
| TST | Rm,Rn | $Rn \wedge Rm$; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 0010nnnnnnmmmm1000 | — | Test result |
| TST | #imm,R0 | $R0 \wedge \text{imm}$; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 11001000iiiiiiii | — | Test result |
| TST.B | #imm,@(R0,GBR) | $(R0 + \text{GBR}) \wedge \text{imm}$; when result = 0, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 11001100iiiiiiii | — | Test result |
| XOR | Rm,Rn | $Rn \oplus Rm \rightarrow Rn$ | 0010nnnnnnmmmm1010 | — | — |
| XOR | #imm,R0 | $R0 \oplus \text{imm} \rightarrow R0$ | 11001010iiiiiiii | — | — |
| XOR.B | #imm,@(R0,GBR) | $(R0 + \text{GBR}) \oplus \text{imm} \rightarrow (R0 + \text{GBR})$ | 11001110iiiiiiii | — | — |

Table 97. Shift instructions

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-------|--|--------------------|------------|-------|
| ROTL | Rn | $T \leftarrow Rn \leftarrow \text{MSB}$ | 0100nnnn00000100 | — | MSB |
| ROTR | Rn | $\text{LSB} \rightarrow Rn \rightarrow T$ | 0100nnnn00000101 | — | LSB |
| ROTCL | Rn | $T \leftarrow Rn \leftarrow T$ | 0100nnnn00100100 | — | MSB |
| ROTCR | Rn | $T \rightarrow Rn \rightarrow T$ | 0100nnnn00100101 | — | LSB |
| SHAD | Rm,Rn | When $Rn \geq 0$, $Rn \ll Rm \rightarrow Rn$ When $Rn < 0$, $Rn \gg Rm \rightarrow [\text{MSB} \rightarrow Rn]$ | 0100nnnnnnnnnn1100 | — | — |
| SHAL | Rn | $T \leftarrow Rn \leftarrow 0$ | 0100nnnn00100000 | — | MSB |
| SHAR | Rn | $\text{MSB} \rightarrow Rn \rightarrow T$ | 0100nnnn00100001 | — | LSB |
| SHLD | Rm,Rn | When $Rn \geq 0$, $Rn \ll Rm \rightarrow Rn$ When $Rn < 0$, $Rn \gg Rm \rightarrow [0 \rightarrow Rn]$ | 0100nnnnnnnnnn1101 | — | — |
| SHLL | Rn | $T \leftarrow Rn \leftarrow 0$ | 0100nnnn00000000 | — | MSB |
| SHLR | Rn | $0 \rightarrow Rn \rightarrow T$ | 0100nnnn00000001 | — | LSB |
| SHLL2 | Rn | $Rn \ll 2 \rightarrow Rn$ | 0100nnnn00001000 | — | — |
| SHLR2 | Rn | $Rn \gg 2 \rightarrow Rn$ | 0100nnnn00001001 | — | — |
| SHLL8 | Rn | $Rn \ll 8 \rightarrow Rn$ | 0100nnnn00011000 | — | — |
| SHLR8 | Rn | $Rn \gg 8 \rightarrow Rn$ | 0100nnnn00011001 | — | — |
| SHLL16 | Rn | $Rn \ll 16 \rightarrow Rn$ | 0100nnnn00101000 | — | — |
| SHLR16 | Rn | $Rn \gg 16 \rightarrow Rn$ | 0100nnnn00101001 | — | — |

Table 98. Branch instructions

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-------|---|-------------------|------------|-------|
| BF | label | When $T = 0$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ When $T = 1$, nop | 10001011ddddddddd | — | — |
| BF/S | label | Delayed branch; when $T = 0$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ When $T = 1$, nop | 10001111ddddddddd | — | — |
| BT | label | When $T = 1$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ When $T = 0$, nop | 10001001ddddddddd | — | — |
| BT/S | label | Delayed branch; when $T = 1$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ When $T = 0$, nop | 10001101ddddddddd | — | — |
| BRA | label | Delayed branch, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ | 1010ddddddddddddd | — | — |
| BRAF | Rn | $Rn + \text{PC} + 4 \rightarrow \text{PC}$ | 0000nnnn00100011 | — | — |
| BSR | label | Delayed branch, $\text{PC} + 4 \rightarrow \text{PR}$, $\text{disp} \times 2 + \text{PC} + 4 \rightarrow \text{PC}$ | 1011ddddddddddddd | — | — |

Table 98. Branch instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-----|---|------------------|------------|-------|
| BSRF | Rn | Delayed branch, PC + 4 \rightarrow PR, Rn + PC + 4 \rightarrow PC | 0000nnnn00000011 | — | — |
| JMP | @Rn | Delayed branch, Rn \rightarrow PC | 0100nnnn00101011 | — | — |
| JSR | @Rn | Delayed branch, PC + 4 \rightarrow PR, Rn \rightarrow PC | 0100nnnn00001011 | — | — |
| RTS | | Delayed branch, PR \rightarrow PC | 0000000000001011 | — | — |

In [Table 99](#), the *Series* column indicates which ST40 series the instruction applies to. If the entry in this column is blank, it means that the instruction is available on all ST40 variants.

Table 99. System control instructions

| Instruction | | Operation | Instruction code | Privileged | T bit | Series |
|-------------|------------|--|------------------|------------|-------|--------|
| CLRMACH | | 0 \rightarrow MACH, MACL | 0000000000101000 | — | — | |
| CLRS | | 0 \rightarrow S | 0000000001001000 | — | — | |
| CLRT | | 0 \rightarrow T | 0000000000001000 | — | 0 | |
| ICBI | @Rn | Invalidates instruction cache block and refetches next instruction | 0000nnnn11100011 | — | — | 300 |
| LDC | Rm,SR | Rm \rightarrow SR | 0100mmmm00001110 | Privileged | LSB | |
| LDC | Rm,GBR | Rm \rightarrow GBR | 0100mmmm00011110 | — | — | |
| LDC | Rm,VBR | Rm \rightarrow VBR | 0100mmmm00101110 | Privileged | — | |
| LDC | Rm,SSR | Rm \rightarrow SSR | 0100mmmm00111110 | Privileged | — | |
| LDC | Rm,SPC | Rm \rightarrow SPC | 0100mmmm01001110 | Privileged | — | |
| LDC | Rm,SGR | Rm \rightarrow SGR | 0100mmmm00111010 | Privileged | — | |
| LDC | Rm,DBR | Rm \rightarrow DBR | 0100mmmm11111010 | Privileged | — | |
| LDC | Rm,Rn_BANK | Rm \rightarrow Rn_BANK (n = 0 to 7) | 0100mmmm1nnn1110 | Privileged | — | |
| LDC.L | @Rm+,SR | (Rm) \rightarrow SR, Rm + 4 \rightarrow Rm | 0100mmmm00000111 | Privileged | LSB | |
| LDC.L | @Rm+,GBR | (Rm) \rightarrow GBR, Rm + 4 \rightarrow Rm | 0100mmmm00010111 | — | — | |
| LDC.L | @Rm+,VBR | (Rm) \rightarrow VBR, Rm + 4 \rightarrow Rm | 0100mmmm00100111 | Privileged | — | |
| LDC.L | @Rm+,SSR | (Rm) \rightarrow SSR, Rm + 4 \rightarrow Rm | 0100mmmm00110111 | Privileged | — | |
| LDC.L | @Rm+,SPC | (Rm) \rightarrow SPC, Rm + 4 \rightarrow Rm | 0100mmmm01000111 | Privileged | — | |
| LDC.L | @Rm+,SGR | (Rm) \rightarrow SGR, Rm + 4 \rightarrow Rm | 0100mmmm00110110 | Privileged | — | |

Table 99. System control instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit | Series |
|-------------|--------------|--|-------------------|------------|-------|--------|
| LDC.L | @Rm+,DBR | (Rm) →DBR, Rm + 4 →Rm | 0100mmmm11110110 | Privileged | — | |
| LDC.L | @Rm+,Rn_BANK | (Rm) →Rn_BANK, Rm + 4 →Rm | 0100mmmm1nnn0111 | Privileged | — | |
| LDS | Rm,MACH | Rm →MACH | 0100mmmm00001010 | — | — | |
| LDS | Rm,MACL | Rm →MACL | 0100mmmm00011010 | — | — | |
| LDS | Rm,PR | Rm →PR | 0100mmmm00101010 | — | — | |
| LDS.L | @Rm+,MACH | (Rm) →MACH, Rm + 4 →Rm | 0100mmmm00000110 | — | — | |
| LDS.L | @Rm+,MACL | (Rm) →MACL, Rm + 4 →Rm | 0100mmmm00010110 | — | — | |
| LDS.L | @Rm+,PR | (Rm) →PR, Rm + 4 →Rm | 0100mmmm00100110 | — | — | |
| LDTLB | | PTEH/PTEL →TLB | 000000000111000 | Privileged | — | |
| MOVCA.L | R0,@Rn | R0 →(Rn) (without fetching cache block) | 0000nnnn11000011 | — | — | |
| NOP | | No operation | 0000000000001001 | — | — | |
| OCBI | @Rn | Invalidates operand cache block | 0000nnnn10010011 | — | — | |
| OCBP | @Rn | Writes back and invalidates operand cache block | 0000nnnn10100011 | — | — | |
| OCBWB | @Rn | Writes back operand cache block | 0000nnnn10110011 | — | — | |
| PREF | @Rn | (Rn) →operand cache | 0000nnnn10000011 | — | — | |
| RTE | | Delayed branch, SSR/SPC →SR/PC | 000000000101011 | Privileged | — | |
| SETS | | 1 →S | 0000000001011000 | — | — | |
| SETT | | 1 →T | 0000000000011000 | — | 1 | |
| SLEEP | | Sleep or standby | 0000000000011011 | Privileged | — | |
| STC | SR,Rn | SR →Rn | 0000nnnn00000010 | Privileged | — | |
| STC | GBR,Rn | GBR →Rn | 0000nnnn00010010 | — | — | |
| STC | VBR,Rn | VBR →Rn | 0000nnnn00100010 | Privileged | — | |
| STC | SSR,Rn | SSR →Rn | 0000nnnn00110010 | Privileged | — | |
| STC | SPC,Rn | SPC →Rn | 0000nnnn01000010 | Privileged | — | |
| STC | SGR,Rn | SGR →Rn | 0000nnnn00111010 | Privileged | — | |
| STC | DBR,Rn | DBR →Rn | 0000nnnn11111010 | Privileged | — | |
| STC | Rm_BANK,Rn | Rm_BANK →Rn (m = 0 to 7) | 0000nnnn1mmmm0010 | Privileged | — | |

Table 99. System control instructions (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit | Series |
|-------------|---------------|--|-------------------|------------|-------|--------|
| STC.L | SR, @-Rn | Rn - 4 → Rn, SR →(Rn) | 0100nnnn00000011 | Privileged | — | |
| STC.L | GBR, @-Rn | Rn - 4 → Rn, GBR →(Rn) | 0100nnnn00010011 | — | — | |
| STC.L | VBR, @-Rn | Rn - 4 → Rn, VBR →(Rn) | 0100nnnn00100011 | Privileged | — | |
| STC.L | SSR, @-Rn | Rn - 4 → Rn, SSR →(Rn) | 0100nnnn00110011 | Privileged | — | |
| STC.L | SPC, @-Rn | Rn - 4 → Rn, SPC →(Rn) | 0100nnnn01000011 | Privileged | — | |
| STC.L | SGR, @-Rn | Rn - 4 → Rn, SGR →(Rn) | 0100nnnn00110010 | Privileged | — | |
| STC.L | DBR, @-Rn | Rn - 4 → Rn, DBR →(Rn) | 0100nnnn11110010 | Privileged | — | |
| STC.L | Rm_BANK, @-Rn | Rn - 4 → Rn, Rm_BANK →(Rn) (m = 0 to 7) | 0100nnnn1mmmm0011 | Privileged | — | |
| STS | MACH, Rn | MACH → Rn | 0000nnnn00001010 | — | — | |
| STS | MACL, Rn | MACL → Rn | 0000nnnn00011010 | — | — | |
| STS | PR, Rn | PR → Rn | 0000nnnn00101010 | — | — | |
| STS.L | MACH, @-Rn | Rn - 4 → Rn, MACH →(Rn) | 0100nnnn00000010 | — | — | |
| STS.L | MACL, @-Rn | Rn - 4 → Rn, MACL →(Rn) | 0100nnnn00010010 | — | — | |
| STS.L | PR, @-Rn | Rn - 4 → Rn, PR →(Rn) | 0100nnnn00100010 | — | — | |
| SYNCO | | Memory barrier for operand accesses | 0000000010101011 | — | — | 300 |
| TRAPA | #imm | PC + 2 → SPC, SR → SSR, #imm << 2 → TRA, 0x160 → EXPEVT, VBR + 0x0100 → PC | 11000011iiiiiii | — | — | |

Table 100. Floating-point single-precision arithmetic instructions (only if FPU present and FPSCR.PR=0)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|---------------|--|------------------|------------|-------------------|
| FLDI0 | FRn | $0x00000000 \rightarrow FRn$ | 1111nnnn10001101 | — | — |
| FLDI1 | FRn | $0x3F800000 \rightarrow FRn$ | 1111nnnn10011101 | — | — |
| FABS | FRn | $FRn \wedge 0x7FFF\ FFFF \rightarrow FRn$ | 1111nnnn01011101 | — | — |
| FADD | FRm, FRn | $FRn + FRm \rightarrow FRn$ | 1111nnnnmmmm0000 | — | — |
| FCMP/EQ | FRm, FRn | When $FRn = FRm$, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 1111nnnnmmmm0100 | — | Comparison result |
| FCMP/GT | FRm, FRn | When $FRn > FRm$, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 1111nnnnmmmm0101 | — | Comparison result |
| FDIV | FRm, FRn | $FRn / FRm \rightarrow FRn$ | 1111nnnnmmmm0011 | — | — |
| FIPR | FVm, FVn | inner_product [FVm, FVn] $\rightarrow FR[n+3]$ | 1111nnmm11101101 | — | — |
| FLOAT | FPUL, FRn | (float) FPUL $\rightarrow FRn$ | 1111nnnn00101101 | — | — |
| FMAC | FR0, FRm, FRn | $FR0 * FRm + FRn \rightarrow FRn$ | 1111nnnnmmmm1110 | — | — |
| FMUL | FRm, FRn | $FRn * FRm \rightarrow FRn$ | 1111nnnnmmmm0010 | — | — |
| FNEG | FRn | $FRn \oplus 0x80000000 \rightarrow FRn$ | 1111nnnn01001101 | — | — |
| FSCA | FPUL, DRn | FSINA_S [FPUL] $\rightarrow FRn$, FCOSA_S [FPUL] $\rightarrow FRn+1$ | 1111nnnn01111101 | — | — |
| FSQRT | FRn | $\sqrt{FRn} \rightarrow FRn$ | 1111nnnn01101101 | — | — |
| FSRRA | FRn | FSRRA_S [FRn] $\rightarrow FRn$ | 1111nnnn01111101 | — | — |
| FSUB | FRm, FRn | $FRn - FRm \rightarrow FRn$ | 1111nnnnmmmm0001 | — | — |
| FTRC | FRm, FPUL | (long) FRm $\rightarrow FPUL$ | 1111mmmm00111101 | — | — |
| FTRV | XMTRX, FVn | transform_vector [XMTRX, FVn] $\rightarrow FVn$ | 1111nn0111111101 | — | — |

Table 101. Floating-point double-precision arithmetic instructions (only if FPU present and FPSCR.PR=1)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|-----------|--|--------------------|------------|-------------------|
| FABS | DRn | $0x7FFF\ FFFF\ FFFF\ FFFF \wedge DRn \rightarrow DRn$ | 1111nnnn001011101 | — | — |
| FADD | DRm, DRn | $DRn + DRm \rightarrow DRn$ | 1111nnnn0mmmm00000 | — | — |
| FCMP/EQ | DRm, DRn | When $DRn = DRm$, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 1111nnnn0mmmm00100 | — | Comparison result |
| FCMP/GT | DRm, DRn | When $DRn > DRm$, $1 \rightarrow T$ Otherwise, $0 \rightarrow T$ | 1111nnnn0mmmm00101 | — | Comparison result |
| FDIV | DRm, DRn | $DRn / DRm \rightarrow DRn$ | 1111nnnn0mmmm00011 | — | — |
| FCNVDS | DRm, FPUL | double_to_float [DRm] $\rightarrow FPUL$ | 1111mmmm010111101 | — | — |
| FCNVSD | FPUL, DRn | float_to_double [FPUL] $\rightarrow DRn$ | 1111nnnn010101101 | — | — |

Table 101. Floating-point double-precision arithmetic instructions (only if FPU present and FPSCR.PR=1) (continued)

| Instruction | | Operation | Instruction code | Privileged | T bit |
|-------------|----------|----------------------------------|------------------|------------|-------|
| FLOAT | FPUL,DRn | (float)FPUL →DRn | 1111nnn000101101 | — | — |
| FMUL | DRm,DRn | DRn * DRm →DRn | 1111nnn0mmm00010 | — | — |
| FNEG | DRn | DRn ⊕ 0x8000 0000 0000 0000 →DRn | 1111nnn001001101 | — | — |
| FSQRT | DRn | √DRn →DRn | 1111nnn001101101 | — | — |
| FSUB | DRm,DRn | DRn – DRm →DRn | 1111nnn0mmm00001 | — | — |
| FTRC | DRm,FPUL | (long) DRm →FPUL | 1111mmm000111101 | — | — |

[Table 102](#) shows the instruction set for moving floating point values between registers and between registers and memory. None of these instructions are privileged and none affect the T-bit.

Table 102. Floating-point data transfer instructions (only if FPU present)

| Instruction | | Operation | Instruction code | FPSCR.SZ |
|-------------|--------------|--------------------------|------------------|----------|
| FMOV | FRm,FRn | FRm →FRn | 1111nnnnnnmm1100 | 0 |
| FMOV.S | @Rm,FRn | (Rm) →FRn | 1111nnnnnnmm1000 | 0 |
| FMOV.S | @(R0,Rm),FRn | (R0 + Rm) →FRn | 1111nnnnnnmm0110 | 0 |
| FMOV.S | @Rm+,FRn | (Rm) →FRn, Rm + 4 →Rm | 1111nnnnnnmm1001 | 0 |
| FMOV.S | FRm,@Rn | FRm →Rn | 1111nnnnnnmm1010 | 0 |
| FMOV.S | FRm,@-Rn | Rn-4 →Rn, FRm →Rn | 1111nnnnnnmm1011 | 0 |
| FMOV.S | FRm,@(R0,Rn) | FRm →(R0 + Rn) | 1111nnnnnnmm0111 | 0 |
| FMOV | @Rm,DRn | (Rm) →DRn | 1111nnn0mmm1000 | 1 |
| FMOV | @(R0,Rm),DRn | (R0 + Rm) →DRn | 1111nnn0mmm0110 | 1 |
| FMOV | @Rm+,DRn | (Rm) →DRn, Rm + 8 →Rm | 1111nnn0mmm1001 | 1 |
| FMOV | DRm,@Rn | DRm →Rn | 1111nnnnmm01010 | 1 |
| FMOV | DRm,@-Rn | Rn-8 →Rn, DRm →Rn | 1111nnnnmm01011 | 1 |
| FMOV | DRm,@(R0,Rn) | DRm →(R0 + Rn) | 1111nnnnmm00111 | 1 |
| FLDS | FRm,FPUL | FRm →FPUL | 1111mmmm00011101 | any |
| FSTS | FPUL,FRn | FPUL →FRn | 1111nnnn00001101 | any |
| FMOV | DRm,DRn | DRm →DRn | 1111nnn0mmm01100 | 1 |
| FMOV | DRm,XDn | DRm →XDn | 1111nnn1mmm01100 | 1 |
| FMOV | XDm,DRn | XDm →DRn | 1111nnn0mmm11100 | 1 |
| FMOV | XDm,XDn | XDm →XDn | 1111nnn1mmm11100 | 1 |
| FMOV | @Rm,XDn | (Rm) →XDn | 1111nnn1mmmm1000 | 1 |

Table 102. Floating-point data transfer instructions (only if FPU present) (continued)

| Instruction | | Operation | Instruction code | FPSCR.SZ |
|-------------|--------------|--------------------------|-------------------|----------|
| FMOV | @Rm+,XDn | (Rm) →XDn, Rm + 8 →Rm | 1111nnn1mmmm1001 | 1 |
| FMOV | @(R0,Rm),XDn | (R0 + Rm) →XDn | 1111nnn1mmmm0110 | 1 |
| FMOV | XDm,@Rn | XDm →(Rn) | 1111nnnnmmmm11010 | 1 |
| FMOV | XDm,@-Rn | Rn - 8 →Rn, XDm →(Rn) | 1111nnnnmmmm11011 | 1 |
| FMOV | XDm,@(R0,Rn) | XDm →(R0+Rn) | 1111nnnnmmmm10111 | 1 |

None of the instructions in [Table 103](#) are privileged and none affect the T-bit. The FPCHG instruction is only available on the ST40-300 series. The others are available on all series that have an FPU.

Table 103. Floating-point control instructions (only if FPU present)

| Instruction | | Operation | Instruction code | Series |
|-------------|------------|-------------------------|------------------|--------|
| FPCHG | | ~FPSCR.PR →FPSCR.PR | 1111011111111101 | 300 |
| FRCHG | | ~FPSCR.FR →FPSCR.FR | 1111101111111101 | — |
| FSCHG | | ~FPSCR.SZ →FPSCR.SZ | 1111001111111101 | — |
| LDS | Rm,FPSCR | Rm →FPSCR | 0100mmmm01101010 | — |
| LDS | Rm,FPUL | Rm →FPUL | 0100mmmm01011010 | — |
| LDS.L | @Rm+,FPSCR | (Rm) →FPSCR, Rm+4 →Rm | 0100mmmm01100110 | — |
| LDS.L | @Rm+,FPUL | (Rm) →FPUL, Rm+4 →Rm | 0100mmmm01010110 | — |
| STS | FPSCR,Rn | FPSCR →Rn | 0000nnnn01101010 | — |
| STS | FPUL,Rn | FPUL →Rn | 0000nnnn01011010 | — |
| STS.L | FPSCR,@-Rn | Rn - 4 →Rn, FPSCR →(Rn) | 0100nnnn01100010 | — |
| STS.L | FPUL,@-Rn | Rn - 4 →Rn, FPUL →(Rn) | 0100nnnn01010010 | — |

8 Instruction specification

The behavior of instructions is specified using a simple notational language to describe the effects of each instruction on the architectural state of the machine.

The language consists of the following features:

- a simple variable and type system
- expressions
- statements
- notation for the architectural state of the machine
- an abstract sequential model of instruction execution

These features are described in the following sections. Additional mechanisms are defined to model memory, synchronization instructions, cache instructions and floating-point. The final section gives example instruction specifications.

Each instruction is described using informal text as well as the formal notational language. Sometimes it is inappropriate for one of these descriptions to convey the full semantics. In such cases these two descriptions must be taken together to constitute the full specification.

8.1 Variables and types

Variables are used to hold state. The type of a variable determines the set of values that the variable can take and the available operators to manipulate that variable. The supported scalar types are integers, booleans and bit-fields. One-dimensional arrays of the scalar types are also supported.

The architectural state of the machine is represented by a set of variables. Each of these variables has an associated type, which is either a bit-field or an array of bit-fields. Bit-fields are used to give a bit-accurate representation.

Additional variables are used to hold temporary values. The type of temporary variables is implicit, and determined by their context rather than explicit declaration. The type of a temporary variable is an integer, a boolean or an array of these.

8.1.1 Integer

An integer variable can take the value of any mathematical integer. No limits are imposed on the range of integers supported. Integers obey their standard mathematical properties. Integer operations do not overflow. The integer operators are defined so that singularities do not occur. For example, no definition is given to the result of divide by zero; the operator is simply not available when the divisor is zero.

The representation of literal integer values is achieved using the following notations:

- decimal numbers are represented by the regular expression: `{0-9}+`
- hexadecimal numbers are represented by the regular expression: `0x{0-9a-fA-F}+`
- binary numbers are represented by the regular expression: `0b{0-1}+`

These notations are standard and map onto integer values in the obvious way. Underscore characters ('_') can be inserted into any of the above literal representations. These do not change the represented value but can be used as spacers to aid readability.

The notations allow only zero and positive numbers to be represented directly. A monadic integer negation operator can subsequently be used to derive a negative value.

8.1.2 Boolean

A boolean variable can take two values:

- boolean false: the literal representation of boolean false is 'FALSE'
- boolean true: the literal representation of boolean true is 'TRUE'

8.1.3 Bit-fields

Bit-fields are provided to define 'bit-accurate' storage.

Bit-fields containing arbitrary numbers of bits are supported. A bit-field of b bits contains bits numbered from 0 (the least significant bit) up to $b-1$ (the most significant bit). Each bit can take the value 0 or the value 1. Bit-fields are mapped to, and from, integers in the usual way. If bit i of a b -bit, bit-field, where i is in $[0, b)$, is set then it contributes 2^i to the integral value of the bit-field. The integral value of the bit-field as a whole is an integer in the range $[0, 2^b)$.

When a bit-field is read, it gives its integral value. When a bit-field is written with an integral value, the integer must be in the range of values supported by the bit-field. Typically, the only operations applied directly to bit-fields are conversions to other types.

8.1.4 Arrays

One-dimensional arrays of the above types are also available. Indexing into an n -element array A is achieved using the notation $A[i]$ where A is an array of some type and i is an integer in the range $[0, n)$. This selects the i^{th} element of the array A . If i is zero this selects the first entry, and if i is $n-1$ then this selects the last entry. The type of the selected element is the base type of the array.

Multi-dimensional arrays are not provided.

8.1.5 Floating point values

Floating-point types and operators are not provided. Instead, the value in a floating-point register is represented as a bit-field. The organization of the bit-field is consistent with an IEEE754 format.

When a floating-point register is read, an integral representation of that bit-pattern is returned. When an integral value is written into a floating-point register, the value written is the bit-pattern of that integer. Thus, reading and writing is achieved as bit-pattern transfers, and not by interpreting the bit-patterns as real numbers.

The language does not provide direct means to interpret these bit-patterns as real numbers. Instead, functions are provided which give the required functionality. For example, arithmetic on real numbers is represented using a function notation.

8.2 Expressions

Expressions are constructed from monadic operators, dyadic operators and functions applied to variable and literal values.

There are no defined precedence and associativity rules for the operators. Parentheses are used to specify the expression unambiguously.

Sub-expressions can be evaluated in any order. If a particular evaluation order is required, then sub-expressions must be split into separate statements.

8.2.1 Integer arithmetic operators

Since the notation uses straightforward mathematical integers, the set of standard mathematical operators is available and already defined.

The standard dyadic operators are listed in [Table 104](#).

Table 104. Standard dyadic operators

| Operation | Description |
|-----------------|------------------------|
| $i + j$ | Integer addition |
| $i - j$ | Integer subtraction |
| $i \times j$ | Integer multiplication |
| i / j | Integer division |
| $i \setminus j$ | Integer remainder |

The standard monadic operators are described in [Table 105](#).

Table 105. Standard monadic operators

| Operator | Description |
|----------|------------------|
| $- i$ | Integer negation |
| $ i $ | Integer modulus |

The division operator truncates towards zero. The remainder operator is consistent with this. The sign of the result of the remainder operator follows the sign of the dividend. Division or remainder with a divisor of zero results in a singularity, and its behavior is not defined.

For a numerator (n) and a denominator (d), the following properties hold where $d \neq 0$:

$$\begin{aligned}
 n &= d \times (n / d) + (n \setminus d) \\
 (-n) / d &= -(n / d) = n / (-d) \\
 (-n) \setminus d &= -(n \setminus d) \\
 n \setminus (-d) &= n \setminus d \\
 0 \leq (n \setminus d) < d &\text{ where } n \geq 0 \text{ and } d > 0
 \end{aligned}$$

8.2.2 Integer shift operators

The available integer shift operators are listed in [Table 106](#).

Table 106. Shift operators

| Operation | Description |
|-----------|---------------------|
| $n \ll b$ | Integer left shift |
| $n \gg b$ | Integer right shift |

The shift operators are defined on integers as follows where $b \geq 0$:

$$n \ll b = n \times 2^b$$

$$n \gg b = \begin{cases} n / 2^b & \text{where } n \geq 0 \\ (n - 2^b + 1) / 2^b & \text{where } n < 0 \end{cases}$$

Note: Right shifting rounds the result towards minus infinity. This contrasts with division, which rounds towards zero, and is the reason why the right shift definition is separate for positive and negative n .

8.2.3 Integer bitwise operators

The available integer bitwise operators are listed in [Table 107](#).

Table 107. Bitwise operators

| Operation | Description |
|--|---|
| $i \wedge j$ | Integer bitwise AND |
| $i \vee j$ | Integer bitwise OR |
| $i \oplus j$ | Integer bitwise XOR |
| $\sim i$ | Integer bitwise NOT |
| $n_{\langle b \text{ FOR } m \rangle}$ | Integer bit-field extraction: extract m bits starting at bit b from integer n |
| $n_{\langle b \rangle}$ | Integer bit-field extraction: extract 1 bit starting at bit b from integer n |

In order to define bitwise operations all integers are considered as having an infinitely long two's complement representation. Bit 0 is the least significant bit of this representation, bit 1 is the next higher bit, and so on. The value of bit b , where $b \geq 0$, in integer n is given by:

$$\text{BIT}(n, b) = (n / 2^b) \setminus 2 \quad \text{where } n \geq 0$$

$$\text{BIT}(-n, b) = 1 - \text{BIT}(n - 1, b) \quad \text{where } n > 0$$

Care must be taken whenever the infinitely long two's complement representation of a negative number is constructed. This representation will contain an infinite number of higher bits with the value 1 representing the sign. Typically, a subsequent conversion operation is used to discard these upper bits and return the result back to a finite value.

Bitwise AND (\wedge), OR (\vee), XOR (\oplus) and NOT (\neg) are defined on integers as follows, where b takes all values such that $b \geq 0$:

$$\begin{aligned}\text{BIT}(i \wedge j, b) &= \text{BIT}(i, b) \times \text{BIT}(j, b) \\ \text{BIT}(i \vee j, b) &= \text{BIT}(i \wedge j, b) + \text{BIT}(i \oplus j, b) \\ \text{BIT}(i \oplus j, b) &= (\text{BIT}(i, b) + \text{BIT}(j, b)) \setminus 2 \\ \text{BIT}(\neg i, b) &= 1 - \text{BIT}(i, b)\end{aligned}$$

Note: Bitwise NOT of any finite positive i will result in a value containing an infinite number of higher bits with the value 1 representing the sign.

Bitwise extraction is defined on integers as follows, where $b \geq 0$ and $m > 0$:

$$\begin{aligned}n \langle b \text{ FOR } m \rangle &= (n \gg b) \wedge (2^m - 1) \\ n \langle b \rangle &= n \langle b \text{ FOR } 1 \rangle\end{aligned}$$

The result of $n \langle b \text{ FOR } m \rangle$ is an integer in the range $[0, 2^m)$.

8.2.4 Relational operators

Relational operators are defined to compare integral values and give a boolean result.

Table 108. Relational operators

| Operation | Description |
|------------|---|
| $i = j$ | Result is true if i is equal to j , otherwise false |
| $i \neq j$ | Result is true if i is not equal to j , otherwise false |
| $i < j$ | Result is true if i is less than j , otherwise false |
| $i > j$ | Result is true if i is greater than j , otherwise false |
| $i \leq j$ | Result is true if i is less than or equal to j , otherwise false |
| $i \geq j$ | Result is true if i is greater than or equal to j , otherwise false |

8.2.5 Boolean operators

Boolean operators are defined to perform logical AND, OR, XOR and NOT. These operators have boolean sources and result. Additionally, the conversion operator INT is defined to convert a boolean source into an integer result.

Table 109. Boolean operators

| Operation | Description |
|--------------------|--|
| $i \text{ AND } j$ | Result is true if i and j are both true, otherwise false |
| $i \text{ OR } j$ | Result is true if either/both i and j are true, otherwise false |
| $i \text{ XOR } j$ | Result is true if exactly one of i and j are true, otherwise false |

Table 109. Boolean operators (continued)

| Operation | Description |
|-----------|---|
| NOT i | Result is true if i is false, otherwise false |
| INT i | Result is 0 if i is false, otherwise 1 |

8.2.6 Single-value functions

In some cases it is inconvenient or inappropriate to describe an expression directly in the specification language. In these cases a function call is used to represent the desired behavior.

A single-value function evaluates to a single value (the result), which can be used in an expression. The type of the result value can be determined by the expression context from which the function is called. There are also multiple-value functions which evaluate to multiple values. These are only available in an assignment context, and are described in [Section 8.3.2: Assignment on page 196](#).

Functions can contain side-effects.

Two monadic functions are defined to support conversions between integral representations of finite-precision signed and unsigned number spaces. These functions are often used to convert between bit-fields and integer values.

Table 110. Integer conversion operators

| Function | Description |
|-----------------------------|---|
| ZeroExtend _n (i) | Convert integer i to an n-bit 2's complement unsigned range |
| SignExtend _n (i) | Convert integer i to an n-bit 2's complement signed range |

These two functions are defined as follows, where $n > 0$:

$$\text{ZeroExtend}_n(i) = i_{\langle 0 \text{ FOR } n \rangle}$$

$$\text{SignExtend}_n(i) = \begin{cases} i_{\langle 0 \text{ FOR } n \rangle} & \text{where } i_{\langle n-1 \rangle} = 0 \\ i_{\langle 0 \text{ FOR } (n-1) \rangle} - 2^n & \text{where } i_{\langle n-1 \rangle} = 1 \end{cases}$$

For syntactic convenience, conversion functions are also defined for converting an integer to a single bit and to a 32-bit register. [Table 111](#) shows the additional functions provided.

Table 111. Conversion operators from integers to bit-fields

| Operation | Description |
|-------------|--|
| Bit(i) | Convert lowest bit of integer i to a 1-bit value This is a convenient notation for $i_{\langle 0 \rangle}$ |
| Register(i) | Convert lowest 32 bits of integer i to a 32-bit value This is a convenient notation for $i_{\langle 0 \text{ FOR } 32 \rangle}$ |

Floating-point conversions

The specification language manipulates floating-point values as integers containing the associated IEEE754 bit-pattern. The layout of these bit-patterns is described in [Chapter 6: Floating-point unit on page 163](#). The language does not support a floating-point type.

Conversion functions are defined to support floating-point. Floating-point values are held as either scalar values in a single register, or vector values in multiple registers. The available register formats are:

- one 32-bit value in a single-precision register
- one 64-bit value in a double-precision register
- two 32-bit values in a pair of single-precision registers
- four 32-bit values in a four-entry vector of single-precision registers
- sixteen 32-bit values in a four-by-four matrix of single-precision registers

Conversions are available to convert between register bit-fields in these formats and integers or arrays of integers holding the appropriate IEEE754 bit-patterns.

The following conversions are provided to convert from floating-point registers:

Table 112. Conversion from floating-point register formats

| Operation | Description |
|------------------------------------|---|
| FloatValue ₃₂ (r) | Convert a single-precision floating-point register into a 32-bit integer bit-pattern. |
| FloatValue ₆₄ (r) | Convert a double-precision floating-point register into a 64-bit integer bit-pattern. |
| FloatValueVector ₃₂ (r) | Convert a 4-entry vector of single-precision floating-point registers into an array of 4 x 32-bit integer bit-patterns. |
| FloatValueMatrix ₃₂ (r) | Convert a 16-entry matrix of single-precision floating-point registers into an array of 16 x 32-bit integer bit-patterns. |

The following conversions are provided to convert to floating-point registers:

Table 113. Conversion to floating-point register formats

| Operation | Description |
|---------------------------------------|---|
| FloatRegister ₃₂ (i) | Convert a 32-bit integer bit-pattern into a single-precision floating-point register. |
| FloatRegister ₆₄ (i) | Convert a 64-bit integer bit-pattern into a double-precision floating-point register. |
| FloatRegisterPair ₃₂ (i) | Convert a 2 element array of two 32-bit integer bit patterns into a double-precision floating-point register. Element 0 is mapped to the upper half of the 64-bit register. |
| FloatRegisterVector ₃₂ (a) | Convert an array of 4 x 32-bit integer bit-patterns into a 4-entry vector of single-precision floating-point registers. |
| FloatRegisterMatrix ₃₂ (a) | Convert an array of 16 x 32-bit integer bit-patterns into a 16-entry matrix of single-precision floating-point registers. |

8.3 Statements

An instruction specification consists of a sequence of statements. These statements are processed sequentially in order to specify the effect of the instruction on the architectural state of the machine. The available statements are discussed in this section.

Each statement has a semi-colon terminator. A sequence of statements can be aggregated into a statement block using '{' to introduce the block and '}' to terminate the block. A statement block can be used anywhere that a statement can.

8.3.1 Undefined behavior

The statement:

```
UNDEFINED ( ) ;
```

indicates that the resultant behavior is architecturally undefined.

A particular implementation can choose to specify an implementation-defined behavior in such cases. It is very likely that any implementation-defined behavior will vary from implementation to implementation. Exploitation of implementation-defined behavior should be avoided to allow software to be portable between implementations.

In cases where architecturally undefined behavior can occur in user mode, the implementation will ensure that implemented behavior does not break the protection model. Thus, the implemented behavior will be some execution flow that is permitted for that user mode thread.

8.3.2 Assignment

The ' \leftarrow ' operator is used to denote assignment of an expression to a variable. An example assignment statement is:

```
variable  $\leftarrow$  expression;
```

The expression can be constructed from variables, literals, operators and functions as described in [Section 8.2: Expressions on page 191](#). The expression is fully evaluated before the assignment takes place. The variable can be an integer, a boolean, a bit-field or an array of one of these types.

Assignment to architectural state

This is where the variable is part of the architectural state (as described in [Table 114: Scalar architectural state on page 199](#)). The type of the expression and the type of the variable must match.

Assignment to a temporary

Alternatively, if the variable is not part of the architectural state, then it is a temporary variable. The type of the variable is determined by the type of expression. A temporary variable must be assigned to, before it is used in the instruction specification.

Assignment of an undefined value

An assignment of the following form results in a variable being initialized with an architecturally undefined value:

```
variable  $\leftarrow$  UNDEFINED;
```

After assignment the variable will hold a value which is valid for its type. However, the value is architecturally undefined. The actual value can be unpredictable; that is to say the value indicated by UNDEFINED can vary with each use of UNDEFINED. Architecturally-undefined values can occur in both user and privileged modes.

A particular implementation can choose to specify an implementation-defined value in such cases. It is very likely that any implementation-defined values will vary from implementation to implementation. Exploitation of implementation-defined values should be avoided to allow software to be portable between implementations.

Assignment of multiple values

Multi-value functions are used to return multiple values, and are only available when used in a multiple assignment context. The syntax consists of a list of comma-separated variables, an assignment symbol followed by a function call. The function is evaluated and returns multiple results into the variables listed. The number of variables and the number of results of the function must match. The assigned variables must all be distinct (that is, no aliases).

For example, a two-valued assignment from a function call with three parameters can be represented as:

```
variable1, variable2 ← call(param1, param2, param3);
```

8.3.3 Conditional

Conditional behavior is specified using 'IF', 'ELSE IF' and 'ELSE'.

Conditions are expressions that result in a boolean value. If the condition after an 'IF' is true, then its block of statements is executed and the whole conditional then completes. If the condition is false, then any 'ELSE IF' clauses are processed, in turn, in the same fashion. If no conditions are met and there is an 'ELSE' clause then its block of statements is executed. Finally, if no conditions are met and there is no 'ELSE' clause, then the statement has no effect apart from the evaluation of the condition expressions.

The 'ELSE IF' and 'ELSE' clauses are optional. In ambiguous cases, the 'ELSE' matches with the nearest 'IF'.

For example:

```
IF (condition1)
    block1
ELSE IF (condition2)
    block2
ELSE
    block3
```

8.3.4 Repetition

Repetitive behavior is specified using the following construct:

```
REPEAT i FROM m FOR n STEP s block
```

The block of statements is iterated n times, with the integer i taking the values:

m, m + s, m + 2s, m + 3s, up to m + (n - 1) × s.

The behavior is equivalent to textually writing the block n times with i being substituted with the appropriate value in each copy of the block.

The value of `n` must be greater or equal to 0, and the value of `s` must be non-zero. The values of the expressions for `m`, `n` and `s` must be constant across the iteration. The integer `i` must not be assigned to within the iterated block. The 'STEP `s`' can be omitted in which case the step-size takes the default value of 1.

8.3.5 Exceptions

Exception handling is triggered by a `THROW` statement. When an exception is thrown, no further statements are executed from the instruction specification and control passes to an exception handler. The actions associated with the launch of the handler are not shown in the instruction specification, but are described separately in [Chapter 5: Exceptions on page 146](#).

There are two forms of throw statement:

```
THROW type;
```

and:

```
THROW type, value;
```

where `type` indicates the type of exception which is launched, and `value` is an optional argument to the exception handling sequence.

The full set of exceptions is described in [Chapter 5: Exceptions on page 146](#).

8.3.6 Procedures

Procedure statements contain a procedure name followed by a list of comma-separated arguments contained within parentheses followed by a semi-colon. The execution of procedures typically causes side-effects to the architectural state of the machine.

Procedures are generally used where it is difficult or inappropriate to specify the effect of an instruction using the abstract execution model. A fuller description of the effect of the instruction will be given in the surrounding text.

An example procedure with two parameters is:

```
proc(param1, param2);
```

8.4 Architectural state

The architectural state is described in [Chapter 2: Programming model on page 18](#). The notations used in the model to refer to this state are summarized in [Table 114](#) and [Table 115](#). Each item of scalar architectural state is a bit-field of a particular width. Each item of array architectural state is an array of bit-fields of a particular width.

Table 114. Scalar architectural state

| Architectural state | Type is a bit-field containing: | Description |
|------------------------------------|---------------------------------|--|
| MD (SR.MD) | 1 bit | User (0) or privileged (1) mode |
| PC | 32 bits | 32-bit program counter |
| MMUCR | 32 bits | For details of the MMU control register see Chapter 3: Memory management unit (MMU) on page 32 . |
| FPSCR | 32 bits | 32-bit floating-point status and control register |
| GBR | 32 bits | Global base register |
| MACL | 32 bits | Multiply-accumulate low |
| MACH | 32 bits | Multiply-accumulate high |
| PR | 32 bits | Procedure link register |
| T | 1 bit | Condition code flag |
| S | 1 bit | Multiply-accumulate saturation flag |
| M | 1 bit | Divide-step M flag |
| Q | 1 bit | Divide-step Q flag |
| FPUL | 32 bits | FPU communication register |
| R_i where i is in $[15:0]$ | 32 bits | 16 x 32-bit general purpose registers |
| FR_i where i is in $[31:0]$ | 32 bits | 32 x 32-bit floating-point registers |
| DR_{2i} where i is in $[15:0]$ | 64 bits | 16 x 64-bit floating-point registers |

Table 115. Array architectural state

| Architectural state | Type is an array of bit-fields each containing: | Description |
|--|---|--|
| FP_{2i} where i is in $[31:0]$ | 32 bits | 32 pairs of 32-bit floating-point registers |
| FV_{4i} where i is in $[15:0]$ | 32 bits | 16 vectors of 4 x 32-bit floating-point registers |
| $MTRX_{16i}$ where i is in $[3:0]$ | 32 bits | 4 matrices of 16 x 32-bit floating-point registers |
| $MEM[i]$ where i is in $[0, 2^{32})$ | 8 bits | 2^{32} bytes of memory |
| $UTLB[i]$ where i is in $[63:0]$ | a UTLB entry | Used for translation, for further details see Chapter 3: Memory management unit (MMU) on page 32 . |

Note: FR , FP , FV , $MTRX$ and DR provide different views of the same architectural state.

There is no implicit meaning to the value held by the collection of bits in a register. The interpretation of the register is supplied by each instruction that reads or writes the register value.

PC denotes the program counter of the currently executing instruction. PC' denotes the program counter of the next instruction that is to be executed.

The IsDelaySlot() function appears in the descriptions of instructions whose behavior is affected by being executed in the delay slot of a branching operation.

Table 116. Support functions for general execution

| Function | Description |
|---------------|--|
| IsDelaySlot() | Returns true if instruction is being executed in a delay slot. |

8.5 Memory model

Instruction specification uses a simple model of memory. It assumes, for example, that any caches have no architectural visibility. For typical well-disciplined instruction sequences these effects will not be architecturally visible. However, a fuller description of the behavior in other cases is defined by the text of the architecture manual.

MEM is an array of bytes indexed by a virtual address. Elements in arrays are selected using array indexing notation: MEM[i] selects the i^{th} entry in the MEM array. The total range of array indices into MEM is $[0, 2^{32})$, though not all of this memory is available on all implementations.

Array slicing can be used to view an array as consisting of elements of a larger size. The notation MEM[s FOR n], where $n > 0$, denotes a memory slice containing the elements MEM[s], MEM[s+1] through to MEM[s+n-1]. The type of this slice is a bit-field exactly large enough to contain a concatenation of the n selected elements. In this case it contains 8n bits since the base type of MEM is byte.

The order of the concatenation depends on the endianness of the processor.

- If the processor is operating in a little-endian mode, the concatenation order obeys the following condition as i (the byte number) varies in the range [0, n):

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8i \text{ FOR } 8 \rangle} = \text{MEM}[s + i]$$

This equivalence states that byte number i, using little-endian byte numbering (that is, byte 0 is bits 0 to 7), in the bit-field MEM[s FOR n] is the i^{th} byte in memory counting upwards from MEM[s].

- If the processor is operating in a big-endian mode, the concatenation order obeys the following condition as i (the byte number) varies in the range [0, n):

$$(\text{MEM}[s \text{ FOR } n])_{\langle 8(n-1-i) \text{ FOR } 8 \rangle} = \text{MEM}[s + i]$$

This equivalence states that byte number i, using big-endian byte numbering (for example, byte 0 is bits 8n-8 to 8n-1), in the bit-field MEM[s FOR n] is the i^{th} byte in memory counting upwards from MEM[s].

For syntactic convenience, functions and procedures are provided to read, write and swap memory. The basic primitives support aligned accesses. Misaligned read and write primitives support the instructions for misaligned load and store.

Additionally, mechanisms are provided for reading and writing pairs of values. Pair access requires that each half of the pair is endianness converted separately, and that the lower half is written into memory at the provided address while the upper half is written into that address plus the object size. This maintains the ordering of the halves of the pair as they are transferred between registers and memory. Pair access is used only for loading and storing pairs of single-precision floating-point registers (see [Chapter 6: Floating-point unit on page 163](#)).

8.5.1 Support functions

The specification of the memory instructions relies on the support functions listed in [Table 117](#). These functions are used to model the behavior of the memory management unit described in [Chapter 3: Memory management unit \(MMU\) on page 32](#).

Table 117. Support functions for memory access

| Function | Description |
|-----------------------------|---|
| AddressUnavailable(address) | Returns true if the provided address is outside of the available part of the virtual address space. For further details refer to Chapter 3: Memory management unit (MMU) on page 32 . |
| MMU() | Returns true if the MMU is enabled. |
| InstFetchMultiHit(address) | Returns true if more than mapping for an instruction to be fetched from the specified address is found. |
| InstFetchMiss(address) | Returns true if the provided address does not have a mapping for an instruction fetch. |
| DataAccessMultiHit(address) | Returns true if more than mapping for a data access to the specified address is found. |
| DataAccessMiss(address) | Returns true if the provided address does not have a mapping for a data access. |
| ReadProhibited(address) | Returns true if the provided address has no read permission for the current privilege. |
| WriteProhibited(address) | Returns true if the provided address has no write permission for the current privilege. |
| ExecuteProhibited(address) | Returns true if the provided address has no execute permission for the current privilege. |
| DirtyBit(address) | Returns the value of the dirty bit (D) in the UTLB for the translation used for the specified address. |
| IsLittleEndian() | Returns true if processor is little-endian. |

More detailed properties of translation miss detection are not modelled here. The conditions that determine whether an access is a translation miss or a hit depend on the MMU and cache.

DataAccessMiss is used to check for the absence of a data translation. This function is used for all data accesses when the MMU is enabled. InstFetchMiss is used to check for instruction fetches.

8.5.2 Instruction fetch

The following function is provided to fetch an instruction.

Table 118. Support functions for instruction fetch

| Function | Description |
|--------------------|---|
| InstFetch(address) | Fetch an instruction at specified address |

The InstFetch function takes an integer parameter to indicate the address being accessed. The number of bits read is 16. The required bytes are read from memory, interpreted according to endianness, and an integer result returns the opcode value.

The assignment:

```
result ← InstFetch(a);
```

is equivalent to:

```
IF (MMU() AND InstFetchMultiHit(a)) THROW ITLBMULTIHIT,a
IF (AddressUnavailable(a) OR ((a^1) ≠ 0)) THROW IADDERR,a;
IF (MMU() AND InstFetchMiss(a)) THROW ITLBMISS,a;
IF (MMU() AND ExecuteProhibited(a)) THROW EXECPROT,a;
result ← MEM[a FOR width];
```

8.5.3 Reading memory

Functions are provided to read memory.

Table 119. Support functions to read memory

| Function | Description |
|---------------------------------------|---|
| ReadMemory _n (address) | Aligned memory read of an n-bit value |
| ReadMemoryPair _n (address) | Aligned memory read of a pair of n-bit values |

The ReadMemory_n function takes an integer parameter to indicate the address being accessed. The number of bits being read (n) is one of 8, 16, 32 or 64 bits. The required bytes are read from memory, interpreted according to endianness, and an integer result returns the read bit-field value. If the read memory value is to be interpreted as signed, then a sign-extension should be used on the result.

The assignment:

```
result ← ReadMemoryn(a);
```

is equivalent to:

```
width ← n >> 3;
IF (MMU() AND DataAccessMultiHit(a)) THROW OTLBMULTIHIT,a
IF (AddressUnavailable(a) OR ((a^(width-1)) ≠ 0)) THROW RADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS,a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT,a;
result ← MEM[a FOR width];
```

ReadMemoryPair_n reads a pair of n-bit values and returns a 2n-bit value. The alignment check requires alignment for a 2n-bit access. The upper half of the value is read from the

supplied address, the lower half $n/8$ bytes later, with the endianness applied separately to each half. The statement:

```
result ← ReadMemoryPairn(a);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← width << 1;
IF (MMU() AND DataAccessMultiHit(a)) THROW OTLBMULTIHIT, a
IF (AddressUnavailable(a) OR ((a^(pairwidth-1)) ≠ 0)) THROW
RADDERR, a;
IF (MMU() AND DataAccessMiss(a)) THROW RTLBMISS, a;
IF (MMU() AND ReadProhibited(a)) THROW READPROT, a;
high ← MEM[a FOR width];
low ← MEM[a+width FOR width];
result ← (high << n)+low;
```

8.5.4 Prefetching memory

A function is provided to denote memory prefetch.

Table 120. Support procedure to prefetch memory

| Function | Description |
|-------------------------|-----------------|
| PrefetchMemory(address) | Memory prefetch |

This is used for a software-directed data prefetch from a specified virtual address. This is a hint to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed.

The statement:

```
result ← PrefetchMemory(a);
```

is equivalent to:

```
IF (NOT AddressUnavailable(address))
  IF (MMU() AND DataAccessMultiHit(a))
    IF (NOT (MMU() AND DataAccessMiss(address)))
      IF (NOT (MMU() AND ReadProhibited(address)))
        PREF(address);
result ← 0;
```

where PREF is a cache operation defined in [Section 8.7: Cache model on page 205](#). This function does not raise exceptions. PrefetchMemory evaluates to zero for syntactic convenience.

8.5.5 Writing memory

Procedures are provided to write memory.

Table 121. Support procedures to write memory

| Function | Description |
|---|--|
| WriteMemory _n (address, value) | Aligned memory write to an n-bit value |
| WriteMemoryPair _n (address, value) | Aligned memory write to a pair of n-bit values |

The WriteMemory_n procedure takes an integer parameter to indicate the address being accessed, followed by an integer parameter containing the value to be written. The number of bits being written (n) is one of 8, 16 or 32 bits. The written value is interpreted as a bit-field of the required size; all higher bits of the value are discarded. The bytes are written to memory, ordered according to endianness. The statement:

```
WriteMemoryn(a, value);
```

is equivalent to:

```
width ← n >> 3;
IF (MMU() AND DataAccessMultiHit(a)) THROW OTLBMULTIHIT,a
IF (AddressUnavailable(a) OR ((a^(width-1)) ≠ 0)) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
IF (MMU() AND NOT DirtyBit(a)) THROW FIRSTWRITE,a;
MEM[a FOR width] ← value<0 FOR n>;
```

WriteMemoryPair_n writes a 2n-bit value as a pair of n-bit values. The alignment check requires alignment for a 2n-bit access. The upper half of the value is written to the supplied address, the lower half n/8 bytes later, with the endianness applied separately to each half. The statement:

```
WriteMemoryPairn(a, value);
```

is equivalent to:

```
width ← n >> 3;
pairwidth ← width << 1;
IF (MMU() AND DataAccessMultiHit(a)) THROW OTLBMULTIHIT,a
IF (AddressUnavailable(a) OR ((a^(pairwidth-1)) ≠ 0)
) THROW WADDERR,a;
IF (MMU() AND DataAccessMiss(a)) THROW WTLBMISS,a;
IF (MMU() AND WriteProhibited(a)) THROW WRITEPROT,a;
IF (MMU() AND NOT DirtyBit(a)) THROW FIRSTWRITE,a;
MEM[a FOR width] ← value<n FOR n>;
MEM[a+width FOR width] ← value<0 FOR n>;
```

8.6 Sleep and synchronization operations

The SLEEP operation is used to enter sleep mode. The SYNCO performs a synchronization operation which is used to enforce a strict ordering between operand data accesses. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Table 122. Procedures to model sleep operation

| Procedure | Description |
|-----------|---------------------------------------|
| SLEEP() | Procedure to enter sleep mode |
| SYNCO() | Procedure to synchronize operand data |

8.7 Cache model

Cache operations are used to allocate, prefetch and cohere lines in caches. The effects of these operations are beyond the scope of the specification language, and are therefore modelled using procedure calls. The behavior of these procedure calls is elaborated in the text of the manual.

Table 123. Procedures to model cache operations

| Procedure | Description |
|-----------------|--|
| ALLOCO(address) | Procedure to allocate an operand cache block. |
| ICBI(address) | Procedure to invalidate an instruction cache block |
| OCBI(address) | Procedure to invalidate an operand cache block. |
| OCBP(address) | Procedure to purge an operand cache block. |
| OCBWB(address) | Procedure to write-back an operand cache block. |
| PREF (address) | Procedure to prefetch an operand cache block. |

8.8 Floating-point model

The floating-point specification is abstracted using functions to hide the low-level details. Additional information is provided in a tabular form to describe special and exceptional cases. [Chapter 6: Floating-point unit on page 163](#) provides a textual description of floating-point operation.

8.8.1 Functions to access SR and FPSCR

The floating-point instruction specifications use a function notation to access SR and FPSCR state. The used functions are described in [Table 124](#).

Table 124. SR and FPSCR access

| Function | Description |
|-------------------|--|
| FpuIsDisabled(SR) | True if SR.FD is 1, otherwise false |
| FpuFlagI(FPSCR) | True if FPSCR.FLAG.I (sticky flag for inexact) is 1, otherwise false |
| FpuFlagU(FPSCR) | True if FPSCR.FLAG.U (sticky flag for underflow) is 1, otherwise false |
| FpuFlagO(FPSCR) | True if FPSCR.FLAG.O (sticky flag for overflow) is 1, otherwise false |
| FpuFlagZ(FPSCR) | True if FPSCR.FLAG.Z (sticky flag for divide by zero) is 1, otherwise false |
| FpuFlagV(FPSCR) | True if FPSCR.FLAG.V (sticky flag for invalid) is 1, otherwise false |
| FpuCauseI(FPSCR) | True if FPSCR.CAUSE.I (cause flag for inexact) is 1, otherwise false |
| FpuCauseU(FPSCR) | True if FPSCR.CAUSE.U (cause flag for underflow) is 1, otherwise false |
| FpuCauseO(FPSCR) | True if FPSCR.CAUSE.O (cause flag for overflow) is 1, otherwise false |
| FpuCauseZ(FPSCR) | True if FPSCR.CAUSE.Z (cause flag for divide by zero) is 1, otherwise false |
| FpuCauseV(FPSCR) | True if FPSCR.CAUSE.V (cause flag for invalid) is 1, otherwise false |
| FpuCauseE(FPSCR) | True if FPSCR.CAUSE.E (cause flag for FPU error) is 1, otherwise false |
| FpuEnableI(FPSCR) | True if FPSCR.ENABLE.I (exception enable for inexact) is 1, otherwise false |
| FpuEnableU(FPSCR) | True if FPSCR.ENABLE.U (exception enable for underflow) is 1, otherwise false |
| FpuEnableO(FPSCR) | True if FPSCR.ENABLE.O (exception enable for overflow) is 1, otherwise false |
| FpuEnableZ(FPSCR) | True if FPSCR.ENABLE.Z (exception enable for divide by zero) is 1, otherwise false |
| FpuEnableV(FPSCR) | True if FPSCR.ENABLE.V (exception enable for invalid) is 1, otherwise false |

8.8.2 Functions to model floating-point behavior

Functions are used to model almost all of the floating-point behavior. Each function is associated with a list of results and a list of parameters. The functions encapsulate the computation associated with the instruction. This includes handling of input denormalized values, special case detection, exceptional cases and the floating-point arithmetic.

The following tables summarize the functions used by each instruction. The table shows how the parameters are interpreted and how the results are computed. The n^{th} parameter is denoted as P_n and the n^{th} result as RES_n .

The parameters and results of these functions are all modeled as integer values. For floating-point parameters and results, these values are integer bit-patterns representing the IEEE754 formats. Multi-value results are used to return two results: the computed result and a new value for FPSCR. If the new value of FPSCR causes an exception to be raised, then the destination register will not be updated with the computed result.

Table 125. Floating-point dyadic arithmetic

| Instruction | Function | RES0 | RES1 | P0, P1 | P2 |
|-------------|----------|--|-----------|--------|-----------|
| FADD | FADD_S | Single result of $(P0 +_{\text{IEEE754}} P1)$ | New FPSCR | Single | Old FPSCR |
| FADD | FADD_D | Double result of $(P0 +_{\text{IEEE754}} P1)$ | New FPSCR | Double | Old FPSCR |
| FSUB | FSUB_S | Single result of $(P0 -_{\text{IEEE754}} P1)$ | New FPSCR | Single | Old FPSCR |
| FSUB | FSUB_D | Double result of $(P0 -_{\text{IEEE754}} P1)$ | New FPSCR | Double | Old FPSCR |
| FMUL | FMUL_S | Single result of $(P0 \times_{\text{IEEE754}} P1)$ | New FPSCR | Single | Old FPSCR |
| FMUL | FMUL_D | Double result of $(P0 \times_{\text{IEEE754}} P1)$ | New FPSCR | Double | Old FPSCR |
| FDIV | FDIV_S | Single result of $(P0 /_{\text{IEEE754}} P1)$ | New FPSCR | Single | Old FPSCR |
| FDIV | FDIV_D | Double result of $(P0 /_{\text{IEEE754}} P1)$ | New FPSCR | Double | Old FPSCR |

Table 126. Floating-point monadic arithmetic

| Instruction | Function | RES0 | RES1 | P0 | P1 |
|--------------|----------|---|------------|--------|-----------|
| FABS | FABS_S | Single result of absolute P0 | (not used) | Single | Old FPSCR |
| FABS | FABS_D | Double result of absolute P0 | (not used) | Double | Old FPSCR |
| FNEG | FNEG_S | Single result of negating P0 | (not used) | Single | Old FPSCR |
| FNEG | FNEG_D | Double result of negating of P0 | (not used) | Double | Old FPSCR |
| FSQRT | FSQRT_S | Single result of $_{\text{IEEE754}}\sqrt{P0}$ | New FPSCR | Single | Old FPSCR |
| FSQRT | FSQRT_D | Double result of $_{\text{IEEE754}}\sqrt{P0}$ | New FPSCR | Double | Old FPSCR |

Table 127. Floating-point comparisons

| Instruction | Function | RES0 | RES1 | P0, P1 | P2 |
|----------------|----------|---|-----------|--------|-----------|
| FCMP/EQ | FCMPEQ_S | Boolean result of ($P0 \approx_{IEEE754} P1$) | New FPSCR | Single | Old FPSCR |
| FCMP/EQ | FCMPEQ_D | Boolean result of ($P0 \approx_{IEEE754} P1$) | New FPSCR | Double | Old FPSCR |
| FCMP/GT | FCMPGT_S | Boolean result of ($P0 >_{IEEE754} P1$) | New FPSCR | Single | Old FPSCR |
| FCMP/GT | FCMPGT_D | Boolean result of ($P0 >_{IEEE754} P1$) | New FPSCR | Double | Old FPSCR |

Table 128. Floating-point conversions

| Instruction | Function | RES0 | RES1 | P0 | P1 |
|---------------|----------|---|-----------|------------|-----------|
| FCNVSD | FCNV_SD | P0 is converted to double result | New FPSCR | Single | Old FPSCR |
| FCNVDS | FCNV_DS | P0 is converted to single result | New FPSCR | Double | Old FPSCR |
| FTRC | FTRC_SL | P0 is converted to signed 32-bit integer result | New FPSCR | Single | Old FPSCR |
| FTRC | FTRC_DL | P0 is converted to signed 32-bit integer result | New FPSCR | Double | Old FPSCR |
| FLOAT | FLOAT_LS | P0 is converted to single result | New FPSCR | 32-bit int | Old FPSCR |
| FLOAT | FLOAT_LD | P0 is converted to double result | New FPSCR | 32-bit int | Old FPSCR |

Table 129. Floating-point multiply-accumulate

| Instruction | Function | RES0 | RES1 | P0, P1, P2 | P3 |
|-------------|----------|--|-----------|------------|-----------|
| FMAC | FMAC_S | Single result of fused ($P0 \times P1$) + P2 | New FPSCR | Single | Old FPSCR |

Table 130. Special-purpose floating-point dyadic arithmetic

| Instruction | Function | RES0 | RES1 | P0 | P1 | P2 |
|-------------|----------|---|-----------|---------------------|--------------------|-----------|
| FIPR | FIPR_S | Single result of inner product of P0 with P1 | New FPSCR | Array of 4 singles | Array of 4 singles | Old FPSCR |
| FTRV | FTRV_S | Array of 4 single results of matrix transform of P0 with P1 | New FPSCR | Array of 16 singles | Array of 4 singles | Old FPSCR |

Table 131. Special-purpose floating-point monadic arithmetic

| Instruction | Function | RES0 | RES1 | P0 | P1 |
|--------------|----------|--|-----------|--------|-----------|
| FSRRA | FSRRA_S | Single result approximating reciprocal square root of P0 | New FPSCR | Single | Old FPSCR |

8.8.3 Floating-point special cases and exceptions

A special-case table is provided for each floating-point instruction that is considered an operation and has at least one input that is interpreted as a floating-point value. This table enumerates all different possible combinations of input values and the results returned by the instruction in the absence of an exception being raised.

The entries in the table are IEEE754 floating-point values as described in [Chapter 6: Floating-point unit on page 163](#). Each cell entry in the table describes the result returned for a particular combination of floating-point inputs. If the result is invariant, its value is given in the cell. If the result is variable, the name of the appropriate operation is entered in the cell. If the cell contains 'n/a' then this indicates that an exception is always raised for that combination of inputs and that the implementation does not associate any value with the result.

8.9 Abstract sequential model

This section describes the abstract sequential model that is used to specify how instructions are executed on the ST40. It is described in terms of transitions in the explicit architectural state of the device plus some hidden internal state, PC" which is used in specifying the behavior of delayed branches. This additional internal state is initialized to PC+2, that is, it points to the instruction immediately after that at which execution will commence.

The steps associated with executing each instruction are:

1. Check for asynchronous events, such as interrupt or reset, and initiate handling if required. Asynchronous events are not accepted between a delayed branch and a delay slot. They are delayed until after the delay slot.
2. Check the current program counter (PC) for instruction address exceptions, and initiate handling if required. See [Section 5.5.2: General exceptions on page 156](#).
3. Fetch the instruction bytes from the address in memory, as indicated by the current program counter, 2 bytes need to be fetched for each instruction.
4. Set the default value for PC' to the value of PC".
5. Assume, as default, continued sequential execution by setting PC" to PC'+2.
6. Decode and execute the instruction. This includes checks for synchronous events, such as exceptions and panics, and initiation of handling of these if required. Synchronous events are not accepted between a delayed branch and a delay slot. They are detected either before the delayed branch or after the delay slot.

The execution of an instruction can update the PC as follows:

- The instruction can change PC' to perform a branch immediately after this instruction has completed. It must also update PC" to the value of PC'+2 to ensure correct sequential execution after the control flow.
- The instruction can change PC" to perform a branch or procedure call after the next instruction has completed. Thus this state change will propagate in to the PC after the next instruction has executed.

Any changes made to PC' or PC" over-ride the default values.

7. Set the current program counter (PC) to the value of the next program counter (PC').

The actions associated with the handling of asynchronous and synchronous events are described in [Chapter 5: Exceptions on page 146](#). The actions required by step 6 depend on the instruction, and are specified by the instruction specification for that instruction.

There is a potential ambiguity when an instruction makes a delayed state change (that is it modifies PC”) and the immediately following instruction (which is in a delay slot) reads or writes the PC. In fact this cannot happen as all instructions which can read or write the PC are illegal in a delay slot and cause an ILLSLOT exception.

8.10 Example instructions

8.10.1 ADD #imm, Rn

An example specification for this instruction is shown below.

ADD #imm, Rn

| | | |
|------|----|----|
| 0111 | n | s |
| 15 | 12 | 11 |
| | 8 | 7 |
| | | 0 |

```

imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);
op2 ← op2 + imm;
Rn ← Register(op2);

```

The top half of this figure shows the assembly syntax and the binary encoding of the instruction. Particular fields within the encoding are identified by single characters. The opcode field, and any extension field, contain the literal encoding values associated with that instruction. Reserved fields must be encoded with the literal value given in the figure. Operand fields contain register designators or immediate constants.

The lower half of this figure specifies the effects of the execution of the instruction on the architectural state of the machine. The specification statements are organized into 3 stages as follows:

1. The first two statements read all required source information:

```

imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);

```

The first statement reads the value of s, interprets it as a sign-extended 8-bit integer value and assigns this to a temporary integer called ‘imm’. The name ‘imm’ corresponds to the name of the immediate used in the assembly syntax. The second statement reads the value of R_n register, interprets it as a sign-extended 32-bit integer value and assigns this to a temporary integer called op2.

2. The next statement implements the addition:

```

op2 ← op2 + imm;

```

This statement does not refer to any architectural state. It adds the 2 integers ‘imm’ and ‘op2’ together, and assigns the result to a temporary integer called ‘op2’. Note that since this is a conventional mathematical addition, the result can contain more significant bits of information than the sources.

3. The final statement updates the architectural state:

```

Rn ← Register(op2);

```

The integer ‘op2’ is converted back to a register bit-field, assigned to the R_n register.

8.10.2 FADD FRm, FRn

An example specification for this instruction is shown below.

FADD FRm, FRn

| | | | |
|------|----|----|------|
| 1111 | n | m | 0000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

The specification statements are organized as follows:

1. Read all required source information:

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);

```

Execute the instruction:

```

IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;

```

```
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
  THROW FPUExc, fps;
```

The behavior of the floating-point single-precision addition is modelled by the FADD_S procedure. This procedure is given the two source operands and the current value of FPSCR, and calculates the result and the new value of FPSCR. It is responsible for detecting special cases and exceptions, and setting the result and new FPSCR values accordingly.

This instruction contains exception cases. These are detected by IF statements and are raised by THROW statements. When a THROW statement is executed, no further statements from the specification are processed. In exception cases, this specification makes no updates to architectural state. Instead, a handler is launched for the exception as described in [Chapter 5: Exceptions on page 146](#). The THROW statement includes arguments to specify the kind of exception and any necessary parameters of that exception. For an FPUExc exception, the THROW statement includes an updated value of 'fps' which the exception handler uses to initialize FPSCR during the launch sequence.

2. Update the architectural state:

```
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);
```

9 Instruction descriptions

This chapter provides an alphabetical list of the instructions in the ST40 instruction set.

9.1 Alphabetical list of instructions

Instructions are listed in this section in alphabetical order.

ADD Rm, Rn

Description: This instruction adds R_m to R_n and places the result in R_n .

Operation:

ADD Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 1100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
op2 ← op2 + op1;
Rn ← Register(op2);

```

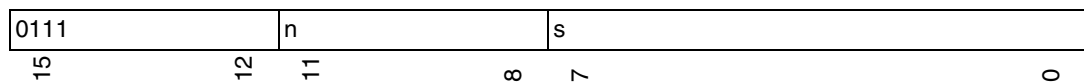
Note:

ADD #imm, Rn

Description: This instruction adds R_n to the sign-extended 8-bit immediate s and places the result in R_n .

Operation:

ADD #imm, Rn



```

imm ← SignExtend8(s);
op2 ← SignExtend32(Rn);
op2 ← op2 + imm;
Rn ← Register(op2);

```

Note: The '#imm' in the assembly syntax represents the immediate s after sign extension.

ADDC Rm, Rn

Description: This instruction adds R_m , R_n and the T-bit. The result of the addition is placed in R_n . and the carry-out from the addition is placed in the T-bit.

Operation:

ADDC Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 1110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
op2 ← (op2 + op1) + t;
t ← op2< 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);
```

Note:

ADDV Rm, Rn

Description: This instruction adds R_m to R_n and places the result in R_n . The T-bit is set to 1 if the addition result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

Operation:

ADDV Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 1111 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
op2 ← op2 + op1;  
t ← INT ((op2 < (- 231)) OR (op2 ≥ 231));  
Rn ← Register(op2);  
T ← Bit(t);
```

Note:

AND Rm, Rn

Description: This instruction performs bitwise AND of R_m with R_n and places the result in R_n .

Operation:

AND Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1001 |
| 15 | 12 | 11 | 8 |
| 7 | 4 | 3 | 0 |

```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2 ^ op1;
Rn ← Register(op2);

```

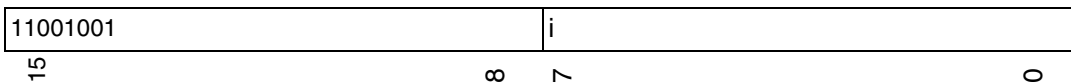
Note: This instruction performs a 32-bit bitwise AND.

AND #imm, R0

Description: This instruction performs bitwise AND of R_0 with the zero-extended 8-bit immediate i and places the result in R_0 .

Operation:

AND #imm, R0



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 ^ imm;
R0 ← Register(r0);

```

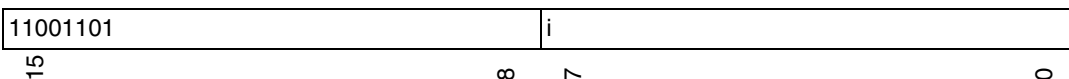
Note: This instruction performs a 32-bit bitwise AND. The '#imm' in the assembly syntax represents the immediate i after zero extension.

AND.B #imm, @(R0, GBR)

Description: This instruction performs a bitwise AND of an immediate constant with 8 bits of data held in memory. The virtual address is calculated by adding R_0 and GBR. The 8 bits of data at the virtual address are read. A bitwise AND is performed of the read data with the zero-extended 8-bit immediate i . The result is written back to the 8 bits of data at the same virtual address.

Operation:

AND.B #imm, @(R0, GBR)



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ^ imm;
WriteMemory8(address, value);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

Note: Zero-extension is performed on the virtual address computation allowing wrap around to occur.

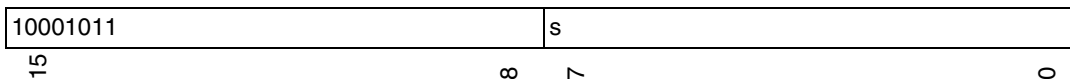
The '#imm' in the assembly syntax represents the immediate i after zero extension.

BF label

Description: This instruction is a conditional branch. The 8-bit displacement *s* is sign-extended, doubled and added to PC+4 to form the target address. If the T-bit is 1, the branch is not taken. If the T-bit is 0, the target address is copied to the PC.

Operation:

BF label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
newpc ← SignExtend32(PC');
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 0)
{
    temp ← ZeroExtend32(pc + 4 + label);
    newpc ← temp;
    delayedpc ← temp + 2;
}
PC' ← Register(newpc);
PC'' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

This is not a delayed branch instruction. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

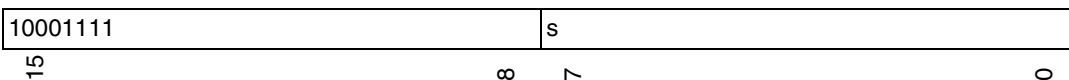
If the branch target address is invalid then the IADDERR trap is not delivered until after the branch instruction completes its execution and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BF/S label

Description: This instruction is a delayed conditional branch. The 8-bit displacement s is sign-extended, doubled and added to $PC+4$ to form the target address. If the T-bit is 1, the branch is not taken. If the T-bit is 0, the delay slot is executed and then the target address is copied to the PC.

Operation:

BF/S label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 0)
{
    temp ← ZeroExtend32(pc + 4 + label);
    delayedpc ← temp;
}
PC'' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate s after sign extension and scaling.

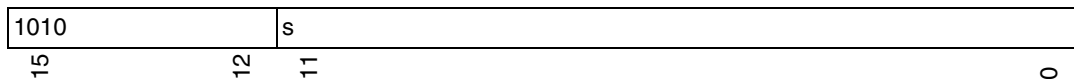
If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BRA label

Description: This instruction is a delayed unconditional branch. The 12-bit displacement *s* is sign-extended, doubled and added to PC+4 to form the target address. The delay slot is executed and then the target address is copied to the PC.

Operation:

BRA label



```
pc ← SignExtend32(PC);
label ← SignExtend12(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
temp ← ZeroExtend32(pc + 4 + label);
delayedpc ← temp;
PC'' ← Register(delayedpc);
```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BRAF Rn

Description: This instruction is a delayed unconditional branch. The target address is calculated by adding R_n to PC+4.

Operation:

BRAF Rn

| | | |
|------|----|----------|
| 0000 | n | 00100011 |
| 15 | 12 | 8 |

```
pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← ZeroExtend32(pc + 4 + op1);
delayedpc ← target;
PC' ← Register(delayedpc);
```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching occurs. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

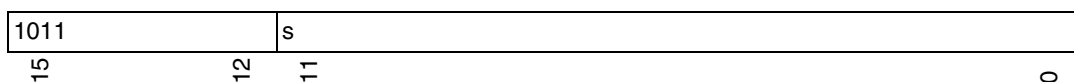
If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BSR label

Description: This instruction is a delayed unconditional branch used for branching to a subroutine. The 12-bit displacement *s* is sign-extended, doubled and added to PC+4 to form the target address. The delay slot is executed and then the target address is copied to the PC. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

Operation:

BSR label



```

pc ← SignExtend32(PC);
label ← SignExtend12(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
pr ← pc + 4;
temp ← ZeroExtend32(pc + 4 + label);
delayedpc ← temp;
PR ← Register(pr);
PC' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot. The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BSRF Rn

Description: This instruction is a delayed unconditional branch used for branching to a far subroutine. The target address is calculated by adding R_n to PC+4. The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

Operation:

BSRF Rn

| | | |
|------|----|----------|
| 0000 | n | 00000011 |
| 15 | 12 | 8 |

```

pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
pr ← pc + 4;
target ← ZeroExtend32(pc + 4 + op1);
delayedpc ← target;
PR ← Register(pr);
PC' ← Register(delayedpc);
    
```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

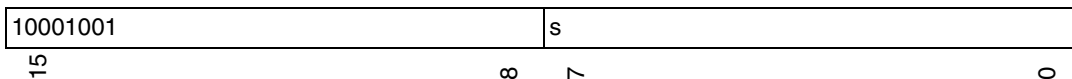
If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BT label

Description: This instruction is a conditional branch. The 8-bit displacement *s* is sign-extended, doubled and added to PC+4 to form the target address. If the T-bit is 0, the branch is not taken. If the T-bit is 1, the target address is copied to the PC.

Operation:

BT label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
newpc ← SignExtend32(PC');
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 1)
{
    temp ← ZeroExtend32(pc + 4 + label);
    newpc ← temp;
    delayedpc ← temp + 2;
}
PC' ← Register(newpc);
PC'' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

This is not a delayed branch instruction. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate *s* after sign extension and scaling.

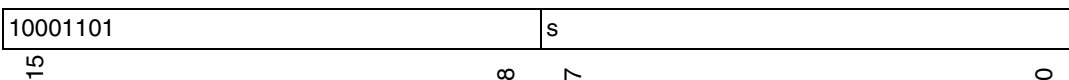
If the branch target address is invalid then the IADDERR trap is not delivered until after the branch instruction completes its execution and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

BT/S label

Description: This instruction is a delayed conditional branch. The 8-bit displacement s is sign-extended, doubled and added to $PC+4$ to form the target address. If the T-bit is 0, the branch is not taken. If the T-bit is 1, the delay slot is executed and then the target address is copied to the PC.

Operation:

BT/S label



```

t ← ZeroExtend1(T);
pc ← SignExtend32(PC);
delayedpc ← SignExtend32(PC'');
label ← SignExtend8(s) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (t = 1)
{
    temp ← ZeroExtend32(pc + 4 + label);
    delayedpc ← temp;
}
PC'' ← Register(delayedpc);
  
```

Exceptions: ILLSLOT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'label' in the assembly syntax represents the immediate s after sign extension and scaling.

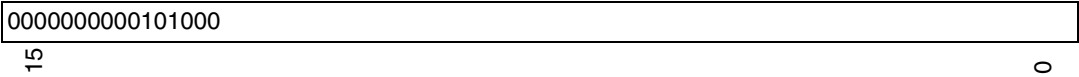
If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

CLRMAC

Description: This instruction clears MACL and MACH.

Operation:

CLRMAC



```
mac1 ← 0;
mach ← 0;
MACL ← ZeroExtend32(mac1);
MACH ← ZeroExtend32(mach);
```

Note:

CLRS

Description: This instruction clears the S-bit.

Operation:

CLRS

0000000001001000

150

s ← 0;
S ← Bit(s);

Note:

CLRT

Description: This instruction clears the T-bit.

Operation:

CLRT



Note:

CMP/EQ Rm, Rn

Description: This instruction sets the T-bit if the value of R_n is equal to the value of R_m , otherwise it clears the T-bit.

Operation:

CMP/EQ Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 0000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
t ← INT (op2 = op1);  
T ← Bit(t);
```

Note:

Description: This instruction sets the T-bit if the value of R_0 is equal to the sign-extended 8-bit immediate s , otherwise it clears the T-bit.

CMP/EQ #imm, R0



Note: The '#imm' in the assembly syntax represents the immediate s after sign extension.

CMP/GE Rm, Rn

Description: This instruction sets the T-bit if the signed value of R_n is greater than or equal to the signed value of R_m , otherwise it clears the T-bit.

Operation:

CMP/GE Rm, Rn

| | | | | | | | | | | | | | | | |
|------|----|----|---|---|---|---|---|---|--|--|--|------|--|--|--|
| 0011 | | | | n | | | | m | | | | 0011 | | | |
| 15 | 12 | 11 | 8 | 7 | 4 | 3 | 0 | | | | | | | | |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
t ← INT (op2 ≥ op1);  
T ← Bit(t);
```

Note:

CMP/GT Rm, Rn

Description: This instruction sets the T-bit if the signed value of R_n is greater than the signed value of R_m , otherwise it clears the T-bit.

Operation:

CMP/GT Rm, Rn

| | | | | | | | | | | | | | | | |
|------|----|----|--|---|---|--|--|---|---|--|--|------|--|--|--|
| 0011 | | | | n | | | | m | | | | 0111 | | | |
| 15 | 12 | 11 | | 8 | 7 | | | 4 | 3 | | | 0 | | | |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
t ← INT (op2 > op1);  
T ← Bit(t);
```

Note:

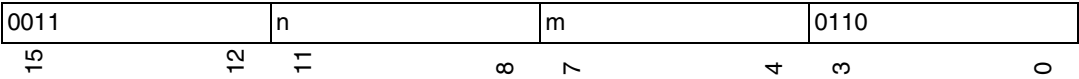


CMP/HI Rm, Rn

Description: This instruction sets the T-bit if the unsigned value of R_n is greater than the unsigned value of R_m , otherwise it clears the T-bit.

Operation:

CMP/HI Rm, Rn



```
op1 ← ZeroExtend32(SignExtend32(Rm));  
op2 ← ZeroExtend32(SignExtend32(Rn));  
t ← INT (op2 > op1);  
T ← Bit(t);
```

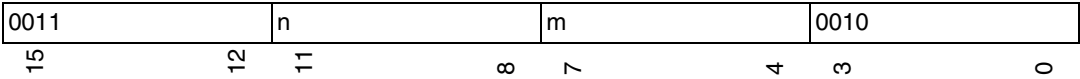
Note:

CMP/HS Rm, Rn

Description: This instruction sets the T-bit if the unsigned value of R_n is greater than or equal to the unsigned value of R_m , otherwise it clears the T-bit.

Operation:

CMP/HS Rm, Rn



```
op1 ← ZeroExtend32(SignExtend32(Rm));  
op2 ← ZeroExtend32(SignExtend32(Rn));  
t ← INT (op2 ≥ op1);  
T ← Bit(t);
```

Note:



CMP/PL Rn

Description: This instruction sets the T-bit if the signed value of R_n is greater than 0, otherwise it clears the T-bit.

Operation:

CMP/PL Rn

| | | |
|------|-------|----------|
| 0100 | n | 00010101 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← SignExtend32(Rn);  
t ← INT (op1 > 0);  
T ← Bit(t);
```

Note:

CMP/PZ Rn

Description: This instruction sets the T-bit if the signed value of R_n is greater than or equal to 0, otherwise it clears the T-bit.

Operation:

CMP/PZ Rn

| | | |
|------|-------|----------|
| 0100 | n | 00010001 |
| 15 | 12 11 | 8 7 0 |

$$\begin{aligned} \text{op1} &\leftarrow \text{SignExtend}_{32}(R_n); \\ t &\leftarrow \text{INT}(\text{op1} \geq 0); \\ T &\leftarrow \text{Bit}(t); \end{aligned}$$

Note:

CMP/STR Rm, Rn

Description: This instruction sets the T-bit if any byte in R_n has the same value as the corresponding byte in R_m , otherwise it clears the T-bit.

Operation:

CMP/STR Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
temp ← op1 ⊕ op2;
t ← INT (temp< 0 FOR 8 > = 0);
t ← (INT (temp< 8 FOR 8 > = 0)) ∨ t;
t ← (INT (temp< 16 FOR 8 > = 0)) ∨ t;
t ← (INT (temp< 24 FOR 8 > = 0)) ∨ t;
T ← Bit(t);

```

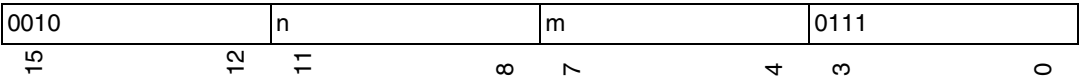
Note:

DIV0S Rm, Rn

Description: This instruction initializes the divide-step state for a signed division. The Q-bit is initialized with the sign-bit of the dividend, and the M-bit with the sign-bit of the divisor. The T-bit is initialized to 0 if the Q-bit and the M-bit are the same, otherwise it is initialized to 1.

Operation:

DIV0S Rm, Rn



```
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
q ← op2 < 31 FOR 1 >;
m ← op1 < 31 FOR 1 >;
t ← m ⊕ q;
Q ← Bit(q);
M ← Bit(m);
T ← Bit(t);
```

Note:

DIV0U

Description: This instruction initializes the divide-step state for an unsigned division. The Q-bit, M-bit and T-bit are all set to 0.

Operation:

DIV0U

00000000000011001

150

q ← 0;
m ← 0;
t ← 0;
Q ← Bit (q);
M ← Bit (m);
T ← Bit (t);

Note:

DIV1 Rm, Rn

Description: This instruction is used to perform a single-bit divide-step for the division of a dividend held in R_n by a divisor held in R_m . The Q-bit, M-bit and T-bit are used to hold additional state through a divide-step sequence. Each DIV1 consumes 1 bit of the dividend from R_n , and produces 1 bit of result. The divide initialization and step instructions do not detect divide-by-zero nor overflow. If required, these cases should be checked using additional instructions.

Operation:

DIV1 Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 0100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
q ← ZeroExtend1(Q);
m ← ZeroExtend1(M);
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
oldq ← q;
q ← op2 < 31 FOR 1 >;
op2 ← ZeroExtend32(op2 << 1) ∨ t;
IF (oldq = m)
    op2 ← op2 - op1;
ELSE
    op2 ← op2 + op1;
q ← (q ⊕ m) ⊕ op2 < 32 FOR 1 >;
t ← 1 - (q ⊕ m);
Rn ← Register(op2);
Q ← Bit(q);
T ← Bit(t);
```

Note:

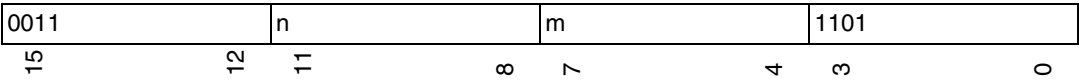


DMULS.L Rm, Rn

Description: This instruction multiplies the signed 32-bit value held in R_m with the signed 32-bit value held in R_n to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

Operation:

DMULS.L Rm, Rn



```
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
mac ← op2 × op1;
mac1 ← mac;
mach ← mac >> 32;
MACL ← ZeroExtend32(mac1);
MACH ← ZeroExtend32(mach);
```

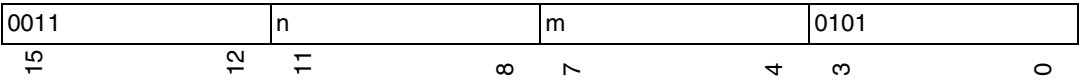
Note:

DMULU.L Rm, Rn

Description: This instruction multiplies the unsigned 32-bit value held in R_m with the unsigned 32-bit value held in R_n to give a full 64-bit result. The lower half of the result is placed in MACL and the upper half in MACH.

Operation:

DMULU.L Rm, Rn



```
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
mac ← op2 × op1;
mac1 ← mac;
mach ← mac >> 32;
MACL ← ZeroExtend32(mac1);
MACH ← ZeroExtend32(mach);
```

Note:



DT Rn

Description: This instruction subtracts 1 from R_n and placed the result in R_n . The T-bit is set if the result is zero, otherwise the T-bit is cleared.

Operation:

DT Rn

| | | |
|------|----|----------|
| 0100 | n | 00010000 |
| 15 | 11 | 0 |

```
op1 ← SignExtend32(Rn);  
op1 ← op1 - 1;  
t ← INT (op1 = 0);  
Rn ← Register(op1);  
T ← Bit(t);
```

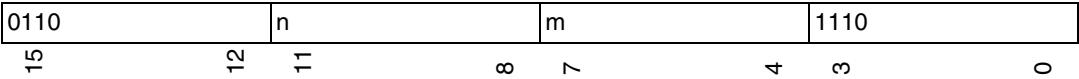
Note:

EXTS.B Rm, Rn

Description: This instruction reads the 8 least significant bits of R_m , sign-extends, and places the result in R_n .

Operation:

EXTS.B Rm, Rn



```
op1 ← SignExtend8(Rm) ;  
op2 ← op1 ;  
Rn ← Register(op2) ;
```

Note:

EXTS.W Rm, Rn

Description: This instruction reads the 16 least significant bits of R_m , sign-extends, and places the result in R_n .

Operation:

EXTS.W Rm, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 1111 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend16(Rm) ;  
op2 ← op1 ;  
Rn ← Register (op2) ;
```

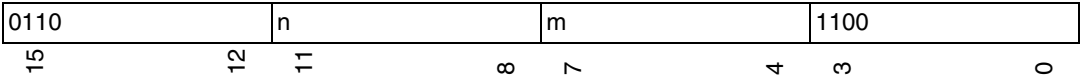
Note:

EXTU.B Rm, Rn

Description: This instruction reads the 8 least significant bits of R_m , zero-extends, and places the result in R_n .

Operation:

EXTU.B Rm, Rn



```
op1 ← ZeroExtend8(Rm) ;  
op2 ← op1 ;  
Rn ← Register(op2) ;
```

Note:



EXTU.W Rm, Rn

Description: This instruction reads the 16 least significant bits of R_m , zero-extends, and places the result in R_n .

Operation:

EXTU.W Rm, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 1101 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

| |
|--|
| $op1 \leftarrow ZeroExtend_{16}(R_m);$ $op2 \leftarrow op1;$ $R_n \leftarrow Register(op2);$ |
|--|

Note:

FABS DRn

Description: This floating-point instruction computes the absolute value of a double-precision floating-point number. It reads DR_n, clears the sign bit and places the result in DR_n.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FABS DRn

| | | |
|------|----|-----------|
| 1111 | n | 001011101 |
| 15 | 31 | 39 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```
sr ← ZeroExtend32(SR);
op1 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← FABS_D(op1);
DR2n ← FloatRegister64(op1);
```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.



FABS FRn

Description: This floating-point instruction computes the absolute value of a single-precision floating-point number. It reads FR_n , clears the sign bit and places the result in FR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FABS FRn

| | | |
|------|----|----------|
| 1111 | n | 01011101 |
| 15 | 12 | 0 |

| |
|--|
| Available only when PR=0 |
| <pre> sr ← ZeroExtend₃₂(SR); op1 ← FloatValue₃₂(FR_n); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; op1 ← FABS_S(op1); FR_n ← FloatRegister₃₂(op1); </pre> |

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FADD DRm, DRn

Description: This floating-point instruction performs a double-precision floating-point addition. It adds DR_m to DR_n and places the result in DR_n . The rounding mode is determined by FPSCR.RM.

Operation:

FADD DRm, DRn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00000 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);
```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FADD FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point addition. It adds FR_m to FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FADD FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0000 |
| 15 | 12 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FADD_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FADD special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are differently signed infinities.
3. Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
4. Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

| op1 → ↓ op2 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|------|------|------|------|--------------------|------|------|
| +, -NORM | ADD | op2 | op2 | +INF | -INF | n/a | qNaN | qNaN |
| +0 | op1 | +0 | +0 | +INF | -INF | n/a | qNaN | qNaN |
| -0 | op1 | +0 | -0 | +INF | -INF | n/a | qNaN | qNaN |
| +INF | +INF | +INF | +INF | +INF | qNaN | n/a | qNaN | qNaN |
| -INF | -INF | -INF | -INF | qNaN | -INF | n/a | qNaN | qNaN |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a | n/a | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'ADD' case is described by the IEEE754 specification.

FCMP/EQ DRm, DRn

Description: This floating-point instruction performs a double-precision floating-point equality comparison. It sets the T-bit to 1 if DR_m is equal to DR_n, and otherwise sets the T-bit to 0.

Operation:

FCMP/EQ DRm, DRn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00100 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPEQ_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);
    
```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FCMP/EQ FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point equality comparison. It sets the T-bit to 1 if FR_m is equal to FR_n, and otherwise sets the T-bit to 0.

Operation:

FCMP/EQ FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0100 |
| 15 | 12 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPEQ_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception

FCMP/EQ special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a signaling NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

| op1 → ↓ op2 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|-------|-------|-------|-------|--------------------|-------|-------|
| +, -NORM | CMPEQ | false | false | false | false | false | false | false |
| +0 | false | true | true | false | false | false | false | false |
| -0 | false | true | true | false | false | false | false | false |
| +INF | false | false | false | true | false | false | false | false |
| -INF | false | false | false | false | true | false | false | false |
| +, -DNORM | false | false | false | false | false | CMPEQ | false | false |
| qNaN | false | false | false | false | false | false | false | false |
| sNaN | false | false | false | false | false | false | false | false |

Invalid operations are indicated by light shading and raise an exception if enabled.
FPU disabled cases are not shown.

The behavior of the normal 'CMPEQ' case is described by the IEEE754 specification.

FCMP/GT DR_m, DR_n

Description: This floating-point instruction performs a double-precision floating-point greater-than comparison. It sets the T-bit to 1 if DR_n is greater than DR_m, and otherwise sets the T-bit to 0.

Operation:

FCMP/GT DR_m, DR_n

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00101 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPGT_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FCMP/GT FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point greater-than comparison. It sets the T-bit to 1 if FR_n is greater than FR_m, and otherwise sets the T-bit to 0.

Operation:

FCMP/GT FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0101 |
| 15 | 12 | 8 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
t, fps ← FCMPGT_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPSCR ← ZeroExtend32(fps);
T ← Bit(t);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FCMP/GT special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a NaN.

If the instruction does not raise an exception, a result is generated according to the following table.

| op2 → ↓ op1 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|-------|-------|-------|-------|--------------------|-------|-------|
| +, -NORM | CMPGT | CMPGT | CMPGT | true | false | CMPGT | false | false |
| +0 | CMPGT | false | false | true | false | CMPGT | false | false |
| -0 | CMPGT | true | false | true | false | CMPGT | false | false |
| +INF | false | false | false | false | false | false | false | false |
| -INF | true | true | true | true | false | true | false | false |
| +, -DNORM | CMPGT | CMPGT | CMPGT | true | false | CMPGT | false | false |
| qNaN | false | false | false | false | false | false | false | false |
| sNaN | false | false | false | false | false | false | false | false |

Invalid operations are indicated by light shading and raise an exception if enabled.
FPU disabled cases are not shown.

The behavior of the normal 'CMPGT' case is described by the IEEE754 specification.

FCNVDS DR_m, FPUL

Description: This floating-point instruction performs a double-precision to single-precision floating-point conversion. It reads a double-precision value from DR_m, converts it to single-precision and places the result in FPUL. The rounding mode is determined by FPSCR.RM.

Operation:

FCNVDS DR_m, FPUL

| | | |
|---|------------------|------------------|
| 1111 | m | 010111101 |
| $\overline{r_0}$ | $\overline{r_1}$ | $\overline{r_2}$ |
| Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300. | | |
| <pre> sr ← ZeroExtend₃₂(SR); fps ← ZeroExtend₃₂(FPSCR); op1 ← FloatValue₆₄(DR_{2m}); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; fpul, fps ← FCNV_DS(op1, fps); IF (FpuEnableV(fps) AND FpuCauseV(fps)) THROW FPUExc, fps; IF (FpuCauseE(fps)) THROW FPUExc, fps; IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps)) THROW FPUExc, fps; FPSCR ← ZeroExtend₃₂(fps); FPUL ← ZeroExtend₃₂(fpul); </pre> | | |

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FCNVSD FPUL, DRn

Description: This floating-point instruction performs a single-precision to double-precision floating-point conversion. It reads a single-precision value from FPUL, converts it to double-precision and places the result in DR_n. FPSCR.RM has no effect since the conversion is exact.

Operation:

FCNVSD FPUL, DRn

| | | |
|------|---|-----------|
| 1111 | n | 010101101 |
| 15 | 2 | 0 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FCNV_SD(fpul, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
DR2n ← FloatRegister64(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FCNVDS and FCNVSD special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if the input is a signaling NaN.
3. Error: an FPU error is signaled if FPSCR.DN is zero and the input is a denormalized number.
4. Inexact, underflow and overflow: these are checked together and can be signaled in combination. These cases occur for FCNVDS but not for FCNVSD. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised for FCNVDS regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 →

| | | | | | | | | |
|-------|-------|----|----|------|------|--------------------|------|------|
| +NORM | -NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
| CNV | CNV | +0 | -0 | +INF | -INF | n/a | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'CNV' case is described by the IEEE754 specification.

FDIV DR_m, DR_n

Description: This floating-point instruction performs a double-precision floating-point division. It divides DR_n by DR_m and places the result in DR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FDIV DR_m, DR_n

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00011 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FDIV_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FDIV FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point division. It divides FR_n by FR_m and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FDIV FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0011 |
| 15 | 12 | 8 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FDIV_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FDIV special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the division is of a zero by a zero, or of an infinity by an infinity.
3. Divide-by-zero: a divide-by-zero is signaled if the divisor is zero and the dividend is a finite non-zero number.
4. Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either of the following conditions is true: the divisor is a denormalized number, or the dividend is a denormalized number and the divisor is not a zero.
5. Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are

requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated as follows:

| op2 → ↓ op1 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|--------|--------|----------------|----------------|--------------------|------|------|
| +, -NORM | DIV | +0, -0 | -0, +0 | +INF, - INF | -INF, + INF | n/a | qNaN | qNaN |
| +0 | +INF, -INF | qNaN | qNaN | +INF | -INF | +INF, -INF | qNaN | qNaN |
| -0 | -INF, +INF | qNaN | qNaN | -INF | +INF | -INF, +INF | qNaN | qNaN |
| +INF | +0, -0 | +0 | -0 | qNaN | qNaN | n/a | qNaN | qNaN |
| -INF | -0, +0 | -0 | +0 | qNaN | qNaN | n/a | qNaN | qNaN |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a | n/a | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations and divide-by-zero are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'DIV' case is described by the IEEE754 specification.

FIPR FVm, FVn

Description: This floating-point instruction computes the dot-product of two vectors, FV_m and FV_n, and places the result in element 3 of FV_n. Each vector contains four single-precision floating-point values. The dot-product is specified as:

$$FR_{n+3} = \sum_{i=0}^3 FR_{m+i} \times FR_{n+i}$$

This is an approximate computation. The specified error in the result value:

$$\text{spec_error} = \begin{cases} 0 & \text{if}(epm = ez) \\ 2^{epm-24} + 2^{E-24+rm} & \text{if}(epm \neq ez) \end{cases}$$

where

$$rm = \begin{cases} 0 & \text{if}(\text{round} - \text{to} - \text{nearest}) \\ 1 & \text{if}(\text{round} - \text{to} - \text{zero}) \end{cases}$$

E = unbiased exponent value of the result

ez < -252

epm = max (ep₀, ep₁, ep₂, ep₃)

ep_i = pre-normalized exponent of the product FR_{m+i} and FR_{n+i}

eFR_{m+i} = biased exponent value of FR_{m+i}

eFR_{n+i} = biased exponent value of FR_{n+i}

$$ep_i = \begin{cases} ez & \text{if}((FR_{m+i} = 0.0) \text{OR} (FR_{n+i} = 0.0)) \\ \max(eFR_{m+i}, 1) + \max(eFR_{n+i}, 1) - 254 & \text{otherwise} \end{cases}$$

Operation:

FIPR FVm, FVn

| | | | |
|------|---|----|----------|
| 1111 | n | m | 11101101 |
| 15 | 2 | 10 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValueVector32(FV4m);
op2 ← FloatValueVector32(FV4n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2[3], fps ← FIPR_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FV4n ← FloatRegisterVector32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FIPR special cases:

FIPR is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FIPR calculation. This instruction never signals an FPU error.
- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if any of the following arise:
 - Any of the inputs is a signaling NaN.
 - Multiplication of a zero by an infinity.
 - Addition of differently signed infinities where none of the inputs is a qNaN.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results.

3. Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate inner-product algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

Each of the 4 pairs of multiplication operands (op1 and op2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

| op1 → ↓ op2 | +, -NORM, +, -DNORM | +0 | -0 | +INF | -INF | qNaN | sNaN |
|-----------------------|------------------------|--------|--------|----------------|---------------|------|------|
| +, -NORM +, -DNORM | FIPRMUL | +0, -0 | -0, +0 | +INF, - INF | -INF, +INF | qNaN | qNaN |
| +0 | +0, -0 | +0 | -0 | qNaN | qNaN | qNaN | qNaN |
| -0 | -0, +0 | -0 | +0 | qNaN | qNaN | qNaN | qNaN |
| +INF | +INF, -INF | qNaN | qNaN | +INF | -INF | qNaN | qNaN |
| -INF | -INF, +INF | qNaN | qNaN | -INF | +INF | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FIPRMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labelled intermediate 0 to 3) are summed:

| ↓ intermediate 2 | intermediate 0 → intermediate 1 → ↓ intermediate 3 | FIPRMUL, +0, -0 | | | +INF | | | -INF | | |
|------------------------|--|--------------------|------|------|--------------------|------|------|--------------------|------|------|
| | | FIPRMUL, +0, -0 | +INF | -INF | FIPRMUL, +0, -0 | +INF | -INF | FIPRMUL, +0, -0 | +INF | -INF |
| FIPRMUL, +0, -0 | FIPRMUL, +0, -0 | FIPRADD | +INF | -INF | +INF | +INF | qNaN | -INF | qNaN | -INF |
| | +INF | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | -INF | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |
| +INF | FIPRMUL, +0, -0 | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | +INF | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | -INF | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| -INF | FIPRMUL, +0, -0 | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |
| | +INF | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| | -INF | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |

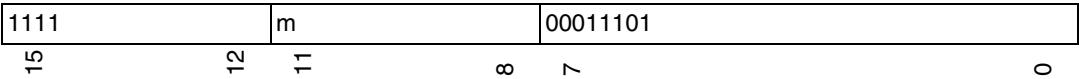
Inexact is signaled in the 'FIPRADD' case. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

FLDS FRm, FPUL

Description: This floating-point instruction copies FR_m to FPUL.
This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

Operation:

FLDS FRm, FPUL



```
sr ← ZeroExtend32(SR);
op1 ← FloatValue32(FRm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul ← op1;
FPUL ← ZeroExtend32(fpul);
```

Exceptions: SLOTFPUDIS, FPUDIS
Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FLDI0 FRn

Description: This floating-point instruction loads a constant representing the single-precision floating-point value of 0.0 into FR_n.

Operation:

FLDI0 FRn

| | | |
|------|------|----------|
| 1111 | n | 10001101 |
| 15 | 1211 | 81 |

Available only when PR=0

sr ←ZeroExtend₃₂(SR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
 THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
 THROW FPUDIS;
op1 ←0x00000000;
FR_n ←FloatRegister₃₂(op1);

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FLDI1 FRn

Description: This floating-point instruction loads a constant representing the single-precision floating-point value of 1.0 into FR_n.

Operation:

FLDI1 FRn

| | | |
|------|------|----------|
| 1111 | n | 10011101 |
| 15 | 1211 | 81 |

Available only when PR=0

sr ←ZeroExtend₃₂(SR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
 THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
 THROW FPUDIS;
op1 ←0x3F800000;
FR_n ←FloatRegister₃₂(op1);

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.



FLOAT FPUL, DRn

Description: This floating-point instruction performs a signed 32-bit integer to double-precision floating-point conversion. It reads a signed 32-bit integer value from FPUL, converts it to a double-precision range and places the result in DR_n. In all cases the provided integer value will be exactly represented in the destination floating-point format. FPSCR.RM has no effect since the conversion is exact.

Operation:

FLOAT FPUL, DRn

| | | |
|------|----|-----------|
| 1111 | n | 000101101 |
| 15 | 31 | 31 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

fpul ← SignExtend32(FPUL);
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FLOAT_LD(fpul, fps);
DR2n ← FloatRegister64(op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FLOAT FPUL, FR_n

Description: This floating-point instruction performs a signed 32-bit integer to single-precision floating-point conversion. It reads a signed 32-bit integer value from FPUL, converts it to a single-precision range and places the result in FR_n. In cases where the integer value cannot be exactly represented in the destination floating-point format, the rounding mode is determined by FPSCR.RM.

Operation:

FLOAT FPUL, FR_n

| | | |
|------|----|----------|
| 1111 | n | 00101101 |
| 15 | 31 | 31 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FLOAT_LS(fpul, fps);
IF (FpuEnableI(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FLOAT special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Inexact: inexact can occur for FLOAT FPUL, FR_n but not for FLOAT FPUL, DR_n. When inexact exceptions are requested by the user, an exception is always raised for FLOAT FPUL, FR_n regardless of whether that condition arose. Overflow and underflow do not occur for either of these instructions.

If the instruction does not raise an exception, the conversion is performed as indicated by the IEEE754 specification.

FMAC FR₀, FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point multiply-accumulate. It multiplies FR₀ by FR_m, adds this intermediate to FR_n and places the result back to FR_n. The multiplication and addition are performed as if the exponent and precision ranges were unbounded, followed by one rounding down to single-precision format. The rounding mode is determined by FPSCR.RM.

Operation:

FMAC FR₀, FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 1110 |
| 15 | 12 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
fr0 ← FloatValue32(FR0);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMAC_S(fr0, op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMAC special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if any of the three inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an

infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

3. Error: an FPU error is signaled if FPSCR.DN is 0 and none of the inputs are a NaN and at least one of the inputs is a denormalized number.
4. Inexact, underflow and overflow: these are checked together and can be signaled in combination. The multiply-accumulate is implemented using a fused-mac algorithm, and these are detected during the conversion of the exactly evaluated intermediate to the single-precision result. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following tables.

Firstly, the operands are checked for sNaN:

| fr0 → op1 → ↓ op2 | other | | sNaN | |
|-------------------------|-------|------|-------|------|
| | other | sNaN | other | sNaN |
| other | | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN |

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, fr0 and op1 are checked for a zero multiplied by an infinity:

| fr0 → ↓ op1 | other | +0 | -0 | +INF | -INF |
|----------------|-------|------|------|------|------|
| other | | | | | |
| +0 | | | | qNaN | qNaN |
| -0 | | | | qNaN | qNaN |
| +INF | | qNaN | qNaN | | |
| -INF | | qNaN | qNaN | | |

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for input qNaN values:

| | | | | |
|-------|-------|------|-------|------|
| fr0 → | other | | qNaN | |
| op1 → | other | qNaN | other | qNaN |
| ↓ op2 | | | | |
| other | | qNaN | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN |

By this stage all operations involving sNaN or qNaN operands have been dealt with. If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, the operands are checked for the addition of differently signed infinities.

| fr0 → | +other | | | | -other | | | | +INF | | | | -INF | | | |
|--------|--------|--------|------|------|--------|--------|------|------|--------|--------|------|------|--------|--------|------|------|
| op1 → | +other | -other | +INF | -INF | +other | -other | +INF | -INF | +other | -other | +INF | -INF | +other | -other | +INF | -INF |
| ↓ op2 | | | | | | | | | | | | | | | | |
| +other | | | | | | | | | | | | | | | | |
| -other | | | | | | | | | | | | | | | | |
| +INF | | | qNaN | | | | qNaN | | | | qNaN | | qNaN | | qNaN | |
| -INF | | | qNaN | | | | | qNaN | qNaN | | qNaN | | | qNaN | | qNaN |

If the result of the previous table is a qNaN, no further analysis is performed. In all other cases, fr0 and op1 are multiplied:

| fr0 → ↓ op1 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM |
|----------------|------------------|--------|--------|------------|------------|--------------------|
| +, -NORM | FULLMUL | +0, -0 | -0, +0 | +INF, -INF | -INF, +INF | n/a |
| +0 | +0, -0 | +0 | -0 | | | n/a |
| -0 | -0, +0 | -0 | +0 | | | n/a |
| +INF | +INF, -INF | | | +INF | -INF | n/a |
| -INF | -INF, +INF | | | -INF | +INF | n/a |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a | n/a |

The empty cells in this table correspond to cases that have already been dealt with. If either source is denormalized, no further analysis is performed. In the 'FULLMUL' case, a multiplication is performed without loss of precision. There is no rounding nor overflow, and this multiplication cannot produce an intermediate infinity.

In the 'FULLMUL', +0, -0, +INF and -INF cases, the 2 addition operands (fr0*op1 and op2) are summed:

| (fr0*op1)→ ↓ op2 | FULLMUL | +0 | -0 | +INF | -INF |
|---------------------|---------|------|------|------|------|
| +, -NORM | FULLADD | op2 | op2 | +INF | -INF |
| +0 | FULLADD | +0 | +0 | +INF | -INF |
| -0 | FULLADD | +0 | -0 | +INF | -INF |
| +INF | +INF | +INF | +INF | +INF | |
| -INF | -INF | -INF | -INF | | -INF |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a |

The two empty cells in this table correspond to cases that have already been dealt with. In the 'FULLADD' cases the fully-precise addition intermediate is rounded to give a single-precision result.

In the above tables, FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

FMOV DRm, DRn

Description: This floating-point instruction reads a pair of single-precision floating-point values from DR_m and copies them to DR_n . This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV DRm, DRn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 01100 |
| 15 | 12 | 11 | 8 | 8 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue64(DR2m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
DR2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV DRm, XDn

Description: This floating-point instruction reads a pair of single-precision floating-point values from DR_m and copies them to XD_n. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV DRm, XDn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 1 | m | 01100 |
| 15 | 12 | 11 | 8 | 4 |

| |
|--|
| Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300. |
| <pre> sr ← ZeroExtend₃₂(SR); op1 ← FloatValue₆₄(DR_{2m}); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; op2 ← op1; XD_{2n} ← FloatRegister₆₄(op2); </pre> |

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV DRm, @Rn

Description: This floating-point instruction stores the contents of the 64-bit register DRm to memory using register indirect with zero-displacement addressing.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV DRm, @Rn

| | | | |
|------|----|---|-------|
| 1111 | n | m | 01010 |
| 15 | 12 | 1 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DRm);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV DRm, @-Rn

Description: This floating-point instruction stores the contents of a 64-bit floating-point registers to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 8 to give the virtual address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV DRm, @-Rn

| | | | |
|------|----|---|-------|
| 1111 | n | m | 01011 |
| 15 | 12 | 1 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 8);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV DRm, @(R0, Rn)

Description: This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . DR_m is written as two consecutive 32-bit values to the virtual address. This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV DRm, @(R0, Rn)

| | | | |
|------|---|---|-------|
| 1111 | n | m | 00111 |
| 15 | 2 | 1 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValue64(DR2m);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

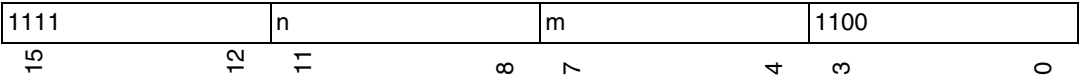
FMOV.S FRm, FRn

Description: This floating-point instruction reads a single-precision floating-point value from FR_m and copies it to FR_n. This is a bit-by-bit copy with no interpretation or conversion of the value.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S FRm, FRn



Available only when SZ=0

```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
FRn ← FloatRegister32(op2);
```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S FR_m, @R_n

Description: This floating-point instruction stores a single-precision floating-point register to memory using register indirect with zero-displacement addressing. The 32-bit value of FR_m is written to the virtual address specified in R_n.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S FR_m, @R_n

| | | | |
|------|----|----|------|
| 1111 | n | m | 1010 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
WriteMemory32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S FR_m, @-R_n

Description: This floating-point instruction stores a single-precision floating-point register to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of FR_m is written to the virtual address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S FR_m, @-R_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 1011 |
| 15 | 12 | 1 | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S FR_m, @(R0, R_n)

Description: This floating-point instruction stores a single-precision floating-point register to memory using register indirect addressing. The virtual address is formed by adding R₀ to R_n. The 32-bit value of FR_m is written to the virtual address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S FR_m, @(R0, R_n)

| | | | |
|------|----|---|------|
| 1111 | n | m | 0111 |
| 15 | 12 | 1 | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValue32(FRm);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
WriteMemory32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV XDm, DRn

Description: This floating-point instruction reads a pair of single-precision floating-point values from XD_m and copies them to DR_n. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV XDm, DRn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 11100 |
| 15 | 12 | 11 | 8 | 8 |
| 15 | 12 | 11 | 8 | 8 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(XD2m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2 ← op1;
DR2n ← FloatRegister64(op2);
```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.



FMOV XDm, XDn

Description: This floating-point instruction reads a pair of single-precision floating-point values from XD_m and copies them to XD_n. This is a bit-by-bit copy with no interpretation or conversion of the values.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV XDm, XDn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 1 | m | 11100 |
| 15 | 12 | 11 | 8 | 4 |
| | | | | 0 |

| |
|--|
| Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300. |
| <pre> sr ← ZeroExtend₃₂(SR); fps ← ZeroExtend₃₂(FPSCR); op1 ← FloatValue₆₄(XD_{2m}); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; op2 ← op1; XD_{2n} ← FloatRegister₆₄(op2); </pre> |

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV XDm, @Rn

Description: This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with zero-displacement addressing. XD_m is written as two consecutive 32-bit values to the virtual address specified in R_n.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV XDm, @Rn

| | | | |
|----------|----------|----------|----------|
| 1111 | n | m | 11010 |
| 15 r5 | 31 r2 | 31 r1 | 31 r0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(XD2m);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV XDm, @-Rn

Description: This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 8 to give the virtual address. XD_m is written as two consecutive 32-bit values to the virtual address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV XDm, @-Rn

| | | | |
|------|----|---|-------|
| 1111 | n | m | 11011 |
| 15 | 12 | 1 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(XD2m);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op2 - 8);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);
op2 ← address;
Rn ← Register(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV XDm, @(R0, Rn)

Description: This floating-point instruction stores a pair of single-precision floating-point registers to memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . XD_m is written as two consecutive 32-bit values to the virtual address.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV XDm, @(R0, Rn)

| | | | |
|------|----|----|-------|
| 1111 | n | m | 10111 |
| 15 | 31 | 31 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← FloatValue64(XD2m);
op2 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op2);
IF (PR=1)
    WriteMemory64(address, op1);
ELSE
    WriteMemoryPair32(address, op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @Rm, DRn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with zero-displacement addressing. Two consecutive 32-bit values are read from the virtual address specified in R_m and loaded into DR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @Rm, DRn

| | | | | |
|------|----|----|---|------|
| 1111 | n | 0 | m | 1000 |
| 15 | 12 | 11 | 8 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
DR2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @Rm+, DRn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with post-increment addressing. Two consecutive 32-bit values are read from the virtual address specified in R_m and loaded into DR_n . R_m is post-incremented by 8.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @Rm+, DRn

| | | | | |
|------|----|---|---|------|
| 1111 | n | 0 | m | 1001 |
| 15 | 12 | 1 | 8 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
op1 ← op1 + 8;
Rm ← Register(op1);
DR2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @(R0, Rm), DRn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . Two consecutive 32-bit values are read from the virtual address and loaded into DR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @(R0, Rm), DRn

| | | | | |
|------|---|---|---|------|
| 1111 | n | 0 | m | 0110 |
| 15 | 2 | 1 | 5 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
DR2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S @Rm, FRn

Description: This floating-point instruction loads a single-precision floating-point register from memory using register indirect with zero-displacement addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into FR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S @Rm, FRn

| | | | |
|------|----|----|------|
| 1111 | n | m | 1000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemory32(address);
FR2n ← FloatRegister32(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S @Rm+, FRn

Description: This floating-point instruction loads a single-precision floating-point register from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into FR_n . R_m is post-incremented by 4.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S @Rm+, FRn

| | | | |
|------|----|----|------|
| 1111 | n | m | 1001 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
op2 ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(op1);
FRn ← FloatRegister32(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV.S @(R0, Rm), FRn

Description: This floating-point instruction loads a single-precision floating-point register from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . A 32-bit value is read from the virtual address and loaded into FR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV.S @(R0, Rm), FRn

| | | | |
|------|----|----|------|
| 1111 | n | m | 0110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

Available only when SZ=0

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
op2 ← ReadMemory32(address);
FRn ← FloatRegister32(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @Rm, XDn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with zero-displacement addressing. Two consecutive 32-bit values are read from the virtual address specified in R_m and loaded into XD_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @Rm, XDn

| | | | | |
|------|----|----|---|------|
| 1111 | n | 1 | m | 1000 |
| 15 | 12 | 11 | 8 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
XD2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @Rm+, XDn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect with post-increment addressing. Two consecutive 32-bit values are read from the virtual address specified in R_m and loaded into XD_n . R_m is post-incremented by 8.

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @Rm+, XDn

| | | | | |
|------|----|----|---|------|
| 1111 | n | 1 | m | 1001 |
| 15 | 12 | 11 | 8 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
op1 ← op1 + 8;
Rm ← Register(op1);
XD2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMOV @(R0, Rm), XDn

Description: This floating-point instruction loads a pair of single-precision floating-point registers from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . Two consecutive 32-bit values are read from the virtual address and loaded into XD_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FMOV @(R0, Rm), XDn

| | | | | |
|------|----|----|---|------|
| 1111 | n | 1 | m | 0110 |
| 15 | 12 | 11 | 8 | 0 |

Available only when SZ=1. When PR=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(r0 + op1);
IF (PR=1)
    op2 ← ReadMemory64(address);
ELSE
    op2 ← ReadMemoryPair32(address)
XD2n ← FloatRegister64(op2);

```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: This instruction can only be executed when SZ=1 as the equivalent single-precision instruction shares the same opcode and is executed when SZ=0. When PR=0 this 64-bit operation behaves as two 32-bit ones, when PR=1 it is handled as a single 64-bit, however, this latter operation is only well-specified on the ST40-300. The distinction is only visible when operating in little-endian mode because when PR=0 it is the upper 32-bits of the 64-bit value which are associated with lower of the two 32-bit memory words.

Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMUL DR_m, DR_n

Description: This floating-point instruction performs a double-precision floating-point multiplication. It multiplies DR_m by DR_n and places the result in DR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FMUL DR_m, DR_n

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00010 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMUL_D(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMUL FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point multiplication. It multiplies FR_m by FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FMUL FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0010 |
| 15 | 12 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FMUL_S(op1, op2, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FMUL special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a signaling NaN, or if this is a multiplication of a zero by an infinity.
3. Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
4. Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

| op1 → ↓ op2 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|--------|--------|------------|------------|--------------------|------|------|
| +, -NORM | MUL | +0, -0 | -0, +0 | +INF, -INF | -INF, +INF | n/a | qNaN | qNaN |
| +0 | +0, -0 | +0 | -0 | qNaN | qNaN | n/a | qNaN | qNaN |
| -0 | -0, +0 | -0 | +0 | qNaN | qNaN | n/a | qNaN | qNaN |
| +INF | +INF, -INF | qNaN | qNaN | +INF | -INF | n/a | qNaN | qNaN |
| -INF | -INF, +INF | qNaN | qNaN | -INF | +INF | n/a | qNaN | qNaN |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a | n/a | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'MUL' case is described by the IEEE754 specification.

FNEG DRn

Description: This floating-point instruction computes the negated value of a double-precision floating-point number. It reads DR_n , inverts the sign bit and places the result in DR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FNEG DRn

| | | |
|------|----|-----------|
| 1111 | n | 001001101 |
| 15 | 12 | 0 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
op1 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← FNEG_D(op1);
DR2n ← FloatRegister64(op1);
    
```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FNEG FRn

Description: This floating-point instruction computes the negated value of a single-precision floating-point number. It FR_n , inverts the sign bit and places the result in FR_n .

This instruction is not considered an arithmetic operation, and it does not signal invalid operations. There are no special floating-point cases for this instruction.

Operation:

FNEG FRn

| | | |
|------|----|----------|
| 1111 | n | 01001101 |
| 15 | 12 | 8 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← FNEG_S(op1);
FRn ← FloatRegister32(op1);

```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FPCHG

Description: This floating-point instruction toggles the FPSCR.PR bit. This has the effect of changing the precision for subsequent floating-point arithmetic operations. When FPSCR.PR becomes 0 subsequent operations are performed with single-precision, when it becomes 1 they are performed with double-precision.

Operation:

FPCHG



| |
|--|
| Available only on ST40-300 |
| <pre>sr ← ZeroExtend₃₂(SR); pr ← ZeroExtend₁(FPSCR.PR); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; pr ← pr ⊕ 1; FPSCR.PR ← Bit(pr);</pre> |

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented on the ST40-300. Execution on other variants will result in unspecified behavior.

FRCHG

Description: This floating-point instruction toggles the FPSCR.FR bit. This has the effect of switching the basic and extended banks of the floating-point register file.

Operation:

FRCHG

| |
|------------------|
| 1111101111111101 |
|------------------|

15

0

| |
|--|
| Available on the ST40-300 regardless of FPSCR.PR setting, and on any variant with an FPU when FPSCR.PR=0 |
|--|

| |
|--|
| <pre> sr ← ZeroExtend₃₂(SR); fr ← ZeroExtend₁(FPSCR.FR); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; fr ← fr ⊕ 1; FPSCR.FR ← Bit(fr); </pre> |
|--|

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present. Execution on other products will result in a FPU disabled exception. The instruction is always available on the ST40-300, but only has well-specified behavior on other variants if executed while FPSCR.PR=0.

FSCA FPUL, DR_n

Description: This floating-point instruction computes the sine and cosine of an angle stored in FPUL. The lower register in DR_n returns the sine of the angle in single-precision floating-point format. The upper register in DR_n returns the cosine of the angle in single-precision floating-point format. The input angle is the amount of rotation expressed as a signed fixed-point number in a 2's complement representation. The value 1 represents an angle of $360^\circ/2^{16}$. The upper 16 bits indicate the number of full rotations and the lower 16 bits indicate the remainder angle between 0° and 360° . This is an approximate computation. The specified error in the result value is:

$$\text{spec_error} = 2^{-21}$$

Operation:

FSCA FPUL, DR_n

| | | |
|------|--------|-----------|
| 1111 | n >> 1 | 011111101 |
| 15 | 12 | 8 |

Available only when PR=0

```

sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1[0], fps ← FSINA_S(fpul, fps);
op1[1], fps ← FCOSA_S(fpul, fps);
IF (FpuEnableI(fps))
    THROW FPUExc, fps;
DRn ← FloatRegisterPair32(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSCA special cases:

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Inexact: this is an approximate instruction and inexact is always signaled. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, the instruction computes an approximate result using an implementation-dependent algorithm.

FSCHG

Description: This floating-point instruction toggles the FPSCR.SZ bit. This has the effect of changing the size of the data transfer for subsequent floating-point loads, stores and moves. Two transfer sizes are available: FPSCR.SZ = 0 indicates 32-bit transfer and FPSCR.SZ = 1 indicates 64-bit transfer.

Operation:

FSCHG

| |
|------------------|
| 1111001111111101 |
|------------------|

15

0

| |
|--|
| Available on the ST40-300 regardless of FPSCR.PR setting, and on any variant with an FPU when FPSCR.PR=0 |
|--|

| |
|--|
| <pre> sr ← ZeroExtend₃₂(SR); sz ← ZeroExtend₁(FPSCR.SZ); IF (FpuIsDisabled(sr) AND IsDelaySlot()) THROW SLOTFPUDIS; IF (FpuIsDisabled(sr)) THROW FPUDIS; sz ← sz ⊕ 1; FPSCR.SZ ← Bit(sz); </pre> |
|--|

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present. Execution on other products will result in a FPU disabled exception. The instruction is always available on the ST40-300, but only has well-specified behavior on other variants if FPSCR.PR=0.

FSQRT DR_n

Description: This floating-point instruction performs a double-precision floating-point square root. It extracts the square root of DR_n and places the result in DR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FSQRT DR_n

| | | |
|------|---|-----------|
| 1111 | n | 001101101 |
| 15 | 2 | 0 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FSQRT_D(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF (FpuEnableI(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSQRT FR_n

Description: This floating-point instruction performs a single-precision floating-point square root. It extracts the square root of FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FSQRT FR_n

| | | |
|------|----|----------|
| 1111 | n | 01101101 |
| 15 | 12 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FSQRT_S(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF (FpuEnableI(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSQRT special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if the input is a signaling NaN, or if this is a square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).
3. Error: an FPU error is signaled if FPSCR.DN is zero and the input is a positive denormalized number.
4. Inexact: only inexact is checked. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, a result is generated according to the following table.

op1 →

| +NORM | -NORM | +0 | -0 | +INF | -INF | +DNORM | -DNORM | qNaN | sNaN |
|-------|-------|----|----|------|------|--------|--------|------|------|
| SQRT | qNaN | +0 | -0 | +INF | qNaN | n/a | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled and inexact cases are not shown.

The behavior of the normal 'SQRT' case is described by the IEEE754 specification.

FSRRA FRn

Description: This floating-point instruction computes the reciprocal of the square root of the value stored in FR_n and places the result in FR_n . This is an approximate computation. The specified error in the result value is:

$$\text{spec_error} = 2^{E-21}, \text{ where } E = \text{unbiased exponent value of the result.}$$

Operation:

FSRRA FRn

| | | |
|------|----|----------|
| 1111 | n | 01111101 |
| 15 | 12 | 0 |

Available only when PR=0

```

sr ← ZeroExtend64(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRFRONT+n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1, fps ← FSRRA_S(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuEnableZ(fps) AND FpuCauseZ(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF (FpuEnableI(fps))
    THROW FPUEXC, fps;
FRn ← FloatRegister32(op1);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSRRA special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if the input is a signaling NaN, or if this is a reciprocal square root of a number less than zero (including negative infinity and negative normalized/denormalized numbers, but excluding negative zero).
3. Divide-by-zero: a divide-by-zero is signaled if this is a reciprocal square root of zero (regardless of the sign of the zero).
4. Error: an FPU error is signaled if FPSCR.DN is 0 and the input is a positive denormalized number.
5. Inexact: this is an approximate instruction and inexact is signaled if this is a reciprocal square root of a positive normalized non-zero finite number. Inexact is

not signaled if the input is a negative normalized number, a zero, an infinity, a denormalized number or a NaN. When inexact exceptions are requested by the user, an exception is always raised regardless of whether that condition arose. Overflow and underflow do not occur.

If the instruction does not raise an exception, a result is generated according to the following table. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

| | | | | | | | | | | |
|-------|-------|-------|------|------|------|------|-------|-------|------|------|
| op1 → | +NORM | -NORM | +0 | -0 | +INF | -INF | +DNRM | -DNRM | qNaN | sNaN |
| | SRRA | qNaN | +INF | -INF | +0 | qNaN | n/a | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations and divide-by-zero are indicated by light shading and raise an exception if enabled. FPU disabled and inexact cases are not shown.

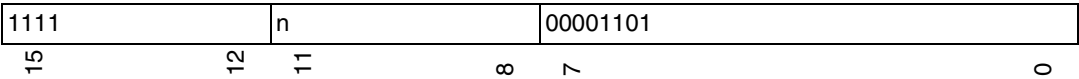
The normal 'SRRA' case uses an implementation-specific algorithm to calculate an approximation of the reciprocal square root of op1.

FSTS FPUL, FRn

Description: This floating-point instruction copies FPUL to FR_n.
This instruction is not considered an arithmetic operation, and it does not signal invalid operations.

Operation:

FSTS FPUL, FRn



```
sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
FRn ← FloatRegister32(op1);
```

Exceptions: SLOTFPUDIS, FPUDIS

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSUB DRm, DRn

Description: This floating-point instruction performs a double-precision floating-point subtraction. It subtracts DR_m from DR_n and places the result in DR_n . The rounding mode is determined by FPSCR.RM.

Operation:

FSUB DRm, DRn

| | | | | |
|------|----|----|---|-------|
| 1111 | n | 0 | m | 00001 |
| 15 | 12 | 11 | 8 | 4 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
op2 ← FloatValue64(DR2n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FSUB_D(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
IF (FpuCauseE(fps))
    THROW FPUEXC, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
DR2n ← FloatRegister64(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSUB FR_m, FR_n

Description: This floating-point instruction performs a single-precision floating-point subtraction. It subtracts FR_m from FR_n and places the result in FR_n. The rounding mode is determined by FPSCR.RM.

Operation:

FSUB FR_m, FR_n

| | | | |
|------|----|---|------|
| 1111 | n | m | 0001 |
| 15 | 12 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
op2 ← FloatValue32(FRn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op2, fps ← FSUB_S(op2, op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
IF (FpuCauseE(fps))
    THROW FPUExc, fps;
IF ((FpuEnableI(fps) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUExc, fps;
FRn ← FloatRegister32(op2);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FSUB special cases:

When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if either input is a signaling NaN, or if the inputs are similarly signed infinities.
3. Error: an FPU error is signaled if FPSCR.DN is zero, neither input is a NaN and either input is a denormalized number.
4. Inexact, underflow and overflow: these are checked together and can be signaled in combination. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, a result is generated according to the following table.

| op2 → ↓ op1 | +NORM, - NORM | +0 | -0 | +INF | -INF | +DNORM, - DNORM | qNaN | sNaN |
|----------------|------------------|------|------|------|------|--------------------|------|------|
| +, -NORM | SUB | SUB | SUB | +INF | -INF | n/a | qNaN | qNaN |
| +0 | op2 | +0 | -0 | +INF | -INF | n/a | qNaN | qNaN |
| -0 | op2 | +0 | +0 | +INF | -INF | n/a | qNaN | qNaN |
| +INF | -INF | -INF | -INF | qNaN | -INF | n/a | qNaN | qNaN |
| -INF | +INF | +INF | +INF | +INF | qNaN | n/a | qNaN | qNaN |
| +, -DNORM | n/a | n/a | n/a | n/a | n/a | n/a | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

FPU error is indicated by heavy shading and always raises an exception. Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled, inexact, underflow and overflow cases are not shown.

The behavior of the normal 'SUB' case is described by the IEEE754 specification.

FTRC DRm, FPUL

Description: This floating-point instruction performs a double-precision floating-point to signed 32-bit integer conversion. It reads a double-precision value from DR_m, converts it to a signed 32-bit integral range and places the result in FPUL. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

Operation:

FTRC DRm, FPUL

| | | |
|------|----|-----------|
| 1111 | m | 000111101 |
| 15 | 31 | 31 |

Available only when PR=1. If SZ=1, the behavior is only well-defined on the ST40-300.

```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue64(DR2m);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FTRC_DL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUEXC, fps;
FPUL ← ZeroExtend32(fpul);
FPSCR ← ZeroExtend32(fps);
```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.



FTRC FR_m, FPUL

Description: This floating-point instruction performs a single-precision floating-point to signed 32-bit integer conversion. It reads a single-precision value from FR_m, converts it to a signed 32-bit integral range and places the result in FPUL. The conversion is achieved by rounding to zero (truncation) with saturation to the limits of the target signed integral range. The value of FPSCR.RM is ignored.

Operation:

FTRC FR_m, FPUL

| | | |
|------|---|----------|
| 1111 | m | 00111101 |
| 15 | 1 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← FloatValue32(FRm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul, fps ← FTRC_SL(op1, fps);
IF (FpuEnableV(fps) AND FpuCauseV(fps))
    THROW FPUExc, fps;
FPSCR ← ZeroExtend32(fps);
FPUL ← ZeroExtend32(fpul);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUExc

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

FTRC special cases:

Regardless of FPSCR.DN, denormalized numbers are treated as 0. These instructions do not cause FPU Error.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if the conversion overflows the target range. This is caused by out-of-range normalized numbers, infinities and NaNs.

If the instruction does not raise an exception, a result is generated according to the following table.

| | | | | | | | | | |
|-------|---------------------|---------------------|----|----|---------------------------------------|---------------------------------------|--------------------|------------------|------------------|
| op1 → | +NORM (in range) | -NORM (in range) | +0 | -0 | +INF or +NORM (out of range) | -INF or -NORM (out of range) | +DNORM, - DNORM | qNaN | sNaN |
| | TRC | TRC | 0 | 0 | +2 ³¹ - 1 | -2 ³¹ | 0 | -2 ³¹ | -2 ³¹ |

Invalid operations are indicated by light shading and raise an exception if enabled. FPU disabled cases are not shown.

The behavior of the normal 'TRC' case is described by the IEEE754 specification, though only the round to zero rounding mode is supported by this instruction.

FTRV XMTRX, FV_n

Description: This floating-point instruction multiplies the matrix, XMTRX, with a vector, FV_n, and places the resulting vector in FV_n. The matrix contains sixteen single-precision floating-point values. The vector contains four single-precision floating-point values. The matrix-vector multiplication is specified as:

$$FR_n = \sum_{i=0}^3 XF_{4i} \times FR_{n+i}$$

$$FR_{n+1} = \sum_{i=0}^3 XF_{1+4i} \times FR_{n+i}$$

$$FR_{n+2} = \sum_{i=0}^3 XF_{2+4i} \times FR_{n+i}$$

$$FR_{n+3} = \sum_{i=0}^3 XF_{3+4i} \times FR_{n+i}$$

This is an approximate computation. The specified error in the p^{th} . element value of the result vector:

$$spec_error_p = \begin{cases} 0 & \text{if}(epm = ez) \\ 2^{epm-24} + 2^{E-24+rm} & \text{if}(epm \neq ez) \end{cases}$$

where

$$rm = \begin{cases} 0 & \text{if}(\text{round} - \text{to} - \text{nearest}) \\ 1 & \text{if}(\text{round} - \text{to} - \text{zero}) \end{cases}$$

E = unbiased exponent value of the result

ez < -252

epm = max (ep₀, ep₁, ep₂, ep₃)

ep_i = pre-normalized exponent of the product XF_{p+4i} and FR_{n+i}

eXF_{p+4i} = biased exponent value of XF_{p+4i}

eFR_{n+i} = biased exponent value of FR_{n+i}

$$ep_i = \begin{cases} ez & \text{if}((XF_{p+4i} = 0.0) \text{OR} (FR_{n+i} = 0.0)) \\ \max(eXF_{p+4i} 1) + \max(eFR_{n+i} 1) - 254 & \text{otherwise} \end{cases}$$

Operation:**FTRV XMTRX, FVn**

| | | |
|------|----|------------|
| 1111 | n | 0111111101 |
| 15 | 12 | 0 |

Available only when PR=0

```

sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
xmtrx ← FloatValueMatrix32(XMTRX);
opl ← FloatValueVector32(FV4n);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
opl, fps ← FTRV_S(xmtrx, opl, fps);
IF (((FpuEnableV(fps) OR FpuEnableI(fps)) OR FpuEnableO(fps)) OR FpuEnableU(fps))
    THROW FPUEXC, fps;
FV4n ← FloatRegisterVector32(opl);
FPSCR ← ZeroExtend32(fps);

```

Exceptions: SLOTFPUDIS, FPUDIS, FPUEXC**Note:** Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.**FTRV special cases:**

FTRV is an approximate instruction. Denormalized numbers are supported:

- When FPSCR.DN is 0, denormalized numbers are treated as their denormalized value in the FTRV calculation. This instruction never signals an FPU error.
- When FPSCR.DN is 1, a positive denormalized number is treated as +0 and a negative denormalized number as -0. This flush-to-zero treatment is applied before exception detection and special case handling.

Exceptional conditions are checked in the order given below. Execution of the instruction is terminated once any check detects an exceptional condition.

1. Disabled: an exception is raised if the FPU is disabled.
2. Invalid: an invalid operation is signaled if any of the inputs is a signaling NaN, there is a multiplication of a zero by an infinity, or there is an addition of differently signed infinities where none of the inputs is a qNaN.

The multiplication is performed with sufficient precision to avoid overflow, and therefore the multiplication of any two finite numbers does not produce an infinity. The multiplication result will be an infinity only if there is a multiplication of an infinity with a normalized number, an infinity with a denormalized number or an infinity with an infinity.

The addition of differently signed infinities is detected if there is (at least) one positive infinity and (at least) one negative infinity in the set of 4 multiplication results in any of the 4 inner-products calculated by this instruction.

This instruction is not capable of checking its inputs for invalid operations and raising an invalid operation exception accordingly. Instead, this instruction always raises an invalid operation exception if this exception is requested by the user. If

this exception is not requested by the user, then qNaN results are correctly produced for invalid operations as described above.

3. Inexact, underflow and overflow: these are checked together and can be signaled in combination. This is an approximate instruction and inexact is signaled except where special cases occur. Precise details of the approximate inner-product algorithm, including the detection of underflow and overflow cases, are implementation dependent. When inexact, underflow or overflow exceptions are requested by the user, an exception is always raised regardless of whether that condition arose.

If the instruction does not raise an exception, results are generated according to the following tables. The special case tables are applied separately with the appropriate vector operands to each of the four inner-products calculated by this instruction.

Each of the 4 pairs of multiplication operands (op1 and op2) is selected from corresponding elements of the two 4-element source vectors and multiplied:

| op1 → ↓ op2 | +, -NORM, +, -DNORM | +0 | -0 | +INF | -INF | qNaN | sNaN |
|-----------------------|------------------------|--------|--------|------------|------------|------|------|
| +, -NORM +, -DNORM | FTRVMUL | +0, -0 | -0, +0 | +INF, -INF | -INF, +INF | qNaN | qNaN |
| +0 | +0, -0 | +0 | -0 | qNaN | qNaN | qNaN | qNaN |
| -0 | -0, +0 | -0 | +0 | qNaN | qNaN | qNaN | qNaN |
| +INF | +INF, -INF | qNaN | qNaN | +INF | -INF | qNaN | qNaN |
| -INF | -INF, +INF | qNaN | qNaN | -INF | +INF | qNaN | qNaN |
| qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| sNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |

If any of the multiplications evaluates to qNaN, then the result of the instruction is qNaN and no further analysis need be performed. In the 'FTRVMUL', +0, -0, +INF and -INF cases, the 4 addition operands (labelled intermediate 0 to 3) are summed:

| | intermediate 0 → intermediate 1 → ↓ intermediate 2 intermediate 3 | FTRVMUL, +0, -0 | | | +INF | | | -INF | | |
|--------------------|---|--------------------|------|------|--------------------|------|------|--------------------|------|------|
| | | FTRVMUL, +0, -0 | +INF | -INF | FTRVMUL, +0, -0 | +INF | -INF | FTRVMUL, +0, -0 | +INF | -INF |
| FTRVMUL, +0, -0 | FTRVMUL, +0, -0 | FTRVADD | +INF | -INF | +INF | +INF | qNaN | -INF | qNaN | -INF |
| | +INF | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | -INF | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |
| +INF | FTRVMUL, +0, -0 | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | +INF | +INF | +INF | qNaN | +INF | +INF | qNaN | qNaN | qNaN | qNaN |
| | -INF | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| -INF | FTRVMUL, +0, -0 | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |
| | +INF | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN | qNaN |
| | -INF | -INF | qNaN | -INF | qNaN | qNaN | qNaN | -INF | qNaN | -INF |

Inexact is signaled in the 'FTRVADD' case. Exception cases are not indicated by shading for this instruction. Where the behavior is not a special case, the instruction computes an approximate result using an implementation-dependent algorithm.

ICBI @Rn

Description: This instruction invalidates the instruction cache block that corresponds to the given address. The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The instruction has no effect on the cache if the virtual address is an uncached one or there is no block in the cache which corresponds to the given virtual address.

The instruction immediately after ICBI is always refetched, regardless of whether the cache is modified.

Operation:

ICBI @Rn

| | | |
|-------|-------------|----------------|
| 0000 | nnnn | 11100011 |
| r_5 | r_2 r_1 | ∞ r_0 |

Available only on the ST40-300

```

op1 ← ZeroExtend32(Rn) ;
IF (IsDelaySlot())
    THROW ILLSLOT;
InstFetch(op1) ;
ICBI(op1) ;
        
```

Exceptions: ILLSLOT, IADDERR, ITLBMULTIHIT, ITLBMISS, EXECPROT

Note: Only implemented on the ST40-300. Execution on other variants will result in unspecified behavior.

The instruction only guarantees that code viewed through the given virtual address is no longer present in the cache. If the invalidated cache block has synonyms present in the cache (see [Section 3.13: Interaction of MMU and cache on page 98](#)), these will remain.

[Table 132](#) below shows the action of the ICBI instruction, depending on the address region where R_n lies and the current translation state of the ST40.

Table 132. ICBI actions and value used for cache tag comparison

| Address region containing Rn | MMUCR.AT | PASCR.SE | Action |
|------------------------------|----------|----------|---|
| U0/P0 or P3 | 0 | 0 | Use Rn directly for tag comparison (ignore bits [31:29]). Invalidate I-cache entry if the tag matches. Re-fetch next instruction. |
| | 0 | 1 | Use Rn directly for tag comparison. Invalidate I-cache entry if the tag matches. Re-fetch next instruction |
| | 1 | n/a | Translate Rn and use the result for tag comparison. If the page is cacheable, invalidate I-cache entry if the tag matches. Re-fetch next instruction. |
| P1 | n/a | 0 | Use Rn directly for tag comparison (ignore bits [31:29]). Invalidate I-cache entry if the tag matches. Re-fetch next instruction. |
| P2 | | 0 | No effect on I-cache. (Instruction re-fetch only.) |
| P1 or P2 | | 1 | Translate Rn and use the result for tag comparison. If the page is cacheable, invalidate I-cache entry if the tag matches. Re-fetch next instruction. |
| P4 | n/a | n/a | No effect on I-cache. (Instruction re-fetch only.) |

The ICBI instruction also guarantees that the cache modification has occurred before the following instruction is fetched. Hence ICBI can be used as a general method of forcing a refetch of the following instruction.

The ITLB lookup procedure is equivalent to that used to fetch instructions from the address in Rn. In particular, if no match is found in the ITLB, the UTLB (or PMB) will be searched. If a match is found in the UTLB (or PMB), the matching entry will automatically be recorded in the ITLB. Hence any of the exceptions associated with attempting to fetch an instruction from the given instruction are possible, that is, ITLBMULTIHIT, ITLBMISS and EXECPROT (See [Section 3.14.2: Coherency mechanisms on ST40-300 series cores on page 103](#) for more details).

The ICBI instruction may not be placed in the delay slot of an instruction. If it is, the ILLSLOT exception is raised.

JMP @Rn

Description: This instruction is a delayed unconditional branch used for jumping to the target address specified in R_n .

Operation:

JMP @Rn

| | | |
|------|-------|----------|
| 0100 | n | 00101011 |
| 15 | 12 11 | 8 7 0 |

```

op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← op1;
delayedpc ← target;
PC' ' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

JSR @Rn

Description: This instruction is a delayed unconditional branch used for jumping to the subroutine starting at the target address specified in R_n . The address of the instruction immediately following the delay slot is copied to PR to indicate the return address.

Operation:

JSR @Rn

| | | |
|------|----|----------|
| 0100 | n | 00001011 |
| 15 | 12 | 8 |

```

pc ← SignExtend32(PC);
op1 ← SignExtend32(Rn);
IF (IsDelaySlot())
    THROW ILLSLOT;
pr ← pc + 4;
target ← op1;
delayedpc ← target;
PR ← Register(pr);
PC' ← Register(delayedpc);

```

Exceptions: ILLSLOT

Note: The delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

LDC Rm, GBR

Description: This instruction copies R_m to GBR.

Operation:

LDC Rm, GBR

| | | |
|------|-------|----------|
| 0100 | m | 00011110 |
| 15 | 12 11 | 8 7 0 |

$op1 \leftarrow \text{SignExtend}_{32}(R_m);$
 $gbr \leftarrow op1;$
 $GBR \leftarrow \text{Register}(gbr);$

Note:

LDC Rm, SR

Description: This instruction copies R_m to SR. This is a privileged instruction.

Operation:

LDC Rm, SR

| | | |
|------|-------|----------|
| 0100 | m | 00001110 |
| 15 | 12 11 | 8 7 0 |

```
md ← ZeroExtend1(MD);
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
sr ← op1;
SR ← Register(sr);
```

Exceptions: RESINST

Note:

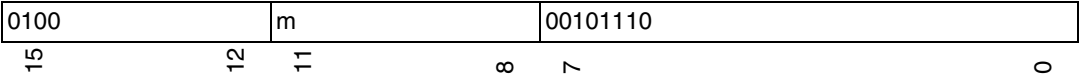


LDC Rm, VBR

Description: This instruction copies R_m to VBR. This is a privileged instruction.

Operation:

LDC Rm, VBR



```
md ←ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
op1 ←SignExtend32(Rm);  
vbr←op1;  
VBR ←Register(vbr);
```

Exceptions: RESINST

Note:

LDC Rm, SSR

Description: This instruction copies R_m to SSR. This is a privileged instruction.

Operation:

LDC Rm, SSR

| | | |
|------|-------|----------|
| 0100 | m | 00111110 |
| 15 | 12 11 | 8 7 0 |

```
md ←ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
op1 ←SignExtend32(Rm);  
ssr ←op1;  
SSR ←Register(ssr);
```

Exceptions: RESINST

Note:

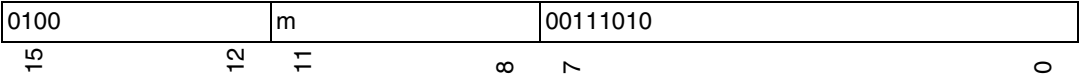


LDC Rm, SGR

Description: This instruction copies R_m to SGR. This is a privileged instruction.

Operation:

LDC Rm, SGR



```
md ←ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
op1 ←SignExtend32(Rm);  
sgr←op1;  
SGR ←Register(sgr);
```

Exceptions: RESINST

Note:

LDC Rm, SPC

Description: This instruction copies R_m to SPC. This is a privileged instruction.

Operation:

LDC Rm, SPC

| | | |
|------|-------|----------|
| 0100 | m | 01001110 |
| 15 | 12 11 | 8 7 0 |

```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ←SignExtend32(Rm);
spc ←op1;
SPC ←Register(spc);
```

Exceptions: RESINST

Note:

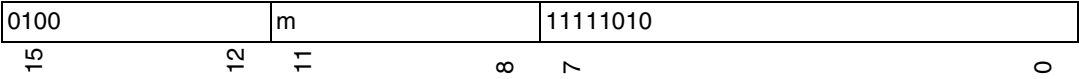


LDC Rm, DBR

Description: This instruction copies R_m to DBR. This is a privileged instruction.

Operation:

LDC Rm, DBR



```
md ← ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
op1 ← SignExtend32(Rm);  
dbr ← op1;  
DBR ← Register(dbr);
```

Exceptions: RESINST

Note:

LDC Rm, Rn_BANK

Description: This instruction copies R_m to Rn_BANK . This is a privileged instruction.

Operation:

LDC Rm, Rn_BANK

| | | | | | | | | |
|------|----|----|---|---|---|---|------|---|
| 0100 | | m | | 1 | n | | 1110 | |
| 15 | 12 | 11 | 8 | 7 | 6 | 4 | 3 | 0 |

```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
opl ←SignExtend32(Rm);
rn_bank←opl;
Rn_BANK ←Register(rn_bank);
```

Exceptions: RESINST

Note:



LDC.L @Rm+, GBR

Description: This instruction loads GBR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into GBR. R_m is post-incremented by 4.

Operation:

LDC.L @Rm+, GBR

| | | |
|------|-------|----------|
| 0100 | m | 00010111 |
| 15 | 12 11 | 8 7 0 |

```

op1 ← SignExtend32( $R_m$ );
address ← ZeroExtend32(op1);
gbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
 $R_m$  ← Register(op1);
GBR ← Register(gbr);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:

LDC.L @Rm+, SR

Description: This instruction loads SR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into SR. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, SR

| | | |
|------|----|----------|
| 0100 | m | 00000111 |
| 15 | 12 | 0 |

```
md ←ZeroExtend1(MD);
IF (IsDelaySlot())
    THROW ILLSLOT;
IF (md = 0)
    THROW RESINST;
op1 ←SignExtend32(Rm);
address ←ZeroExtend32(op1);
sr ←SignExtend32(ReadMemory32(address));
op1 ←op1 + 4;
Rm ←Register(op1);
SR ←Register(sr);
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:

LDC.L @Rm+, VBR

Description: This instruction loads VBR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into VBR. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, VBR

| | | |
|------|----|----------|
| 0100 | m | 00100111 |
| 15 | 12 | 0 |

```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
vbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
VBR ← Register(vbr);
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

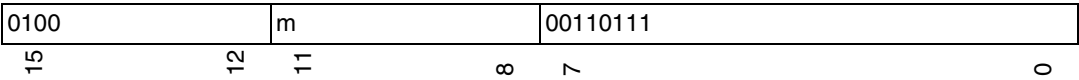
Note:

LDC.L @Rm+, SSR

Description: This instruction loads SSR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into SSR. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, SR



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
ssr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SSR ← Register(ssr);
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:



LDC.L @Rm+, SGR

Description: This instruction loads SGR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into SGR. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, SGR

| | | |
|------|----|----------|
| 0100 | m | 00110110 |
| 15 | 12 | 0 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
sgr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SGR ← Register(sgr);

```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:

LDC.L @Rm+, SPC

Description: This instruction loads SPC from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into SPC. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, SPC

| | | |
|------|----|----------|
| 0100 | m | 01000111 |
| 15 | 12 | 0 |

```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
spc ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
SPC ← Register(spc);
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note:

LDC.L @Rm+, DBR

Description: This instruction loads DBR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into DBR. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, DBR

| | | |
|------|----|----------|
| 0100 | m | 11110110 |
| 15 | 12 | 0 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
dbr ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
DBR ← Register(dbr);
    
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

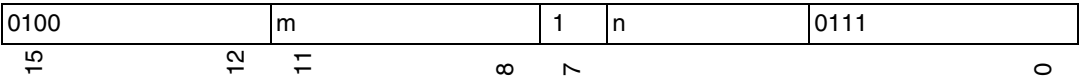
Note:

LDC.L @Rm+, Rn_BANK

Description: This instruction loads Rn_BANK from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into Rn_BANK. R_m is post-incremented by 4. This is a privileged instruction.

Operation:

LDC.L @Rm+, Rn_BANK



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
rn_bank ← SignExtend32(ReadMemory32(address));
op1 ← op1 + 4;
Rm ← Register(op1);
Rn_BANK ← Register(rn_bank);
```

Exceptions: RESINST, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:



LDS Rm, FPSCR

Description: This floating-point instruction copies R_m to FPSCR. The setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

Operation:

LDS Rm, FPSCR

| | | |
|------|----|----------|
| 0100 | m | 01101010 |
| 15 | 12 | 8 |

```

sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
FPSCR ← ZeroExtend32(fps);
    
```

Exceptions: SLOTFPUDIS, FPUDIS

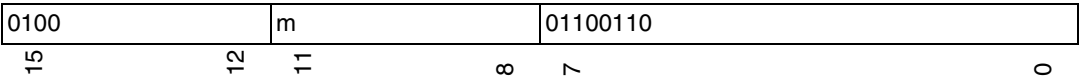
Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.

LDS.L @Rm+, FPSCR

Description: This floating-point instruction loads FPSCR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into FPSCR. R_m is post-incremented by 4. The setting of FPSCR does not cause any floating-point exceptional conditions to be signaled.

Operation:

LDS.L @Rm+, FPSCR



```
sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
value ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(op1);
FPSCR ← ZeroExtend32(fps);
```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: Only implemented when an FPU is present, execution on other products will result in a FPU disabled exception.



LDS Rm, FPUL

Description: This floating-point instruction copies R_m to FPUL.

Operation:

LDS Rm, FPUL

| | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|
| 0100 | | | | m | | | | 01011010 | | | |
| r5 | | | | r1 | | | | 0 | | | |

```
sr ←ZeroExtend32(SR);
op1 ←SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
fpul ←op1;
FPUL ←ZeroExtend32(fpul);
```

Exceptions: SLOTFPUDIS, FPUDIS

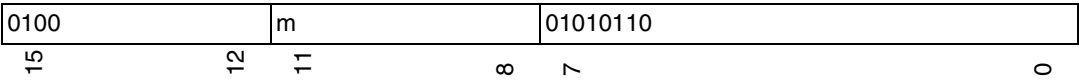
Note:

LDS.L @Rm+, FPUL

Description: This floating-point instruction loads FPUL from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into FPUL. R_m is post-incremented by 4.

Operation:

LDS.L @Rm+, FPUL



```
sr ← ZeroExtend32(SR);
op1 ← SignExtend32(Rm);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1);
fpul ← ReadMemory32(address);
op1 ← op1 + 4;
Rm ← Register(op1);
FPUL ← ZeroExtend32(fpul);
```

Exceptions: SLOTFPUDIS, FPUDIS, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:

LDS Rm, MACH

Description: This instruction copies R_m to MACH.

Operation:

LDS Rm, MACH

| | | |
|------|-------|----------|
| 0100 | m | 00001010 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← SignExtend32(Rm) ;  
mach ← op1 ;  
MACH ← ZeroExtend32(mach) ;
```

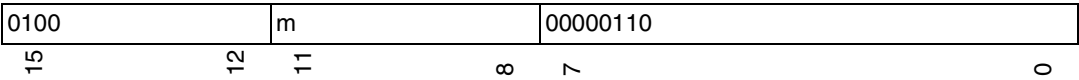
Note:

LDS.L @Rm+, MACH

Description: This instruction loads MACH from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into MACH. R_m is post-incremented by 4.

Operation:

LDS.L @Rm+, MACH



```
op1 ← SignExtend32( $R_m$ );  
address ← ZeroExtend32(op1);  
mach ← SignExtend32(ReadMemory32(address));  
op1 ← op1 + 4;  
 $R_m$  ← Register(op1);  
MACH ← ZeroExtend32(mach);
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:



LDS Rm, MACL

Description: This instruction copies R_m to MACL.

Operation:

LDS Rm, MACL

| | | |
|------|----|----------|
| 0100 | m | 00011010 |
| 15 | 11 | 0 |

```
op1 ← SignExtend32(Rm) ;  
mac1 ← op1 ;  
MACL ← ZeroExtend32(mac1) ;
```

Note:

LDS.L @Rm+, MACL

Description: This instruction loads MACL from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into MACL. R_m is post-incremented by 4.

Operation:

LDS.L @Rm+, MACL

| | | |
|------|----|----------|
| 0100 | m | 00010110 |
| 15 | 12 | 0 |

```
op1 ← SignExtend32(Rm);  
address ← ZeroExtend32(op1);  
mac1 ← SignExtend32(ReadMemory32(address));  
op1 ← op1 + 4;  
Rm ← Register(op1);  
MACL ← ZeroExtend32(mac1);
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:

LDS Rm, PR

Description: This instruction copies R_m to PR.

Operation:

LDS Rm, PR

| | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|
| 0100 | | | | m | | | | 00101010 | | | |
| 15 | | | | 11 | | | | 0 | | | |

```
op1 ← SignExtend32(Rm) ;  
pr ← op1 ;  
PR ← Register(pr) ;
```

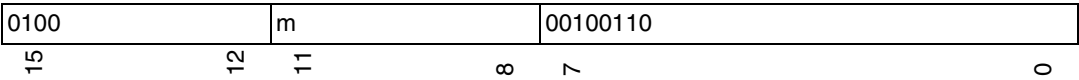
Note:

LDS.L @Rm+, PR

Description: This instruction loads PR from memory using register indirect with post-increment addressing. A 32-bit value is read from the virtual address specified in R_m and loaded into PR. R_m is post-incremented by 4.

Operation:

LDS.L @Rm+, PR



```
op1 ← SignExtend32(Rm);  
address ← ZeroExtend32(op1);  
pr ← SignExtend32(ReadMemory32(address));  
op1 ← op1 + 4;  
Rm ← Register(op1);  
PR ← Register(pr);
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:



LDTLB

Description: This instruction loads the contents of the PTEH/PTEL registers into the UTLB (unified translation lookaside buffer) specified by MMUCR.URC (random counter field in the MMC control register).

LDTLB is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause a RESINST trap.

Operation:

LDTLB

0000000000111000

15

0

```
md ← ZeroExtend1(MD);
IF (md = 0)
  THROW RESINST;
UTLB[MMUCR.URC].ASID ← PTEH.ASID
UTLB[MMUCR.URC].VPN ← PTEH.VPN
UTLB[MMUCR.URC].PPN ← PTEL.PPN
UTLB[MMUCR.URC].SZ ← PTEL.SZ1<<1 + PTEL.SZ0
UTLB[MMUCR.URC].SH ← PTEL.SH
UTLB[MMUCR.URC].PR ← PTEL.PR
UTLB[MMUCR.URC].WT ← PTEL.WT
UTLB[MMUCR.URC].C ← PTEL.C
UTLB[MMUCR.URC].D ← PTEL.D
UTLB[MMUCR.URC].V ← PTEL.V
```

Exceptions: RESINST

Note: As this instruction loads the contents of the PTEH/PTEL registers into a UTLB entry, it should be used either with the MMU disabled, or in the P1 or P2 virtual space with the MMU enabled (see [Chapter 3: Memory management unit \(MMU\) on page 32](#), for details). After this instruction is issued, there must be at least one instruction between the LDTLB instruction and the execution of an instruction from the areas P0, U0, and P3 (that is, via a BRAF, BSRF, JMP, JSR, RTS, or RTE).

MAC.L @Rm+, @Rn+

Description: This instruction reads the signed 32-bit value at the virtual address specified in R_n , and then post-increments R_n by 4. It also reads the signed 32-bit value at the virtual address specified in R_m , and then post-increments R_m by 4. These 2 values are multiplied together to give a 64-bit result, and this result is added to the 64-bit accumulator held in MACL and MACH. This accumulation gives an output with 65 bits of precision.

If the S-bit is 0, the result is the lower 64 bits of the accumulation. If the S-bit is 1, the result is calculated by saturating the accumulation to the signed range $[-2^{47}, 2^{47})$. In either case, the 64-bit result is split into low and high halves, which are placed into MACL and MACH respectively.

Operation:

MAC.L @Rm+, @Rn+

| | | | |
|------|----|----|------|
| 0000 | n | m | 1111 |
| 15 | 31 | 31 | 31 |

```

mac1 ← ZeroExtend32(MACL);
mach ← ZeroExtend32(MACH);
s ← ZeroExtend1(S);
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
m_address ← SignExtend32(Rm);
n_address ← SignExtend32(Rn);
value2 ← SignExtend32(ReadMemory32(ZeroExtend32(n_address)));
n_address ← n_address + 4;
IF (n_field = m_field)
{
    m_address ← m_address + 4;
    n_address ← n_address + 4;
}
value1 ← SignExtend32(ReadMemory32(ZeroExtend32(m_address)));
m_address ← m_address + 4;
mul ← value2 × value1;
mac ← (mach << 32) + mac1;
result ← mac + mul;
IF (s = 1)
    result ← SignedSaturate48(result);
mac1 ← result;
mach ← result >> 32;
Rm ← Register(m_address);
Rn ← Register(n_address);
MACL ← ZeroExtend32(mac1);
MACH ← ZeroExtend32(mach);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.
If R_m and R_n refer to the same register (that is, $m = n$), then this register will be post-incremented twice. The instruction will read two long-words from consecutive memory locations.

MAC.W @Rm+, @Rn+

Description: This instruction reads the signed 16-bit value at the virtual address specified in R_n , and then post-increments R_n by 2. It also reads the signed 16-bit value at the virtual address specified in R_m , and then post-increments R_m by 2. These 2 values are multiplied together to give a 32-bit result.

If the S-bit is 0, the 32-bit multiply result is added to the 64-bit accumulator held in MACL and MACH. This accumulation gives an output with 65 bits of precision, and the result is the lower 64 bits of the accumulation. The result is split into low and high halves, which are placed into MACL and MACH respectively.

If the S-bit is 1, the 32-bit multiply result is added to the 32-bit accumulator held in MACL. This accumulation gives an output with 33 bits of precision, and is saturated to the signed range $[-2^{31}, 2^{31})$, and then placed in MACL. If the accumulation overflows this signed range, then MACH is set to 1 to denote overflow otherwise MACH is unchanged.

Operation:

MAC.W @Rm+, @Rn+

| | | | |
|------|---|---|------|
| 0100 | n | m | 1111 |
| 15 | 2 | 1 | 0 |

```

mac1 ← ZeroExtend32(MACL);
mach ← ZeroExtend32(MACH);
s ← ZeroExtend1(S);
m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
m_address ← SignExtend32(Rm);
n_address ← SignExtend32(Rn);
value2 ← SignExtend16(ReadMemory16(ZeroExtend32(n_address)));
n_address ← n_address + 2;
IF (n_field = m_field)
{
    m_address ← m_address + 2;
    n_address ← n_address + 2;
}
value1 ← SignExtend16(ReadMemory16(ZeroExtend32(m_address)));
m_address ← m_address + 2;
mul ← value2 × value1;
IF (s = 1)
{
    temp ← SignExtend32(mac1) + mul;
    mac1 ← SignedSaturate32(mac1);
    IF (mac1 ≠ temp)
        mach ← 1;
}
ELSE
{
    result ← (mach << 32) + mac1 + mul;
    mac1 ← result;
    mach ← result >> 32;
}

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

If R_m and R_n refer to the same register (that is, $m = n$), then this register will be post-incremented twice. The instruction will read two words from consecutive memory locations.

MOV Rm, Rn

Description: This instruction copies the value of R_m to R_n.

Operation:

MOV Rm, Rn

| | | | | | | | | | | | | | |
|------|----|----|--|---|---|--|--|---|---|--|---|------|--|
| 0110 | | | | n | | | | m | | | | 0011 | |
| 15 | 12 | 11 | | 8 | 7 | | | 4 | 3 | | 0 | | |

op1 ←ZeroExtend₃₂(R_m) ;

op2 ←op1;

R_n ←Register (op2) ;

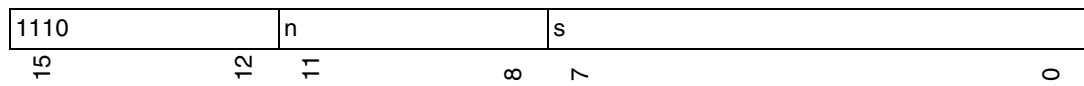
Note:

MOV #imm, Rn

Description: This instruction sign-extends the 8-bit immediate s and places the result in R_n .

Operation:

MOV #imm, Rn



```
imm ← SignExtend8(s);  
op2 ← imm;  
 $R_n$  ← Register(op2);
```

Note: The '#imm' in the assembly syntax represents the immediate s after sign extension.

MOV.B Rm, @Rn

Description: This instruction stores a byte to memory using register indirect with zero-displacement addressing. The virtual address is specified in R_n . The byte to be stored is held in the lowest 8 bits of R_m .

Operation:

MOV.B Rm, @Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2);
WriteMemory8(address, op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note:

MOV.B Rm, @-Rn

Description: This instruction stores a byte to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 1 to give the virtual address. The byte to be stored is held in the lowest 8 bits of R_m .

Operation:

MOV.B Rm, @-Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 1);
WriteMemory8(address, op1);
op2 ← address;
Rn ← Register(op2);
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



MOV.B Rm, @(R0, Rn)

Description: This instruction stores a byte to memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . The byte to be stored is held in the lowest 8 bits of R_m .

Operation:

MOV.B Rm, @(R0, Rn)

| | | | |
|------|----|----|------|
| 0000 | n | m | 0100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory8(address, op1);

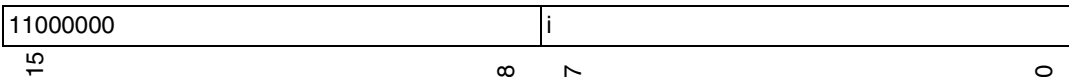
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.B R0, @(disp, GBR)

Description: This instruction stores a byte to memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i. The byte to be stored is held in the lowest 8 bits of R₀.

Operation:**MOV.B R0, @(disp, GBR)**

```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i);
address ← ZeroExtend32(disp + gbr);
WriteMemory8(address, r0);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

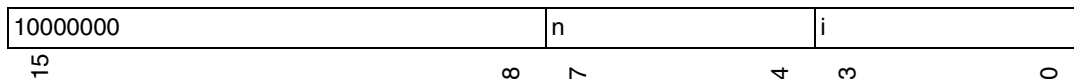
The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.B R0, @(disp, Rn)

Description: This instruction stores a byte to memory using register indirect with displacement addressing. The virtual address is formed by adding R_n and the zero-extended 4-bit immediate i . The byte to be stored is held in the lowest 8 bits of R_0 .

Operation:

MOV.B R0, @(disp, Rn)



```

r0 ← SignExtend32(R0);
disp ← ZeroExtend4(i);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op2);
WriteMemory8(address, r0);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.B @Rm, Rn

Description: This instruction loads a signed byte from memory using register indirect with zero-displacement addressing. The virtual address is specified in R_m . The byte is loaded from the virtual address, sign-extended and placed in R_n .

Operation:

MOV.B @Rm, Rn

| | | | | | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|---|--|--|--|------|--|--|--|
| 0110 | | | | n | | | | m | | | | 0000 | | | |
| 15 | | | | 12 | | | | 8 | | | | 4 | | | |
| | | | | 1 | | | | 0 | | | | | | | |

```
op1 ← SignExtend32(Rm) ;  
address ← ZeroExtend32(op1) ;  
op2 ← SignExtend8(ReadMemory8(address)) ;  
Rn ← Register(op2) ;
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note:



MOV.B @Rm+, Rn

Description: This instruction loads a signed byte from memory using register indirect with post-increment addressing. The byte is loaded from the virtual address specified in R_m and sign-extended. R_m is post-incremented by 1, and then the loaded byte is placed in R_n .

Operation:

MOV.B @Rm+, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 0100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend8(ReadMemory8(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 1;
Rm ← Register(op1);
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: If R_m and R_n refer to the same register (that is, $m = n$), the result placed in this register will be the value loaded from memory.

MOV.B @(R0, Rm), Rn

Description: This instruction loads a signed byte from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_m . The byte is loaded from the virtual address, sign-extended and placed in R_n .

Operation:

MOV.B @(R0, Rm), Rn

| | | | |
|------|----|----|------|
| 0000 | n | m | 1100 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend8(ReadMemory8(address));
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

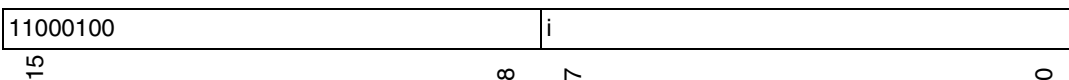
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.B @(disp, GBR), R0

Description: This instruction loads a signed byte from memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i. The byte is loaded from the virtual address, sign-extended and placed in R₀.

Operation:

MOV.B @(disp, GBR), R0



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i);
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend8(ReadMemory8(address));
R0 ← Register(r0);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

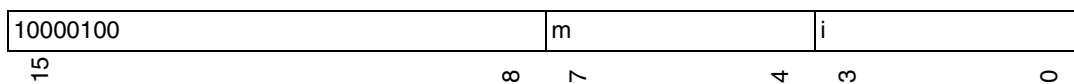
The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.B @(disp, Rm), R0

Description: This instruction loads a signed byte from memory using register indirect with displacement addressing. The virtual address is formed by adding R_m to the zero-extended 4-bit immediate i . The byte is loaded from the virtual address, sign-extended and placed in R_0 .

Operation:

MOV.B @(disp, Rm), R0



```

disp ← ZeroExtend4(i);
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
r0 ← SignExtend8(ReadMemory8(address));
R0 ← Register(r0);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension.

MOV.L Rm, @Rn

Description: This instruction stores a long-word to memory using register indirect with zero-displacement addressing. The virtual address is specified in R_n . The long-word to be stored is held in R_m .

Operation:

MOV.L Rm, @Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0010 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2);
WriteMemory32(address, op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note:

MOV.L Rm, @-Rn

Description: This instruction stores a long-word to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The long-word to be stored is held in R_m .

Operation:

MOV.L Rm, @-Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.L Rm, @(R0, Rn)

Description: This instruction stores a long-word to memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . The long-word to be stored is held in R_m .

Operation:

MOV.L Rm, @(R0, Rn)

| | | | |
|------|----|----|------|
| 0000 | n | m | 0110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory32(address, op1);

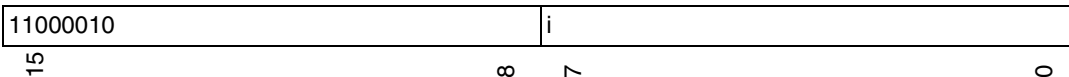
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.L R0, @(disp, GBR)

Description: This instruction stores a long-word to memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i multiplied by 4. The long-word to be stored is held in R_0 .

Operation:**MOV.L R0, @(disp, GBR)**

```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend32(disp + gbr);
WriteMemory32(address, r0);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L Rm, @(disp, Rn)

Description: This instruction stores a long-word to memory using register indirect with displacement addressing. The virtual address is formed by adding R_n to the zero-extended 4-bit immediate i multiplied by 4. The long-word to be stored is held in R_m .

Operation:**MOV.L Rm, @(disp, Rn)**

| | | | |
|------|----|---|---|
| 0001 | n | m | i |
| 15 | 12 | 1 | 0 |

```

op1 ← SignExtend32(Rm);
disp ← ZeroExtend4(i) << 2;
op3 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op3);
WriteMemory32(address, op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L @Rm, Rn

Description: This instruction loads a signed long-word from memory using register indirect with zero-displacement addressing. The virtual address is specified in R_m . The long-word is loaded from the virtual address and placed in R_n .

Operation:

MOV.L @Rm, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 0010 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm) ;
address ← ZeroExtend32(op1) ;
op2 ← SignExtend32(ReadMemory32(address)) ;
Rn ← Register(op2) ;

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note:

MOV.L @Rm+, Rn

Description: This instruction loads a signed long-word from memory using register indirect with post-increment addressing. The long-word is loaded from the virtual address specified in R_m . R_m is post-incremented by 4, and then the loaded long-word is placed in R_n .

Operation:

MOV.L @Rm+, Rn

| | | | |
|------|----|---|------|
| 0110 | n | m | 0110 |
| 15 | 12 | 1 | 0 |

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend32(ReadMemory32(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 4;
Rm ← Register(op1);
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: If R_m and R_n refer to the same register (that is, $m = n$), the result placed in this register is the value loaded from memory.

MOV.L @(R0, Rm), Rn

Description: This instruction loads a signed long-word from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_m . The long-word is loaded from the virtual address and placed in R_n .

Operation:

MOV.L @(R0, Rm), Rn

| | | | |
|------|----|----|------|
| 0000 | n | m | 1110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

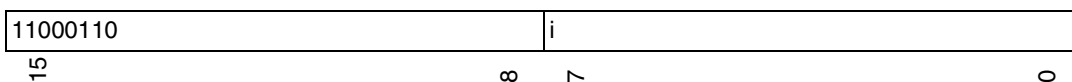
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.L @(disp, GBR), R0

Description: This instruction loads a signed long-word from memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i multiplied by 4. The long-word is loaded from the virtual address and placed in R_0 .

Operation:

MOV.L @(disp, GBR), R0



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i) << 2;
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend32(ReadMemory32(address));
R0 ← Register(r0);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

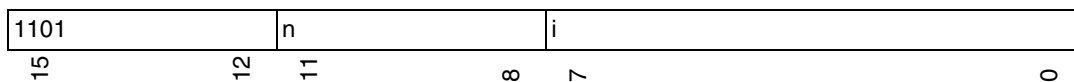
The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L @(disp, PC), Rn

Description: This instruction loads a signed long-word from memory using PC-relative with displacement addressing. The virtual address is formed by calculating PC+4, clearing the lowest 2 bits, and adding the zero-extended 8-bit immediate i multiplied by 4. This address calculation ensures that the virtual address is correctly aligned for a long-word access regardless of the PC alignment. The long-word is loaded from the virtual address and placed in R_n.

Operation:

MOV.L @(disp, PC), Rn



```
pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
    THROW ILLSLOT;
address ← ZeroExtend32(disp + ((pc + 4) ^ (~ 0x3)));
op2 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op2);
```

Exceptions: ILLSLOT, RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.L @(disp, Rm), Rn

Description: This instruction loads a signed long-word from memory using register indirect with displacement addressing. The virtual address is formed by adding R_m to the zero-extended 4-bit immediate i multiplied by 4. The long-word is loaded from the virtual address and placed in R_n .

Operation:

MOV.L @(disp, Rm), Rn

| | | | |
|------|----|---|---|
| 0101 | n | m | i |
| 15 | 12 | 1 | 0 |

```

disp ← ZeroExtend4(i) << 2;
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
op3 ← SignExtend32(ReadMemory32(address));
Rn ← Register(op3);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W Rm, @Rn

Description: This instruction stores a word to memory using register indirect with zero-displacement addressing. The virtual address is specified in R_n. The word to be stored is held in the lowest 16 bits of R_m.

Operation:

MOV.W Rm, @Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0001 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
address ← ZeroExtend32(op2);  
WriteMemory16(address, op1);
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note:

MOV.W Rm, @-Rn

Description: This instruction stores a word to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 2 to give the virtual address. The word to be stored is held in the lowest 16 bits of R_m .

Operation:

MOV.W Rm, @-Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 0101 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 2);
WriteMemory16(address, op1);
op2 ← address;
Rn ← Register(op2);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.W Rm, @(R0, Rn)

Description: This instruction stores a word to memory using register indirect addressing. The virtual address is formed by adding R_0 to R_n . The word to be stored is held in the lowest 16 bits of R_m .

Operation:

MOV.W Rm, @(R0, Rn)

| | | | |
|------|----|----|------|
| 0000 | n | m | 0101 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(r0 + op2);
WriteMemory16(address, op1);

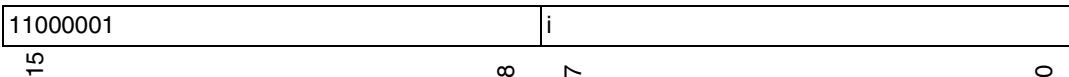
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.W R0, @(disp, GBR)

Description: This instruction stores a word to memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i multiplied by 2. The word to be stored is held in the lowest 16 bits of R_0 .

Operation:**MOV.W R0, @(disp, GBR)**

```

gbr ← SignExtend32(GBR);
r0 ← SignExtend32(R0);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend32(disp + gbr);
WriteMemory16(address, r0);

```

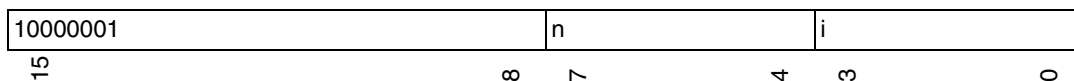
Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W R0, @(disp, Rn)

Description: This instruction stores a word to memory using register indirect with displacement addressing. The virtual address is formed by adding R_n to the zero-extended 4-bit immediate i multiplied by 2. The word to be stored is held in the lowest 16 bits of R_0 .

Operation:**MOV.W R0, @(disp, Rn)**

```

r0 ← SignExtend32(R0);
disp ← ZeroExtend4(i) << 1;
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(disp + op2);
WriteMemory16(address, r0);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W @Rm, Rn

Description: This instruction loads a signed word from memory using register indirect with zero-displacement addressing. The virtual address is specified in R_m . The word is loaded from the virtual address, sign-extended and placed in R_n .

Operation:

MOV.W @Rm, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 0001 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note:

MOV.W @Rm+, Rn

Description: This instruction loads a signed word from memory using register indirect with post-increment addressing. The word is loaded from the virtual address specified in R_m and sign-extended. R_m is post-incremented by 2, and then the loaded word is placed in R_n .

Operation:

MOV.W @Rm+, Rn

| | | | |
|------|----|---|------|
| 0110 | n | m | 0101 |
| 15 | 12 | 1 | 0 |

```

m_field ← ZeroExtend4(m);
n_field ← ZeroExtend4(n);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(op1);
op2 ← SignExtend16(ReadMemory16(address));
IF (m_field = n_field)
    op1 ← op2;
ELSE
    op1 ← op1 + 2;
Rm ← Register(op1);
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: If R_m and R_n refer to the same register (that is, $m = n$), the result placed in this register will be the sign-extended word loaded from memory.

MOV.W @(R0, Rm), Rn

Description: This instruction loads a signed word from memory using register indirect addressing. The virtual address is formed by adding R_0 to R_m . The word is loaded from the virtual address, sign-extended and placed in R_n .

Operation:

MOV.W @(R0, Rm), Rn

| | | | |
|------|----|----|------|
| 0000 | n | m | 1101 |
| 15 | 12 | 11 | 8 |
| 7 | 4 | 3 | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rm);
address ← ZeroExtend32(r0 + op1);
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

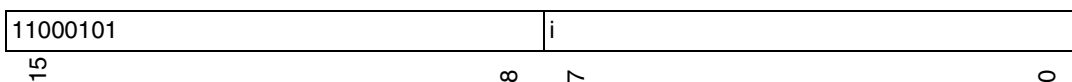
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

MOV.W @(disp, GBR), R0

Description: This instruction loads a signed word from memory using GBR-relative with displacement addressing. The virtual address is formed by adding GBR to the zero-extended 8-bit immediate i multiplied by 2. The word is loaded from the virtual address, sign-extended and placed in R_0 .

Operation:

MOV.W @(disp, GBR), R0



```

gbr ← SignExtend32(GBR);
disp ← ZeroExtend8(i) << 1;
address ← ZeroExtend32(disp + gbr);
r0 ← SignExtend16(ReadMemory16(address));
R0 ← Register(r0);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

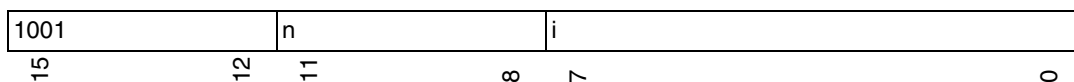
The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W @(disp, PC), Rn

Description: This instruction loads a signed word from memory using PC-relative with displacement addressing. The virtual address is formed by calculating PC+4, and adding the zero-extended 8-bit immediate i multiplied by 2. The word is loaded from the virtual address, sign-extended and placed in R_n.

Operation:

MOV.W @(disp, PC), Rn



```
pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 1;
IF (IsDelaySlot())
    THROW ILLSLOT;
address ← ZeroExtend32(disp + (pc + 4));
op2 ← SignExtend16(ReadMemory16(address));
Rn ← Register(op2);
```

Exceptions: ILLSLOT, RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

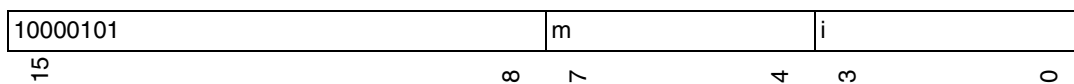
The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOV.W @(disp, Rm), R0

Description: This instruction loads a signed word from memory using register indirect with displacement addressing. The virtual address is formed by adding R_m to the zero-extended 4-bit immediate i multiplied by 2. The word is loaded from the virtual address, sign-extended and placed in R_0 .

Operation:

MOV.W @(disp, Rm), R0



```

disp ← ZeroExtend4(i) << 1;
op2 ← SignExtend32(Rm);
address ← ZeroExtend32(disp + op2);
r0 ← SignExtend16(ReadMemory16(address));
R0 ← Register(r0);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

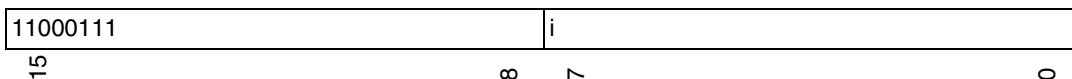
The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOVA @(disp, PC), R0

Description: This instruction calculates an virtual address using PC-relative with displacement addressing. The virtual address is formed by calculating PC+4, clearing the lowest 2 bits, and adding the zero-extended 8-bit immediate i multiplied by 4. This address calculation ensures that the virtual address is correctly aligned for a long-word access regardless of the PC alignment. The virtual address is placed in R₀.

Operation:

MOVA @(disp, PC), R0



```
pc ← SignExtend32(PC);
disp ← ZeroExtend8(i) << 2;
IF (IsDelaySlot())
    THROW ILLSLOT;
r0 ← disp + ((pc + 4) & (~ 0x3));
R0 ← Register(r0);
```

Exceptions: ILLSLOT

Note: The instructions only computes the virtual address, no memory request is made.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

The 'disp' in the assembly syntax represents the immediate i after zero extension and scaling.

MOVCA.L R0, @Rn

Description: This instruction stores the long-word in R_0 to memory at the virtual address specified in R_n . It provides a hint to the implementation that it is not necessary to retrieve the data of this operand cache block from memory. It is implementation-specific as to whether the memory access will occur.

The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

MOVCA.L checks for address error, translation miss and protection exception cases.

Apart from the written long-word, the value of all other locations in the memory block targeted by a MOVCA.L becomes architecturally undefined. However MOVCA.L will not reveal any data that would break the protection model. In particular, if MOVCA.L is executed in user mode, the data in the other locations of the cache block will not belong to privileged mode or to a different user mode context. Programs must not rely on these values. For compatibility with other implementations, software must exercise care when using MOVCA.L.

Operation:

MOVCA.L R0, @Rn

| | | |
|------|---|----------|
| 0000 | n | 11000011 |
| 15 | 2 | 0 |

```

r0 ← SignExtend32(R0);
op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW WADDERR, op1;
IF (MMU() AND DataAccessMultiHit(op1))
    THROW OTLBMULTIHIT, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
    THROW WRITEPROT, op1;
IF (MMU() AND NOT DirtyBit(op1))
    THROW FIRSTWRITE, op1;
ALLOCO(op1);
address ← ZeroExtend32(op1);
WriteMemory32(op1, r0);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note:

MOVT Rn

Description: This instruction copies the T-bit to R_n .

Operation:

MOVT Rn

| | | |
|------|----|----------|
| 0000 | n | 00101001 |
| 15 | 12 | 8 |
| 11 | 7 | 0 |

$t \leftarrow \text{ZeroExtend}_1(T) ;$
 $op1 \leftarrow t ;$
 $R_n \leftarrow \text{Register}(op1) ;$

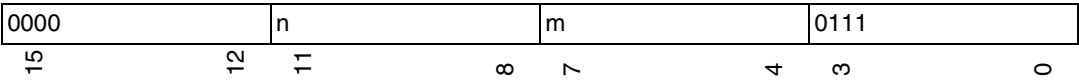
Note:

MUL.L Rm, Rn

Description: This instruction multiplies the 32-bit value in R_m by the 32-bit value in R_n , and places the least significant 32 bits of the result in MACL. The most significant 32 bits of the result are not provided, and MACH is not modified.

Operation:

MUL.L Rm, Rn



```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
mac1 ← op1 × op2;  
MACL ← ZeroExtend32(mac1);
```

Note:



MULS.W Rm, Rn

Description: This instruction multiplies the signed lowest 16 bits of R_m by the signed lowest 16 bits of R_n , and places the full 32-bit result in MACL. MACH is not modified.

Operation:

MULS.W Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1111 |
| 15 | 12 | 11 | 8 |
| 7 | 4 | 3 | 0 |

```

op1 ← SignExtend16(SignExtend32( $R_m$ ));
op2 ← SignExtend16(SignExtend32( $R_n$ ));
mac1 ← op1 × op2;
MACL ← ZeroExtend32(mac1);
    
```

Note:

MULU.W Rm, Rn

Description: This instruction multiplies the unsigned lowest 16 bits of R_m by the unsigned lowest 16 bits of R_n , and places the full 32-bit result in MACL. MACH is not modified.

Operation:

MULU.W Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1110 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← ZeroExtend16(SignExtend32(Rm));  
op2 ← ZeroExtend16(SignExtend32(Rn));  
mac1 ← op1 × op2;  
MACL ← ZeroExtend32(mac1);
```

Note:



NEG Rm, Rn

Description: This instruction subtracts R_m from zero and places the result in R_n .

Operation:

NEG Rm, Rn

| | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|---|---|---|---|---|---|------|---|---|---|
| 0110 | | | | n | | | | m | | | | 1011 | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| <pre>op1 ← SignExtend₃₂(R_m) ; op2 ←-- op1 ; R_n ← Register (op2) ;</pre> | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

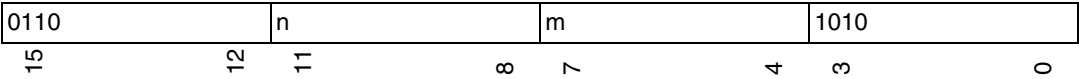
Note:

NEGC Rm, Rn

Description: This instruction subtracts R_m and the T-bit from zero and places the result in R_n . The borrow from the subtraction is placed in the T-bit.

Operation:

NEGC Rm, Rn



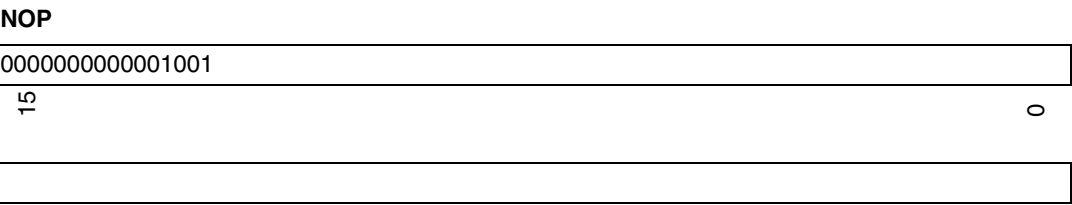
```
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rm);
op2 ← (- op1) - t;
t ← op2 < 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);
```

Note:

NOP

Description: This instruction performs no operation.

Operation:



Note:

NOT Rm, Rn

Description: This instruction performs a bitwise NOT on R_m and places the result in R_n.

Operation:

NOT Rm, Rn

| | | | | | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|---|--|--|--|------|--|--|--|
| 0110 | | | | n | | | | m | | | | 0111 | | | |
| 15 | | | | 12 | | | | 8 | | | | 0 | | | |

```
op1 ← ZeroExtend32(Rm) ;  
op2 ← ~ op1 ;  
Rn ← Register(op2) ;
```

Note:



OCBI @Rn

Description: This instruction invalidates an operand cache block (if any) that corresponds to a specified virtual address. If the data in the operand cache block is dirty, it is discarded without write-back to memory. Immediately after execution of OCBI, assuming no exception was raised, it is guaranteed that the targeted memory block in virtual address space is not present in the operand cache.

The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBI invalidates an implementation-dependent amount of data. For compatibility with other implementations, software must exercise care when using OCBI.

OCBI checks for address error, translation miss and protection exception cases.

Operation:

OCBI @Rn

| | | |
|------|---|----------|
| 0000 | n | 10010011 |
| 15 | 2 | 0 |

```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW WADDERR, op1;
IF (MMU() AND DataAccessMultiHit(op1))
    THROW OTLBMULTIHIT, op1;
IF (MMU() AND DataAccessMiss(op1))
    THROW WTLBMISS, op1;
IF (MMU() AND WriteProhibited(op1))
    THROW WRITEPROT, op1;
IF (MMU() AND NOT DirtyBit(op1))
    THROW FIRSTWRITE, op1;
OCBI(op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note:

OCBP @Rn

Description: This instruction purges an operand cache block (if any) that corresponds to a specified virtual address. If the data in the operand cache block is dirty, it is written back to memory before being discarded. Immediately after execution of OCBP, assuming no exception was raised, it is guaranteed that the targeted memory block in virtual address space is not present in the operand cache.

The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBP checks for address error, translation miss and protection exception cases.

Operation:

OCBP @Rn

| | | |
|------|---|----------|
| 0000 | n | 10100011 |
| 15 | 2 | 0 |

```
op1 ← SignExtend32(Rn);  
IF (AddressUnavailable(op1))  
    THROW RADDERR, op1;  
IF (MMU() AND DataAccessMultiHit(op1))  
    THROW OTLBMULTIHIT, op1;  
IF (MMU() AND DataAccessMiss(op1))  
    THROW RTLBMISS, op1;  
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))  
    THROW READPROT, op1;  
OCBP(op1);
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

Note:



OCBWB @Rn

Description: This instruction write-backs an operand cache block (if any) that corresponds to a specified virtual address. If the data in the operand cache block is dirty, it is written back to memory but is not discarded. Immediately after execution of OCBWB, assuming no exception was raised, it is guaranteed that the targeted memory block in virtual address space will not be dirty in the operand cache.

The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

OCBWB checks for address error, translation miss and protection exception cases.

Operation:

OCBWB @Rn

| | | |
|-------|-------------|----------------|
| 0000 | n | 10110011 |
| r_5 | r_2 r_1 | ∞ r_0 |

```
op1 ← SignExtend32(Rn);  
IF (AddressUnavailable(op1))  
    THROW RADDERR, op1;  
IF (MMU() AND DataAccessMultiHit(op1))  
    THROW OTLBMULTIHIT, op1;  
IF (MMU() AND DataAccessMiss(op1))  
    THROW RTLBMISS, op1;  
IF (MMU() AND (ReadProhibited(op1) AND WriteProhibited(op1)))  
    THROW READPROT, op1;  
OCBWB(op1);
```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMISS, READPROT

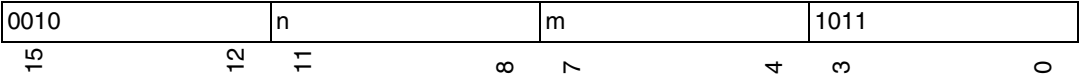
Note:

OR Rm, Rn

Description: This instruction performs a bitwise OR of R_m with R_n and places the result in R_n .

Operation:

OR Rm, Rn



```
op1 ← ZeroExtend32(Rm);  
op2 ← ZeroExtend32(Rn);  
op2 ← op2 v op1;  
Rn ← Register(op2);
```

Note:

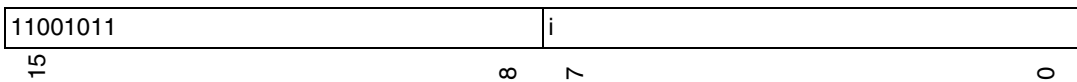


OR #imm, R0

Description: This instruction performs a bitwise OR of R_0 with the zero-extended 8-bit immediate i and places the result in R_0 .

Operation:

OR #imm, R0



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 v imm;
R0 ← Register(r0);

```

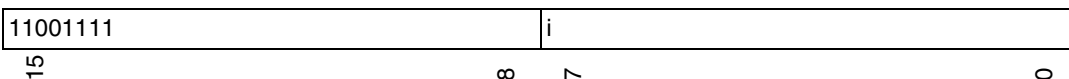
Note: The '#imm' in the assembly syntax represents the immediate i after zero extension.

OR.B #imm, @(R0, GBR)

Description: This instruction performs a bitwise OR of an immediate constant with 8 bits of data held in memory. The virtual address is calculated by adding R_0 and GBR. The 8 bits of data at the virtual address are read. A bitwise OR is performed of the read data with the zero-extended 8-bit immediate i . The result is written back to the 8 bits of data at the same virtual address.

Operation:

OR.B #imm, @(R0, GBR)



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value v imm;
WriteMemory8(address, value);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

PREF @Rn

Description: This instruction indicates a software-directed data prefetch from the specified virtual address. Software can use this instruction to give advance notice that particular data will be required. It is implementation-specific as to whether a prefetch will be performed.

The virtual address specified in R_n identifies a surrounding block of memory, which starts at an address aligned to the cache block size and has a size equal to the cache block size. The cache block size is implementation dependent.

The semantics of a PREF instruction, when applied to an address in the store queues range (0xE0000000 to 0xE3FFFFFF) is quite different to that elsewhere. For details refer to [Section 4.6: Store queues \(SQs\) on page 140](#).

Any OTLBMULTIHIT or RADDERR exception is delivered. RTLBMIS and READPROT are delivered if they arise for accesses to the store queues. For all other accesses, RTLBMIS and READPROT are discarded and the prefetch has no effect.

Operation:

PREF @Rn

| | | |
|------|---|----------|
| 0000 | n | 10000011 |
| 15 | 2 | 0 |

```

op1 ← SignExtend32(Rn);
IF (AddressUnavailable(op1))
    THROW RADDERR, op1
IF (NOT (MMU() AND DataAccessMiss(op1)))
    IF (NOT (MMU() AND ReadProhibited(op1)))
        PREF(op1);

```

Exceptions: RADDERR, READPROT, RTLBMIS, OTLBMULTIHIT

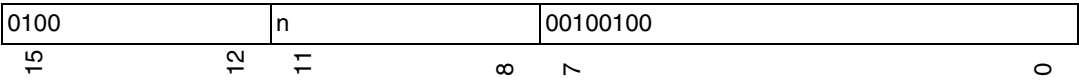
Note:

ROTCL Rn

Description: This instruction performs a one-bit left rotation of the bits held in R_n and the T-bit. The 32-bit value in R_n is shifted one bit to the left, the least significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

Operation:

ROTCL Rn



```
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rn);
op1 ← (op1 << 1) ∨ t;
t ← op1 < 32 FOR 1 >;
Rn ← Register(op1);
T ← Bit(t);
```

Note:

ROTCTR Rn

Description: This instruction performs a one-bit right rotation of the bits held in R_n and the T-bit. The 32-bit value in R_n is shifted one bit to the right, the most significant bit is given the old value of the T-bit, and the bit that is shifted out is moved to the T-bit.

Operation:

ROTCTR Rn

| | | |
|------|----|----------|
| 0100 | n | 00100101 |
| 15 | 12 | 8 |

```

t ← ZeroExtend1(T);
op1 ← ZeroExtend32(Rn);
oldt ← t;
t ← op1 < 0 FOR 1 >;
op1 ← (op1 >> 1) ∨ (oldt << 31);
Rn ← Register(op1);
T ← Bit(t);

```

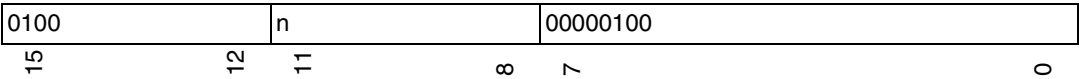
Note:

ROTL Rn

Description: This instruction performs a one-bit left rotation of the bits held in R_n . The 32-bit value in R_n is shifted one bit to the left, and the least significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

Operation:

ROTL Rn



```
op1 ← ZeroExtend32(Rn);  
t ← op1< 31 FOR 1 >;  
op1 ← (op1 << 1) ∨ t;  
Rn ← Register(op1);  
T ← Bit(t);
```

Note:

ROTR Rn

Description: This instruction performs a one-bit right rotation of the bits held in R_n . The 32-bit value in R_n is shifted one bit to the right, and the most significant bit is given the value of the bit that is shifted out. The bit that is shifted out of the operand is also copied to the T-bit.

Operation:

ROTR Rn

| | | |
|------|-------|----------|
| 0100 | n | 00000101 |
| 15 | 12 11 | 8 7 0 |

```

op1 ← ZeroExtend32(Rn);
t ← op1< 0 FOR 1 >;
op1 ← (op1 >> 1) ∨ (t << 31);
Rn ← Register(op1);
T ← Bit(t);

```

Note:

RTE

Description: This instruction returns from an exception or interrupt handling routine by restoring the PC and SR values from SPC and SSR. Program execution continues from the address specified by the restored PC value.

RTE is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an RESINST exception.

Operation:

RTE

0000000000101011

15

0

```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ← SignExtend32(SSR);
pc ← SignExtend32(PC)
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← pc;
delayedpc ← target ^ (~ 0x1);
PC'' ← Register(delayedpc);
```

Exceptions: RESINST, ILLSLOT

Note: Since this is a delayed branch instruction, the instruction in the delay slot is executed before branching and must not generate an exception.

An ILLSLOT exception is raised if this instruction is executed in a delay slot.

Interrupts are not accepted between this instruction and the instruction in the delay slot.

The SR value defined prior to RTE execution is used to fetch the instruction in the RTE delay slot. However, the value of SR used during execution of the instruction in the delay slot, is that restored from SSR by the RTE instruction. It is recommended that, because of this feature, privileged instructions should not be placed in the delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

The behavior is architecturally undefined if the instruction in an RTE delay slot raises an exception. For this reason, it is recommended that only simple instructions that cannot generate exceptions are placed in RTE delay slots (unless considerable care is taken).

RTS

Description: This instruction is a delayed unconditional branch used for returning from a subroutine. The value in PR specifies the target address.

Operation:

RTS

0000000000001011

15

0

```
pr ← SignExtend32(PR);
IF (IsDelaySlot())
    THROW ILLSLOT;
target ← pr;
delayedpc ← target;
PC' ' ← Register(delayedpc);
```

Exceptions: ILLSLOT

Note: Since this is a delayed branch instruction, the delay slot is executed before branching. An ILLSLOT exception is raised if this instruction is executed in a delay slot.

If the branch target address is invalid then IADDERR trap is not delivered until after the instruction in the delay slot has executed and the PC has advanced to the target address, that is the exception is associated with the target instruction not the branch.

SETS

Description: This instruction sets the S-bit to 1.

Operation:

SETS

0000000001011000

150

s ←1;
S ←Bit(s);

Note:



SETT

Description: This instruction sets the T-bit to 1.

Operation:

SETT

00000000000011000

150

t ← 1;
T ← Bit(t);

Note:

SHAD Rm, Rn

Description: This instruction performs an arithmetic shift of R_n , with the dynamic shift direction and shift amount indicated by R_m , and places the result in R_n . If R_m is zero, no shift is performed. If R_m is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of R_m . If R_m is less than zero, this is an arithmetic right shift and the shift amount is given by the least significant 5 bits of R_m subtracted from 32. In the case where R_m indicates an arithmetic right shift by 32, the result is filled with copies of the sign-bit of the original R_n .

Operation:

SHAD Rm, Rn

| | | | |
|------|----|----|------|
| 0100 | n | m | 1100 |
| 15 | 12 | 11 | 8 |
| 7 | 6 | 4 | 3 |
| 0 | | | |

```
op1 ← SignExtend32(Rm);
op2 ← SignExtend32(Rn);
shift_amount ← ZeroExtend5(op1);
IF (op1 ≥ 0)
    op2 ← op2 << shift_amount;
ELSE IF (shift_amount ≠ 0)
    op2 ← op2 >> (32 - shift_amount);
ELSE IF (op2 < 0)
    op2 ← - 1;
ELSE
    op2 ← 0;
Rn ← Register(op2);
```

Note:

SHAL Rn

Description: Arithmetically shifts R_n to the left by one bit and places the result in R_n . The bit that is shifted out of the operand is moved to T-bit.

Operation:

SHAL Rn

| | | |
|------|------------|-----------------|
| 0100 | n | 00100000 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← SignExtend32(Rn);  
t ← op1< 31 FOR 1 >;  
op1 ← op1 << 1;  
Rn ← Register(op1);  
T ← Bit(t);
```

Note:

SHAR Rn

Description: Arithmetically shifts R_n to the right by one bit and places the result in R_n . The bit that is shifted out of the operand is moved to T-bit.

Operation:

SHAR Rn

| | | |
|------|---------|---------------------------|
| 0100 | n | 00100001 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← SignExtend32(Rn);  
t ← op1< 0 FOR 1 >;  
op1 ← op1 >> 1;  
Rn ← Register(op1);  
T ← Bit(t);
```

Note:

SHLD Rm, Rn

Description: This instruction performs a logical shift of R_n , with the dynamic shift direction and shift amount indicated by R_m , and places the result in R_n . If R_m is zero, no shift is performed. If R_m is greater than zero, this is a left shift and the shift amount is given by the least significant 5 bits of R_m . If R_m is less than zero, this is a logical right shift and the shift amount is given by the least significant 5 bits of R_m subtracted from 32. In the case where R_m indicates a logical right shift by 32, the result is 0.

Operation:

SHLD Rm, Rn

| | | | |
|------|----|----|------|
| 0100 | n | m | 1101 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← SignExtend32(Rm) ;
op2 ← ZeroExtend32(Rn) ;
shift_amount ← ZeroExtend5(op1) ;
IF (op1 ≥ 0)
    op2 ← op2 << shift_amount ;
ELSE IF (shift_amount ≠ 0)
    op2 ← op2 >> (32 - shift_amount) ;
ELSE
    op2 ← 0 ;
Rn ← Register(op2) ;
    
```

Note:

SHLL Rn

Description: This instruction performs a logical left shift of R_n by 1 bit and places the result in R_n . The bit that is shifted out is moved to the T-bit.

Operation:

SHLL Rn

| | | |
|------|-------|----------|
| 0100 | n | 00000000 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← ZeroExtend32(Rn);  
t ← op1< 31 FOR 1 >;  
op1 ← op1 << 1;  
Rn ← Register(op1);  
T ← Bit(t);
```

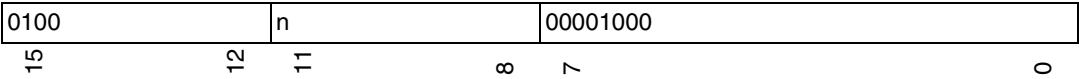
Note:

SHLL2 Rn

Description: This instruction performs a logical left shift of R_n by 2 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLL2 Rn



```
op1 ← ZeroExtend32(Rn) ;  
op1 ← op1 << 2 ;  
Rn ← Register(op1) ;
```

Note:

SHLL8 Rn

Description: This instruction performs a logical left shift of R_n by 8 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLL8 Rn

| | | | | | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|---|--|--|--|
| 0100 | | | | n | | | | 00011000 | | | | | | | |
| 15 | | | | 12 | | | | 8 | | | | 0 | | | |

op1 ←ZeroExtend₃₂(R_n);

op1 ←op1 << 8;

R_n ←Register(op1);

Note:



SHLL16 Rn

Description: This instruction performs a logical left shift of R_n by 16 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLL16 Rn

| | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|
| 0100 | | | | n | | | | 00101000 | | | |
| 15 | | | | 12 | | | | 8 | | | |
| | | | | 4 | | | | 0 | | | |

$$\begin{aligned} \text{op1} &\leftarrow \text{ZeroExtend}_{32}(R_n); \\ \text{op1} &\leftarrow \text{op1} \ll 16; \\ R_n &\leftarrow \text{Register}(\text{op1}); \end{aligned}$$

Note:

SHLR Rn

Description: This instruction performs a logical right shift of R_n by 1 bit and places the result in R_n . The bit that is shifted out is moved to the T-bit.

Operation:

SHLR Rn

| | | | | | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|---|--|--|--|
| 0100 | | | | n | | | | 00000001 | | | | | | | |
| 15 | | | | 12 | | | | 8 | | | | 0 | | | |

```
op1 ← ZeroExtend32(Rn);  
t ← op1< 0 FOR 1 >;  
op1 ← op1 >> 1;  
Rn ← Register(op1);  
T ← Bit(t);
```

Note:

SHLR2 Rn

Description: This instruction performs a logical right shift of R_n by 2 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLR2 Rn

| | | |
|------|-------|----------|
| 0100 | n | 00001001 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← ZeroExtend32(Rn) ;  
op1 ← op1 >> 2 ;  
Rn ← Register(op1) ;
```

Note:

SHLR8 Rn

Description: This instruction performs a logical right shift of R_n by 8 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLR8 Rn

| | | |
|------|-------|----------|
| 0100 | n | 00011001 |
| 15 | 12 11 | 8 7 0 |

```
op1 ← ZeroExtend32(Rn) ;  
op1 ← op1 >> 8 ;  
Rn ← Register(op1) ;
```

Note:

SHLR16 Rn

Description: This instruction performs a logical right shift of R_n by 16 bits and places the result in R_n . The bits that are shifted out are discarded.

Operation:

SHLR16 Rn

| | | |
|------|-------|----------|
| 0100 | n | 00101001 |
| 15 | 12 11 | 8 7 0 |

$$\begin{aligned} op1 &\leftarrow \text{ZeroExtend}_{32}(R_n); \\ op1 &\leftarrow op1 \gg 16; \\ R_n &\leftarrow \text{Register}(op1); \end{aligned}$$

Note:

SLEEP

Description: This instruction places the CPU in the power-down state.

In power-down mode, the CPU retains its internal state, but immediately stops executing instructions and waits for an interrupt request. The PC at the point of sleep is the address of the instruction immediately following the SLEEP instruction. This property ensures that when the CPU receives an interrupt request, and exits the power-down state, the SPC will contain the address of the instruction following the SLEEP.

SLEEP is a privileged instruction, and can only be used in privileged mode. Use of this instruction in user mode will cause an RESINST exception.

Operation:

SLEEP

0000000000011011

15

0

```
md ←ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
SLEEP()
```

Exceptions: RESINST

Note: The effect of SLEEP upon rest of system depends upon the system architecture. Refer to the *Core Support Peripheral Architecture Manual* of the appropriate product for further details.

STC GBR, Rn

Description: This instruction copies GBR to R_n.

Operation:

STC GBR, Rn

| | | | | | | | | | | | |
|------|--|--|--|----|--|--|--|----------|--|--|--|
| 0000 | | | | n | | | | 00010010 | | | |
| r15 | | | | r1 | | | | 0 | | | |

```
gbr ← SignExtend32(GBR);  
op1 ← gbr;  
Rn ← Register(op1);
```

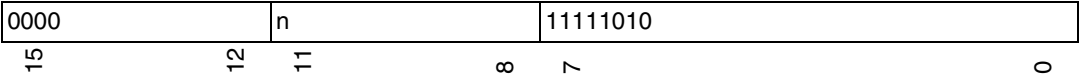
Note:

STC DBR, Rn

Description: This instruction copies DBR to R_n. It is a privileged instruction.

Operation:

STC DBR, R_n



```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
dbr ←SignExtend32(DBR);
op1 ← dbr
Rn ←Register(op1);
```

Exceptions: RESINST

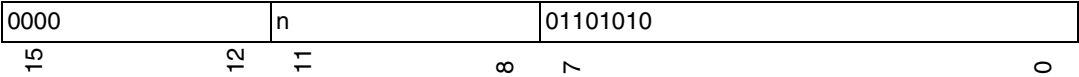
Note:

STS FPSCR, Rn

Description: This floating-point instruction copies FPSCR to R_n.

Operation:

STS FPSCR, Rn



```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← fps;
Rn ← Register(op1);
```

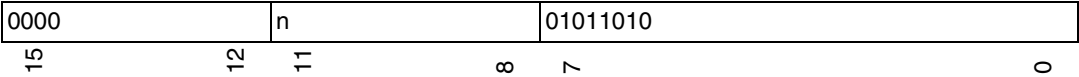
Exceptions: SLOTFPUDIS, FPUDIS

STS FPUL, Rn

Description: This floating-point instruction copies FPUL to R_n.

Operation:

STS FPUL, Rn



```
sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
op1 ← fpul;
Rn ← Register(op1);
```

Exceptions: SLOTFPUDIS, FPUDIS



STS MACH, Rn

Description: This instruction copies MACH to R_n.

Operation:

STS MACH, Rn

| | | | | | | | | | | | |
|------|----|----|---|---|--|--|--|----------|--|---|--|
| 0000 | | | | n | | | | 00001010 | | | |
| 15 | 12 | 11 | 8 | 7 | | | | | | 0 | |

```
mach ← SignExtend32(MACH);  
op1 ← mach;  
Rn ← Register(op1);
```

Note:

STS MACL, Rn

Description: This instruction copies MACL to R_n.

Operation:

STS MACL, Rn

| | | | | | | | | | | | |
|------|----|----|---|---|--|--|--|----------|--|---|--|
| 0000 | | | | n | | | | 00011010 | | | |
| 15 | 12 | 11 | 8 | 7 | | | | | | 0 | |

```
mac1 ← SignExtend32(MACL);  
op1 ← mac1;  
Rn ← Register(op1);
```

Note:



STS PR, Rn

Description: This instruction copies PR to R_n.

Operation:

STS PR, Rn

| | | |
|------|----|----------|
| 0000 | n | 00101010 |
| 15 | 11 | 0 |

```
pr ← SignExtend32(PR);  
op1 ← pr;  
Rn ← Register(op1);
```

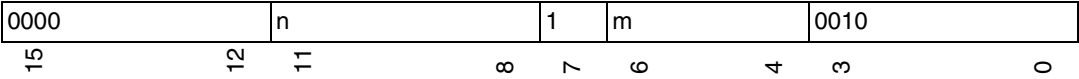
Note:

STC Rm_BANK, Rn

Description: This instruction copies Rm_BANK to R_n. This is a privileged instruction.

Operation:

STC Rm_BANK, Rn



```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ←SignExtend32(Rm_BANK);
op2 ←op1;
Rn ←Register(op2);
```

Exceptions: RESINST

Note:

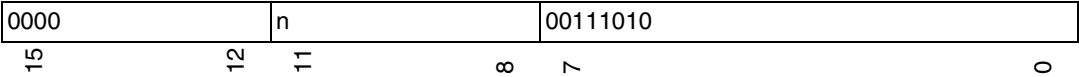


STC SGR, Rn

Description: This instruction copies SGR to R_n. This is a privileged instruction.

Operation:

STC SGR, R_n



```
md ←ZeroExtend1(MD);  
IF (md = 0)  
    THROW RESINST;  
sgr ←SignExtend32(SGR);  
op1 ← sgr  
Rn ←Register(op1);
```

Exceptions: RESINST

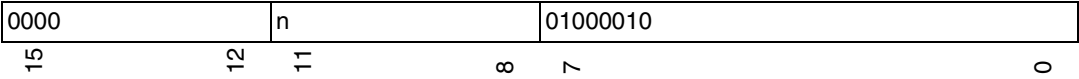
Note:

STC SPC, Rn

Description: This instruction copies SPC to R_n. This is a privileged instruction.

Operation:

STC SPC, R_n



```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
spc ←SignExtend32(SPC);
op1 ← spc
Rn ←Register(op1);
```

Exceptions: RESINST

Note:



STC SR, Rn

Description: This instruction copies SR to R_n. This is a privileged instruction.

Operation:

STC SR, R_n

| | | |
|------|----|----------|
| 0000 | n | 00000010 |
| 15 | 11 | 0 |

```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sr ←SignExtend32(SR);
op1 ← sr
Rn ←Register(op1);
```

Exceptions: RESINST

Note:

STC SSR, Rn

Description: This instruction copies SSR to R_n. This is a privileged instruction.

Operation:

STC SSR, R_n

| | | |
|------|-------|----------|
| 0000 | n | 00110010 |
| 15 | 12 11 | 8 7 0 |

```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ←SignExtend32(SSR);
op1 ← ssr
Rn ←Register(op1);
```

Exceptions: RESINST

Note:

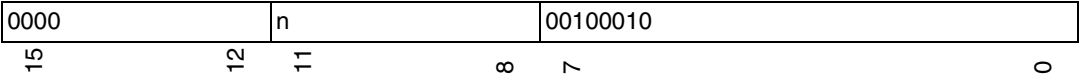


STC VBR, Rn

Description: This instruction copies VBR to R_n. This is a privileged instruction.

Operation:

STC VBR, R_n



```
md ←ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
vbr ←SignExtend32(VBR);
op1 ← vbr
Rn ←Register(op1);
```

Exceptions: RESINST

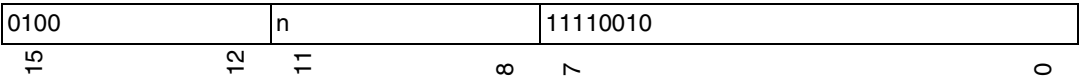
Note:

STC.L DBR, @-Rn

Description: This instruction stores DBR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of DBR is written to the virtual address. This is a privileged instruction.

Operation:

STC.L DBR, @-Rn



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
dbr ← SignExtend32(DBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, dbr);
op1 ← address;
Rn ← Register(op1);
```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



STC.L GBR, @-Rn

Description: This instruction stores GBR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of GBR is written to the virtual address.

Operation:

STC.L GBR, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00010011 |
| 15 | 12 | 8 |

```

gbr ← SignExtend32(GBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, gbr);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

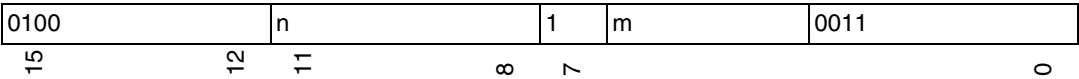
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L Rm_BANK, @-Rn

Description: This instruction stores Rm_BANK to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of Rm_BANK is written to the virtual address. This is a privileged instruction.

Operation:

STC.L Rm_BANK, @-Rn



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
op1 ← SignExtend32(Rm_BANK);
op2 ← SignExtend32(Rn);
address ← ZeroExtend32(op2 - 4);
WriteMemory32(address, op1);
op2 ← address;
Rn ← Register(op2);
```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L SGR, @-Rn

Description: This instruction stores SGR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of SGR is written to the virtual address. This is a privileged instruction.

Operation:

STC.L SGR, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00110010 |
| 15 | 12 | 8 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sgr ← SignExtend32(SGR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, sgr);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L SPC, @-Rn

Description: This instruction stores SPC to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of SPC is written to the virtual address. This is a privileged instruction.

Operation:

STC.L SPC, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 01000011 |
| 15 | 12 | 8 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
spc ← SignExtend32(SPC);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, spc);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L SR, @-Rn

Description: This instruction stores SR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of SR is written to the virtual address. This is a privileged instruction.

Operation:

STC.L SR, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00000011 |
| 15 | 12 | 8 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
sr ← SignExtend32(SR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, sr);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

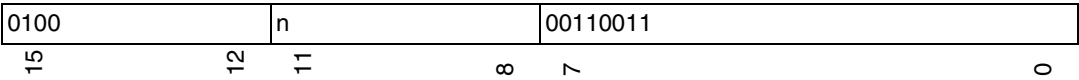
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L SSR, @-Rn

Description: This instruction stores SSR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of SSR is written to the virtual address. This is a privileged instruction.

Operation:

STC.L SSR, @-Rn



```
md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
ssr ← SignExtend32(SSR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, ssr);
op1 ← address;
Rn ← Register(op1);
```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STC.L VBR, @-Rn

Description: This instruction stores VBR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of VBR is written to the virtual address. This is a privileged instruction.

Operation:

STC.L VBR, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00100011 |
| 15 | 12 | 8 |

```

md ← ZeroExtend1(MD);
IF (md = 0)
    THROW RESINST;
vbr ← SignExtend32(VBR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, vbr);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: RESINST, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

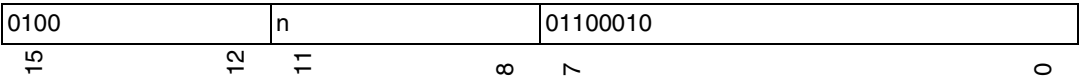
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STS.L FPSCR, @-Rn

Description: This floating-point instruction stores FPSCR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of FPSCR is written to the virtual address.

Operation:

STS.L FPSCR, @-Rn



```
sr ← ZeroExtend32(SR);
fps ← ZeroExtend32(FPSCR);
op1 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
value ← fps;
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, value);
op1 ← address;
Rn ← Register(op1);
```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



STS.L FPUL, @-Rn

Description: This floating-point instruction stores FPUL to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of FPUL is written to the virtual address.

Operation:

STS.L FPUL, @-Rn

| | | |
|------|---|----------|
| 0100 | n | 01010010 |
| 15 | 2 | 0 |

```

sr ← ZeroExtend32(SR);
fpul ← SignExtend32(FPUL);
op1 ← SignExtend32(Rn);
IF (FpuIsDisabled(sr) AND IsDelaySlot())
    THROW SLOTFPUDIS;
IF (FpuIsDisabled(sr))
    THROW FPUDIS;
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, fpul);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: SLOTFPUDIS, FPUDIS, WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STS.L MACH, @-Rn

Description: This instruction stores MACH to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of MACH is written to the virtual address.

Operation:

STS.L MACH, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00000010 |
| 15 | 12 | 8 |

```
mach ← SignExtend32(MACH);  
op1 ← SignExtend32(Rn);  
address ← ZeroExtend32(op1 - 4);  
WriteMemory32(address, mach);  
op1 ← address;  
Rn ← Register(op1);
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.



STS.L MACL, @-Rn

Description: This instruction stores MACL to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of MACL is written to the virtual address.

Operation:

STS.L MACL, @-Rn

| | | |
|------|----|----------|
| 0100 | n | 00010010 |
| 15 | 12 | 8 |

```

mac1 ← SignExtend32(MACL);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, mac1);
op1 ← address;
Rn ← Register(op1);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

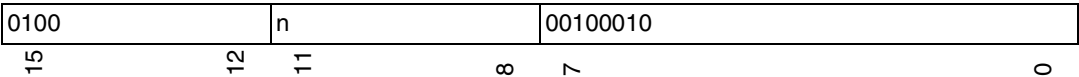
Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

STS.L PR, @-Rn

Description: This instruction stores PR to memory using register indirect with pre-decrement addressing. R_n is pre-decremented by 4 to give the virtual address. The 32-bit value of PR is written to the virtual address.

Operation:

STS.L PR, @-Rn



```
pr ← SignExtend32(PR);
op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1 - 4);
WriteMemory32(address, pr);
op1 ← address;
Rn ← Register(op1);
```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

SUB Rm, Rn

Description: This instruction subtracts R_m from R_n and places the result in R_n .

Operation:

SUB Rm, Rn

| | | | | | | | | | | | | | | | |
|------|----|----|----|----|----|---|---|---|---|---|---|------|---|---|---|
| 0011 | | | | n | | | | m | | | | 1000 | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
op2 ← op2 - op1;  
Rn ← Register(op2);
```

Note:

SUBC Rm, Rn

Description: This instruction subtracts R_m and the T-bit from R_n and places the result in R_n . The borrow from the subtraction is placed in the T-bit.

Operation:

SUBC Rm, Rn

| | | | |
|------|----|----|------|
| 0011 | n | m | 1010 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
t ← ZeroExtend1(T);
op1 ← ZeroExtend32(SignExtend32(Rm));
op2 ← ZeroExtend32(SignExtend32(Rn));
op2 ← (op2 - op1) - t;
t ← op2< 32 FOR 1 >;
Rn ← Register(op2);
T ← Bit(t);
```

Note:

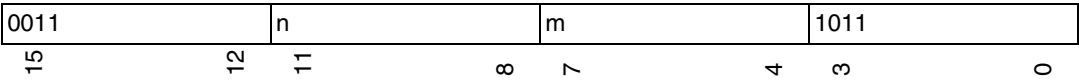


SUBV Rm, Rn

Description: This instruction subtracts R_m from R_n and places the result in R_n . The T-bit is set to 1 if the subtraction result is outside the 32-bit signed range, otherwise the T-bit is set to 0.

Operation:

SUBV Rm, Rn



```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
op2 ← op2 - op1;  
t ← INT ((op2 < (- 231)) OR (op2 ≥ 231));  
Rn ← Register(op2);  
T ← Bit(t);
```

Note:

SWAP.B Rm, Rn

Description: This instruction swaps the values of the lower 2 bytes in R_m and places the result in R_n . Bits [7:0] take the value of bits [15:8]. Bits [15:8] take the value of bits [7:0]. Bits [31:16] are unchanged.

Operation:

SWAP.B Rm, Rn

| | | | |
|------|----|----|------|
| 0110 | n | m | 1000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← ZeroExtend32(Rm);  
op2 ← ((op1< 16 FOR 16 > << 16) ∨ (op1< 0 FOR 8 > << 8)) ∨ op1< 8 FOR 8 >;  
Rn ← Register(op2);
```

Note:

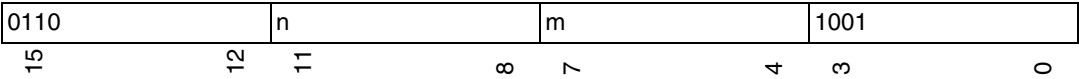


SWAP.W Rm, Rn

Description: This instruction swaps the values of the 2 words in R_m and places the result in R_n . Bits [15:0] take the value of bits [31:16]. Bits [31:16] take the value of bits [15:0].

Operation:

SWAP.W Rm, Rn



```
op1 ← ZeroExtend32(Rm);  
op2 ← (op1 < 0 FOR 16 > << 16) ∨ op1 < 16 FOR 16 >;  
Rn ← Register(op2);
```

Note:

SYNCO

Description: This instruction is used to synchronize data operations. Execution of a **SYNCO** ensures that all data operations from previous instructions are completed before any data operations from subsequent instructions are started.

Operation:

SYNCO

| |
|------------------|
| 0000000010101011 |
| 150 |

| |
|--------------------------------|
| Available only on the ST40-300 |
| SYNCO(); |

Note: Only implemented on the ST40-300. Execution on other variants will result in unspecified behavior.

For more information see [Section 3.14.2: Coherency mechanisms on ST40-300 series cores on page 103](#).

TAS.B @Rn

Description: This instruction performs a test-and-set operation on the byte data at the virtual address specified in R_n . It begins by purging the operand cache block (if any) containing the accessed memory location. The 8 bits of data at the virtual address are read from memory. If the read data is 0 the T-bit is set, otherwise the T-bit is cleared. The highest bit of the 8-bit data (bit 7) is set, and the result is written back to the memory at the same virtual address.

This test-and-set is atomic from the CPU perspective. This instruction cannot be interrupted during its operation.

Operation:

TAS.B @Rn

| | | |
|------|----|----------|
| 0100 | n | 00011011 |
| 15 | 12 | 8 |

```

op1 ← SignExtend32(Rn);
address ← ZeroExtend32(op1);
OCBP(address)
value ← ZeroExtend8(ReadMemory8(address));
t ← INT (value = 0);
value ← value ∨ (1 << 7);
WriteMemory8(address, value);
T ← Bit(t);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

Note: The TAS.B instruction guarantees atomicity of all accesses from the ST40 regardless of their destination in the address space.

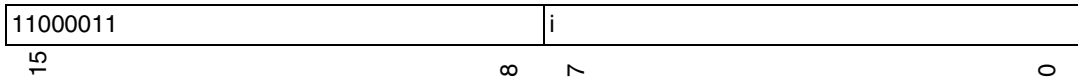
There is no guarantee of atomicity with respect to other bus initiators accessing the same location in memory.

TRAPA #imm

Description: This instruction causes a pre-execution trap. The value of the zero-extended 8-bit immediate *i* is used by the handler launch sequence to characterize the trap.

Operation:

TRAPA #imm



```
imm ← ZeroExtend8(i);
IF (IsDelaySlot())
    THROW ILLSLOT;
THROW TRAP, imm;
```

Exceptions: ILLSLOT, TRAP

Note: An ILLSLOT exception is raised if this instruction is executed in a delay slot.
The '#imm' in the assembly syntax represents the immediate *i* after zero extension.

TST Rm, Rn

Description: This instruction performs a bitwise AND of R_m with R_n . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

Operation:

TST Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1000 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← SignExtend32(Rm);  
op2 ← SignExtend32(Rn);  
t ← INT ((op1 ∧ op2) = 0);  
T ← Bit(t);
```

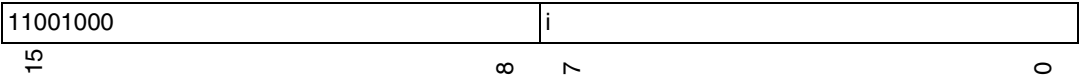
Note:

TST #imm, R0

Description: This instruction performs a bitwise AND of R_0 with the zero-extended 8-bit immediate i . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

Operation:

TST #imm, R0



```
r0 ← SignExtend32(R0);  
imm ← ZeroExtend8(i);  
t ← INT ((r0 ∧ imm) = 0);  
T ← Bit(t);
```

Note: The '#imm' in the assembly syntax represents the immediate i after zero extension.

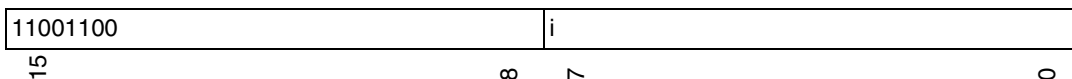


TST.B #imm, @(R0, GBR)

Description: This instruction performs a bitwise test of an immediate constant with 8 bits of data held in memory. The virtual address is calculated by adding R_0 and GBR. The 8 bits of data at the virtual address are read. A bitwise AND is performed of the read data with the zero-extended 8-bit immediate i . If the result is 0, the T-bit is set, otherwise the T-bit is cleared.

Operation:

TST.B #imm, @(R0, GBR)



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
t ← ((value ∧ imm) = 0);
T ← Bit(t);

```

Exceptions: RADDERR, OTLBMULTIHIT, RTLBMIS, READPROT

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

XOR Rm, Rn

Description: This instruction performs a bitwise XOR of R_m with R_n and places the result in R_n .

Operation:

XOR Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1010 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```
op1 ← ZeroExtend32(Rm);  
op2 ← ZeroExtend32(Rn);  
op2 ← op2 ⊕ op1;  
Rn ← Register(op2);
```

Note:

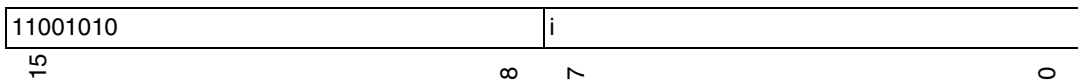


XOR #imm, R0

Description: This instruction performs a bitwise XOR of R_0 with the zero-extended 8-bit immediate i and places the result in R_0 .

Operation:

XOR #imm, R0



```

r0 ← ZeroExtend32(R0);
imm ← ZeroExtend8(i);
r0 ← r0 ⊕ imm;
R0 ← Register(r0);

```

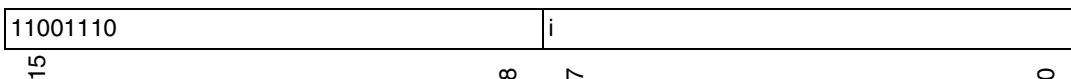
Note: The '#imm' in the assembly syntax represents the immediate i after zero extension.

XOR.B #imm, @(R0, GBR)

Description: This instruction performs a bitwise XOR of an immediate constant with 8 bits of data held in memory. The virtual address is calculated by adding R_0 and GBR. The 8 bits of data at the virtual address are read. A bitwise XOR is performed of the read data with the zero-extended 8-bit immediate i . The result is written back to the 8 bits of data at the same virtual address.

Operation:

XOR.B #imm, @(R0, GBR)



```

r0 ← SignExtend32(R0);
gbr ← SignExtend32(GBR);
imm ← ZeroExtend8(i);
address ← ZeroExtend32(r0 + gbr);
value ← ZeroExtend8(ReadMemory8(address));
value ← value ⊕ imm;
WriteMemory8(address, value);

```

Exceptions: WADDERR, OTLBMULTIHIT, WTLBMISS, READPROT, WRITEPROT, FIRSTWRITE

Note: The virtual address calculation is performed using 32-bit zero extension to cause wrap around if the address-space bounds are exceeded.

The '#imm' in the assembly syntax represents the immediate i after zero extension.

XTRCT Rm, Rn

Description: This instruction extracts the lower 16-bit word from R_m and the upper 16-bit word from R_n , swaps their order, and places the result in R_n . Bits [15:0] of R_n take the value of bits [31:16] of the original R_n . Bits [31:16] of R_n take the value of bits [15:0] of R_m .

Operation:

XTRCT Rm, Rn

| | | | |
|------|----|----|------|
| 0010 | n | m | 1101 |
| 15 | 12 | 11 | 8 |
| | | 7 | 4 |
| | | | 3 |
| | | | 0 |

```

op1 ← ZeroExtend32(Rm);
op2 ← ZeroExtend32(Rn);
op2 ← op2 < 16 FOR 16 > ∨ (op1 < 0 FOR 16 > << 16);
Rn ← Register(op2);

```

Note:

10 ST40-100, 200, 400, 500 performance characteristics

The ST40 CPU core implementation used in the 100, 200, 400 and 500-series has a dual-issue superscalar 5-stage integer pipeline. This section gives a high-level description of the performance characteristics of this particular implementation of the ST40 architecture.

Note: The descriptions of the FPU instruction performance are not relevant to products with no hardware support for floating point operations.

10.1 Pipelines

[Figure 22](#) shows the basic pipelines. Normally, a pipeline consists of five or six stages: instruction fetch (I), decode and register read (D), execution (EX/SX/F0/F1/F2/F3), data access (NA/MA), and write-back (S/FS). An instruction is executed as a combination of basic pipelines. [Figure 23](#) to [Figure 27](#) show the instruction execution patterns.

Figure 22. Basic pipelines

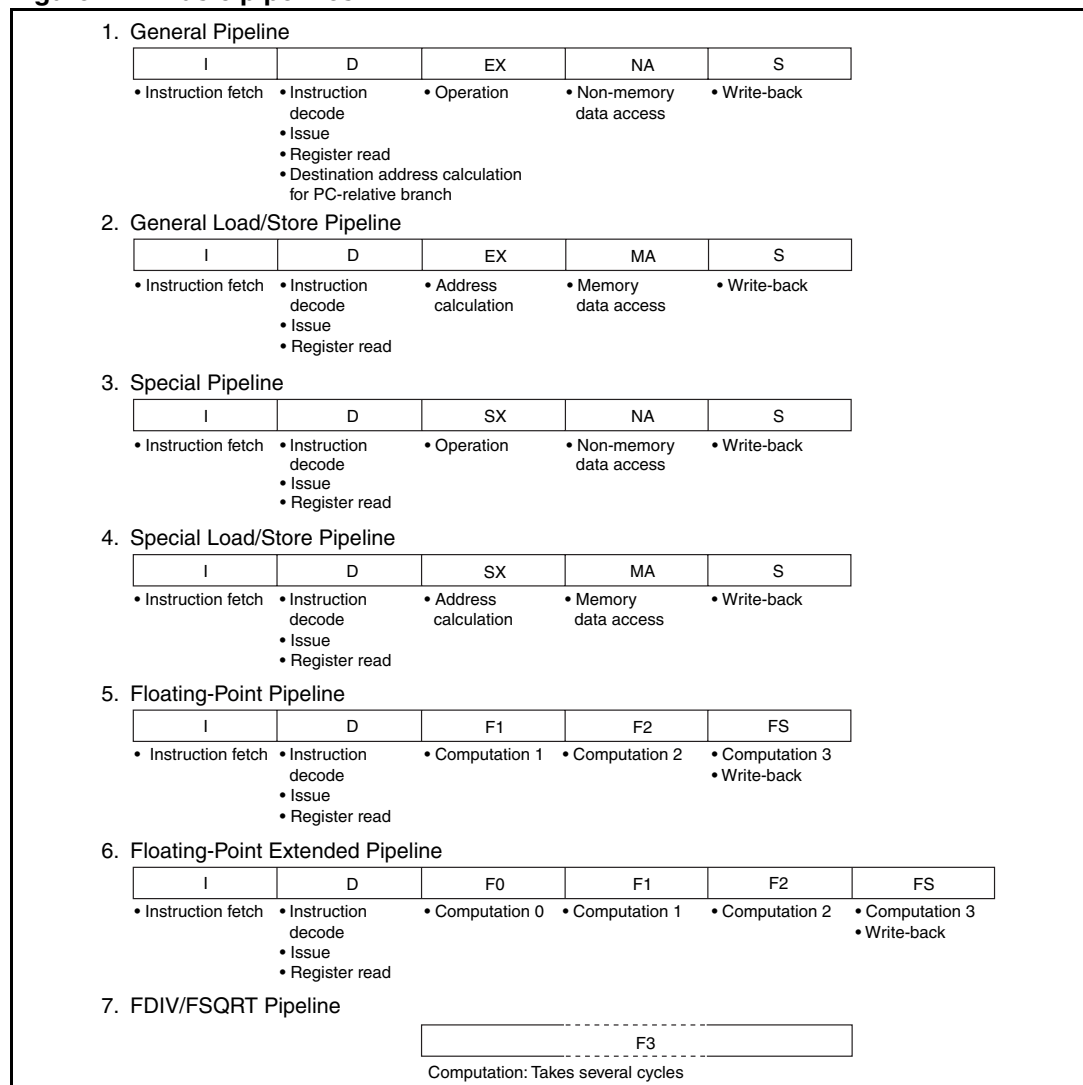


Figure 23. Instruction execution patterns

1. 1-step operation: 1 issue cycle
EXT[SU].[BW], MOV, MOV#, MOVA, MOVT, SWAP[BW], XTRCT, ADD*, CMP*, DIV*, DT, NEG*, SUB*, AND, AND#, NOT, OR, OR#, TST, TST#, XOR, XOR#, ROT*, SHA*, SHL*, BF*, BT*, BRA, NOP, CLRS, CLRT, SETS, SETT, LDS to FPUL, STS from FPUL/FPSCR, FLDI0, FLDI1, FMOV, FLDS, FSTS, single-/double-precision FABS/FNEG

| | | | | |
|---|---|----|----|---|
| I | D | EX | NA | S |
|---|---|----|----|---|
2. Load/store: 1 issue cycle
MOV.[BWL]. FMOV*@, LDS.L to FPUL, LDTLB, PREF, STS.L from FPUL/FPSCR

| | | | | |
|---|---|----|----|---|
| I | D | EX | MA | S |
|---|---|----|----|---|
3. GBR-based load/store: 1 issue cycle
MOV.[BWL]@(d,GBR)

| | | | | |
|---|---|----|----|---|
| I | D | SX | MA | S |
|---|---|----|----|---|
4. JMP, RTS, BRAF: 2 issue cycles

| | | | | | |
|---|---|----|----|----|---|
| I | D | EX | NA | S | |
| | | D | EX | NA | S |
5. TST.B: 3 issue cycles

| | | | | | | |
|---|---|----|----|----|----|---|
| I | D | SX | MA | S | | |
| | | D | SX | NA | S | |
| | | | D | SX | NA | S |
6. AND.B, OR.B, XOR.B: 4 issue cycles

| | | | | | | | |
|---|---|----|----|----|----|----|---|
| I | D | SX | MA | S | | | |
| | | D | SX | NA | S | | |
| | | | D | SX | NA | S | |
| | | | | D | SX | MA | S |
7. TAS.B: 5 issue cycles

| | | | | | | | | |
|---|---|----|----|----|----|----|----|---|
| I | D | EX | MA | S | | | | |
| | | D | EX | MA | S | | | |
| | | | D | EX | NA | S | | |
| | | | | D | EX | NA | S | |
| | | | | | D | EX | MA | S |
8. RTE: 5 issue cycles

| | | | | | | | | |
|---|---|----|----|----|----|----|----|---|
| I | D | EX | NA | S | | | | |
| | | D | EX | NA | S | | | |
| | | | D | EX | NA | S | | |
| | | | | D | EX | NA | S | |
| | | | | | D | EX | NA | S |
9. SLEEP: 4 issue cycles

| | | | | | | | |
|---|---|----|----|----|----|----|---|
| I | D | EX | NA | S | | | |
| | | D | EX | NA | S | | |
| | | | D | EX | NA | S | |
| | | | | D | EX | NA | S |

Figure 24. Instruction execution patterns (continued)

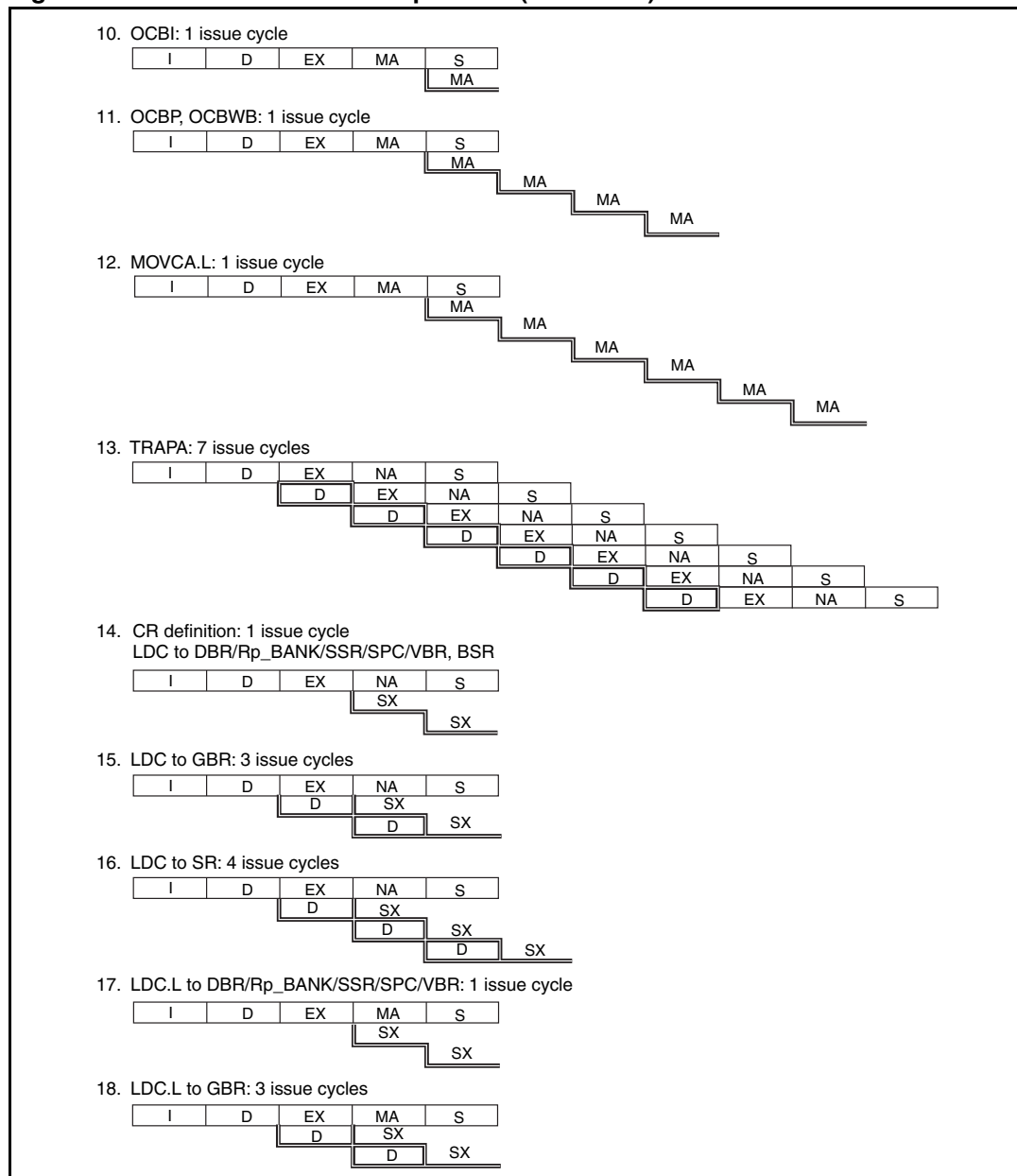


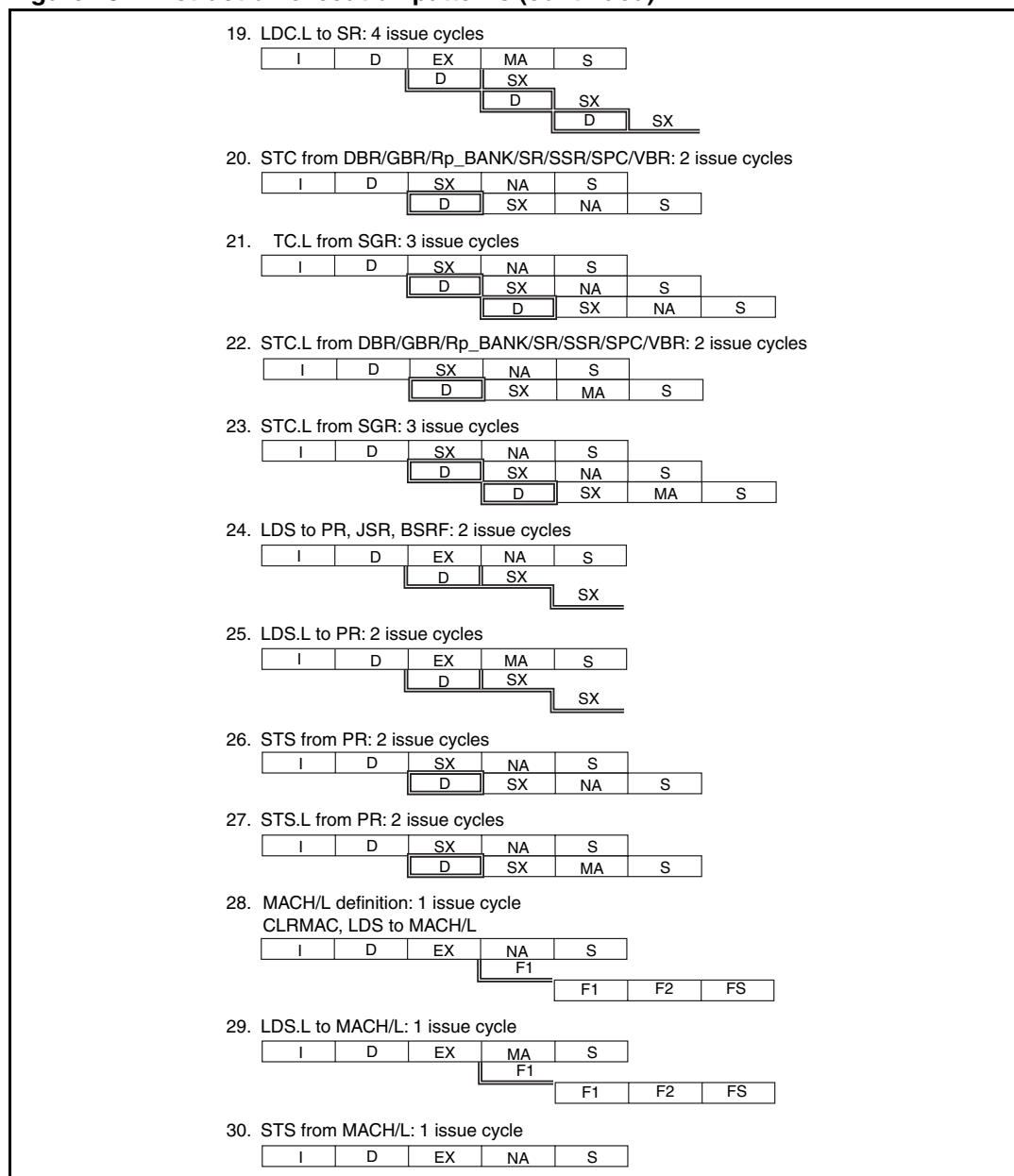
Figure 25. Instruction execution patterns (continued)

Figure 26. Instruction execution patterns (continued)

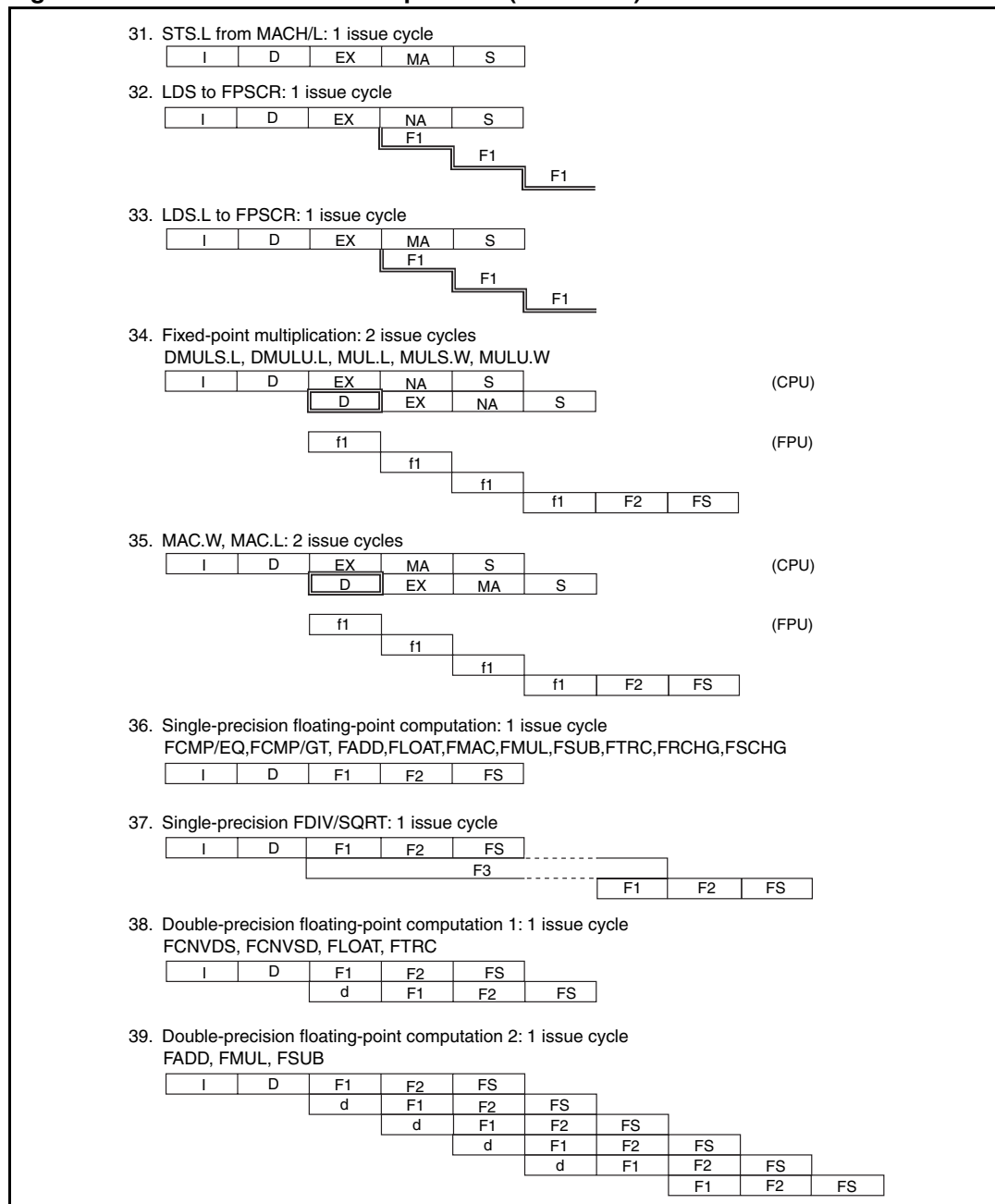
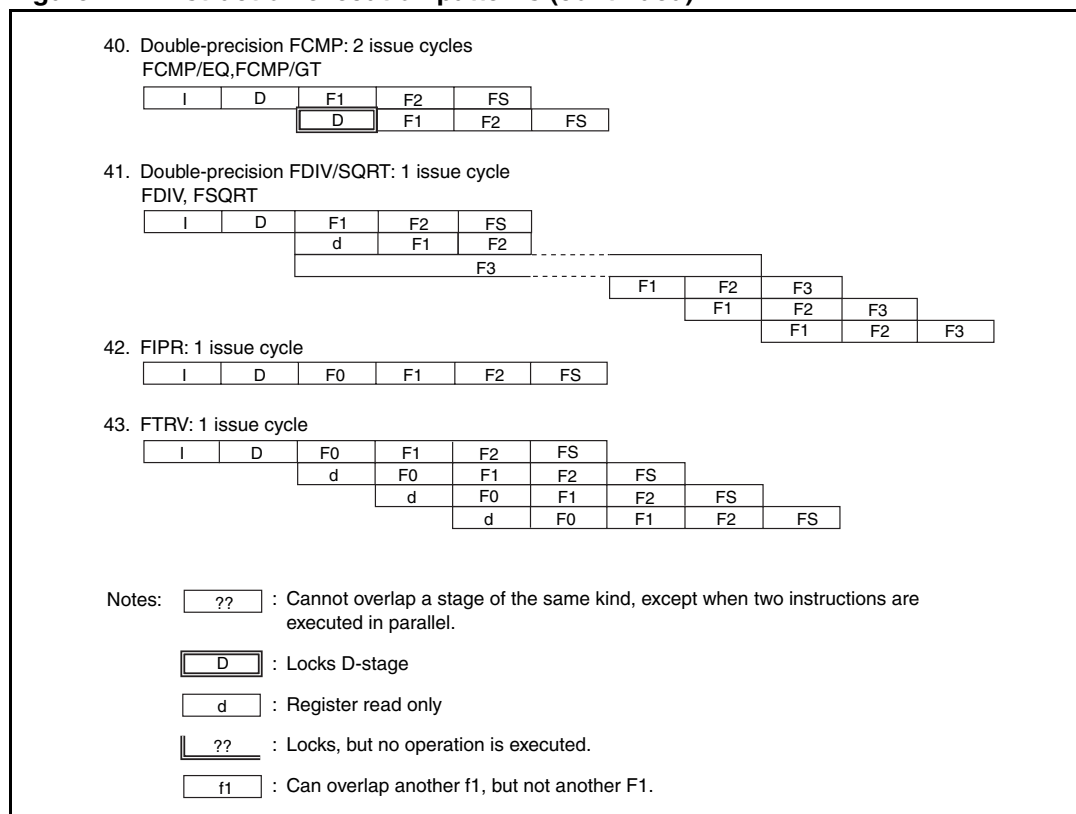


Figure 27. Instruction execution patterns (continued)

10.2 Parallel executables

Instructions are categorized into six groups according to the internal function blocks used, as shown in [Table 133](#). [Table 134](#) shows the parallel executable pairs of instructions in terms of groups. For example, ADD in the EX group and BRA in the BR group can be executed in parallel.

Table 133. Instruction groups

1. MT group

| | | | | | |
|--------|---------|---------|-------|------|---------|
| CLRT | | CMP/HI | Rm,Rn | MOV | Rm,Rn |
| CMP/EQ | #imm,R0 | CMP/HS | Rm,Rn | NOP | |
| CMP/EQ | Rm,Rn | CMP/PL | Rn | SETT | |
| CMP/GE | Rm,Rn | CMP/PZ | Rn | TST | #imm,R0 |
| CMP/GT | Rm,Rn | CMP/STR | Rm,Rn | TST | Rm,Rn |

2. EX group

| | | | | | |
|------|---------|------|-------|--------|----|
| ADD | #imm,Rn | MOVT | Rn | SHLL2 | Rn |
| ADD | Rm,Rn | NEG | Rm,Rn | SHLL8 | Rn |
| ADDC | Rm,Rn | NEGC | Rm,Rn | SHLR | Rn |
| ADDV | Rm,Rn | NOT | Rm,Rn | SHLR16 | Rn |

Table 133. Instruction groups (continued)

| | | | | | |
|--------|---------------|--------|---------|--------|---------|
| AND | #imm,R0 | OR | #imm,R0 | SHLR2 | Rn |
| AND | Rm,Rn | OR | Rm,Rn | SHLR8 | Rn |
| DIV0S | Rm,Rn | ROTCL | Rn | SUB | Rm,Rn |
| DIV0U | | ROTCR | Rn | SUBC | Rm,Rn |
| DIV1 | Rm,Rn | ROTL | Rn | SUBV | Rm,Rn |
| DT | Rn | ROTR | Rn | SWAP.B | Rm,Rn |
| EXTS.B | Rm,Rn | SHAD | Rm,Rn | SWAP.W | Rm,Rn |
| EXTS.W | Rm,Rn | SHAL | Rn | XOR | #imm,R0 |
| EXTU.B | Rm,Rn | SHAR | Rn | XOR | Rm,Rn |
| EXTU.W | Rm,Rn | SHLD | Rm,Rn | XTRCT | Rm,Rn |
| MOV | #imm,Rn | SHLL | Rn | | |
| MOVA | @(disp,PC),R0 | SHLL16 | Rn | | |

3. BR group

| | | | | | |
|------|------|-----|------|------|------|
| BF | disp | BRA | disp | BT | disp |
| BF/S | disp | BSR | disp | BT/S | disp |

4. LS group

| | | | | | |
|-------|--------------|--------|----------------|---------|----------------|
| FABS | DRn | FMOV.S | @Rm+,FRn | MOV.L | R0,@(disp,GBR) |
| FABS | FRn | FMOV.S | FRm,@(R0,Rn) | MOV.L | Rm,@(disp,Rn) |
| FLDI0 | FRn | FMOV.S | FRm,@-Rn | MOV.L | Rm,@(R0,Rn) |
| FLDI1 | FRn | FMOV.S | FRm,@Rn | MOV.L | Rm,@-Rn |
| FLDS | FRm,FPUL | FNEG | DRn | MOV.L | Rm,@Rn |
| FMOV | @(R0,Rm),DRn | FNEG | FRn | MOV.W | @(disp,GBR),R0 |
| FMOV | @(R0,Rm),XDn | FSTS | FPUL,FRn | MOV.W | @(disp,PC),Rn |
| FMOV | @Rm,DRn | LDS | Rm,FPUL | MOV.W | @(disp,Rm),R0 |
| FMOV | @Rm,XDn | MOV.B | @(disp,GBR),R0 | MOV.W | @(R0,Rm),Rn |
| FMOV | @Rm+,DRn | MOV.B | @(disp,Rm),R0 | MOV.W | @Rm,Rn |
| FMOV | @Rm+,XDn | MOV.B | @(R0,Rm),Rn | MOV.W | @Rm+,Rn |
| FMOV | DRm,@(R0,Rn) | MOV.B | @Rm,Rn | MOV.W | R0,@(disp,GBR) |
| FMOV | DRm,@-Rn | MOV.B | @Rm+,Rn | MOV.W | R0,@(disp,Rn) |
| FMOV | DRm,@Rn | MOV.B | R0,@(disp,GBR) | MOV.W | Rm,@(R0,Rn) |
| FMOV | DRm,DRn | MOV.B | R0,@(disp,Rn) | MOV.W | Rm,@-Rn |
| FMOV | DRm,XDn | MOV.B | Rm,@(R0,Rn) | MOV.W | Rm,@Rn |
| FMOV | FRm,FRn | MOV.B | Rm,@-Rn | MOVCA.L | R0,@Rn |
| FMOV | XDm,@(R0,Rn) | MOV.B | Rm,@Rn | OCBI | @Rn |
| FMOV | XDm,@-Rn | MOV.L | @(disp,GBR),R0 | OCBP | @Rn |

Table 133. Instruction groups (continued)

| | | | | | |
|--------|--------------|-------|---------------|-------|---------|
| FMOV | XDm,@Rn | MOV.L | @(disp,PC),Rn | OCBWB | @Rn |
| FMOV | XDm,DRn | MOV.L | @(disp,Rm),Rn | PREF | @Rn |
| FMOV | XDm,XDn | MOV.L | @(R0,Rm),Rn | STS | FPUL,Rn |
| FMOV.S | @(R0,Rm),FRn | MOV.L | @Rm,Rn | | |
| FMOV.S | @Rm,FRn | MOV.L | @Rm+,Rn | | |

5. FE group

| | | | | | |
|---------|-----------|-------|-------------|-------|-----------|
| FADD | DRm,DRn | FIPR | FVm,FVn | FSQRT | DRn |
| FADD | FRm,FRn | FLOAT | FPUL,DRn | FSQRT | FRn |
| FCMP/EQ | FRm,FRn | FLOAT | FPUL,FRn | FSUB | DRm,DRn |
| FCMP/GT | FRm,FRn | FMAC | FR0,FRm,FRn | FSUB | FRm,FRn |
| FCNVDS | DRm,FPUL | FMUL | DRm,DRn | FTRC | DRm,FPUL |
| FCNVSD | FPUL,DRn | FMUL | FRm,FRn | FTRC | FRm,FPUL |
| FDIV | DRm,DRn | FRCHG | | FTRV | XMTRX,FVn |
| FDIV | FRm,FRn | FSCHG | | FSRRA | FRn |
| FSCA | FPUL, DRn | | | | |

6. CO group

| | | | | | |
|---------|----------------|--------|----------------|-------|--------------|
| AND.B | #imm,@(R0,GBR) | LDS | Rm,FPSCR | STC | SR,Rn |
| BRAF | Rm | LDS | Rm,MACH | STC | SSR,Rn |
| BSRF | Rm | LDS | Rm,MACL | STC | VBR,Rn |
| CLRMACH | | LDS | Rm,PR | STC.L | DBR,@-Rn |
| CLRS | | LDS.L | @Rm+,FPSCR | STC.L | GBR,@-Rn |
| DMULS.L | Rm,Rn | LDS.L | @Rm+,FPUL | STC.L | Rp_BANK,@-Rn |
| DMULU.L | Rm,Rn | LDS.L | @Rm+,MACH | STC.L | SGR,@-Rn |
| FCMP/EQ | DRm,DRn | LDS.L | @Rm+,MACL | STC.L | SPC,@-Rn |
| FCMP/GT | DRm,DRn | LDS.L | @Rm+,PR | STC.L | SR,@-Rn |
| JMP | @Rn | LDTLB | | STS | FPSCR,Rn |
| JSR | @Rn | MAC.L | @Rm+,@Rn+ | STS | MACH,Rn |
| LDC | Rm,DBR | MAC.W | @Rm+,@Rn+ | STS | MACL,Rn |
| LDC | Rm,GBR | MUL.L | Rm,Rn | STS | PR,Rn |
| LDC | Rm,Rp_BANK | MULS.W | Rm,Rn | STS.L | FPSCR,@-Rn |
| LDC | Rm,SGR | MULU.W | Rm,Rn | STS.L | FPUL,@-Rn |
| LDC | Rm,SPC | OR.B | #imm,@(R0,GBR) | STS.L | MACH,@-Rn |
| LDC | Rm,SR | RTE | | STS.L | MACL,@-Rn |
| LDC | Rm,SSR | RTS | | STS.L | PR,@-Rn |
| LDC | Rm,VBR | SETS | | TAS.B | @Rn |

Table 133. Instruction groups (continued)

| | | | | |
|-------|--------------|-------|------------|----------------|
| LDC.L | @Rm+,DBR | SLEEP | TRAPA | #imm |
| LDC.L | @Rm+,GBR | STC | DBR,Rn | TST.B |
| LDC.L | @Rm+,Rp_BANK | STC | GBR,Rn | #imm,@(R0,GBR) |
| LDC.L | @Rm+,SGR | STC | Rp_BANK,Rn | XOR.B |
| LDC.L | @Rm+,SPC | STC | SGR,Rn | #imm,@(R0,GBR) |
| LDC.L | @Rm+,SR | STC | SPC,Rn | |
| LDC.L | @Rm+,SSR | STC.L | SSR,@-Rn | |
| LDC.L | @Rm+,VBR | STC.L | VBR,@-Rn | |

Table 134. Parallel executables

| | | 2nd instruction | | | | | |
|-----------------|----|-----------------|----|----|----|----|----|
| | | MT | EX | BR | LS | FE | CO |
| 1st instruction | MT | O | O | O | O | O | X |
| | EX | O | X | O | O | O | X |
| | BR | O | O | X | O | O | X |
| | LS | O | O | O | X | O | X |
| | FE | O | O | O | O | X | X |
| | CO | X | X | X | X | X | X |

Note: O: Can be executed in parallel

X: Cannot be executed in parallel

10.3 Execution cycles and pipeline stalling

Instruction execution cycles are summarized in [Table 135: Execution cycles on page 488](#). Penalty cycles due to a pipeline stall or freeze are not considered in this table.

- Issue rate: interval between the issue of an instruction and that of the next instruction.
- Latency: interval between the issue of an instruction and the generation of its result (completion).
- Instruction execution pattern (see [Figure 23 on page 475](#) to [Figure 27 on page 479](#)).
- Locked pipeline stages.
- Interval between the issue of an instruction and the start of locking.
- Lock time: period of locking in machine cycle units.

The instruction execution sequence is expressed as a combination of the execution patterns shown in [Figure 23](#) to [Figure 27](#). One instruction is separated from the next by the number of machine cycles for its issue rate. Normally, execution, data access, and write-back stages cannot be overlapped onto the same stages of another instruction; the only exception is when two instructions are executed in parallel under parallel executables conditions. Refer to (a) through (d) in [Figure 28 on page 484](#) for some simple examples.

Latency is the interval between issue and completion of an instruction, and is also the interval between the execution of two instructions with an interdependent relationship. When there is interdependency between two instructions fetched simultaneously, the latter of the two is stalled for the following number of cycles:

- (latency) cycles when there is flow dependency (read-after-write)
- (latency - 1) or (latency - 2) cycles when there is output dependency (write-after-write):
 - single/double-precision FDIV, FSQRT is the preceding instruction (latency - 1) cycles
 - the other FE group except above is the preceding instruction (latency - 2) cycles
- 5 or 2 cycles when there is anti-flow dependency (write-after-read), as in the following cases:
 - FTRV is the preceding instruction (5 cycle)
 - a double-precision FADD, FSUB, or FMUL is the preceding instruction (2 cycles)

In the case of flow dependency, latency may be exceptionally increased or decreased, depending on the combination of sequential instructions ([Figure 29 on page 485](#) (e)).

- When a floating-point (FP) computation is followed by an FP register store, the latency of the FP computation may be decreased by 1 cycle.
- If there is a load of the shift amount immediately before an SHAD/SHLD instruction, the latency of the load is increased by 1 cycle.
- If an instruction with a latency of less than 2 cycles, including write-back to an FP register, is followed by a double-precision FP instruction, FIPR, or FTRV, the latency of the first instruction is increased to 2 cycles.

The number of cycles in a pipeline stall due to flow dependency will vary depending on the combination of interdependent instructions or the fetch timing (see [Figure 29 on page 485](#) (e)).

Output dependency occurs when the destination operands are the same in a preceding FE group instruction and a following LS group instruction.

For the stall cycles of an instruction with output dependency, the longest latency to the last write-back among all the destination operands must be applied instead of latency-2 (see [Figure 30 on page 486](#) (f)). A stall due to output dependency with respect to FPSCR, which reflects the result of an FP operation, never occurs. For example, when FADD follows FDIV with no dependency between FP registers, FADD is not stalled even if both instructions update the cause field of FPSCR.

Anti-flow dependency can occur only between a preceding double-precision FADD, FMUL, FSUB, or FTRV and a following FMOV, FLDI0, FLDI1, FABS, FNEG, or FSTS. See [Figure 30 on page 486](#) (g).

If an executing instruction locks any resource, that is, a function block that performs a basic operation, a following instruction that happens to attempt to use the locked resource must be stalled ([Figure 31 on page 487](#) (h)). This kind of stall can be compensated by inserting one or more instructions independent of the locked resource to separate the interfering instructions. For example, when a load instruction and an ADD instruction that references the loaded value are consecutive, the 2-cycle stall of the ADD is eliminated by inserting three instructions without dependency. Software performance can be improved by such instruction scheduling.

Other penalties arise in the event of exceptions or external data accesses, as follows.

- Instruction TLB miss: a penalty of 7 CPU clocks.
- Instruction access to external memory (instruction cache miss for example).
- Data access to external memory (operand cache miss for example).
- Data access to a memory-mapped control register. The penalty differs from register to register, and depends on the kind of operation (read or write), the clock mode, and the bus use conditions when the access is made.

During the penalty cycles of an instruction TLB miss or external instruction access, no instruction is issued, but execution of instructions that have already been issued continues. The penalty for a data access is a pipeline freeze: that is, the execution of uncompleted instructions is interrupted until the arrival of the requested data. The number of penalty cycles for instruction and data accesses is largely dependent on the user's memory subsystems.

Figure 28. Examples of pipelined execution

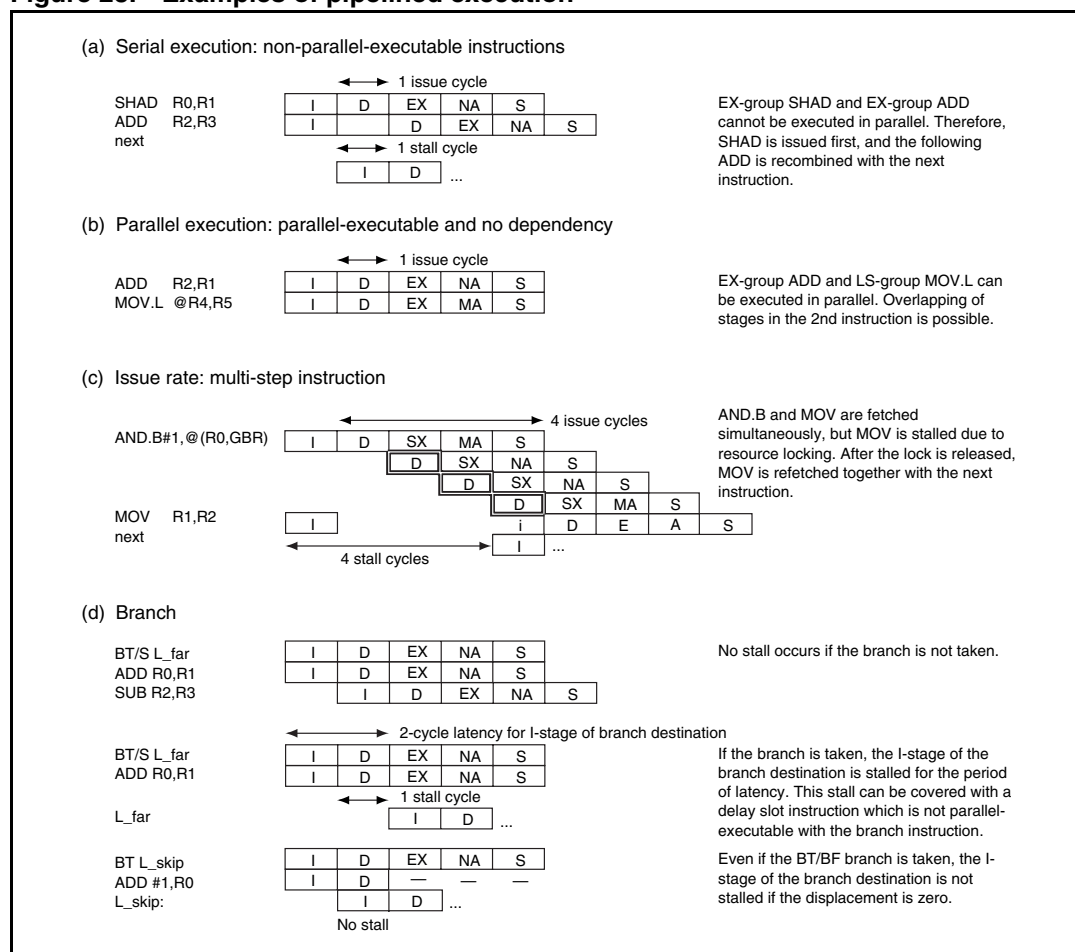


Figure 29. Examples of pipelined execution (continued)

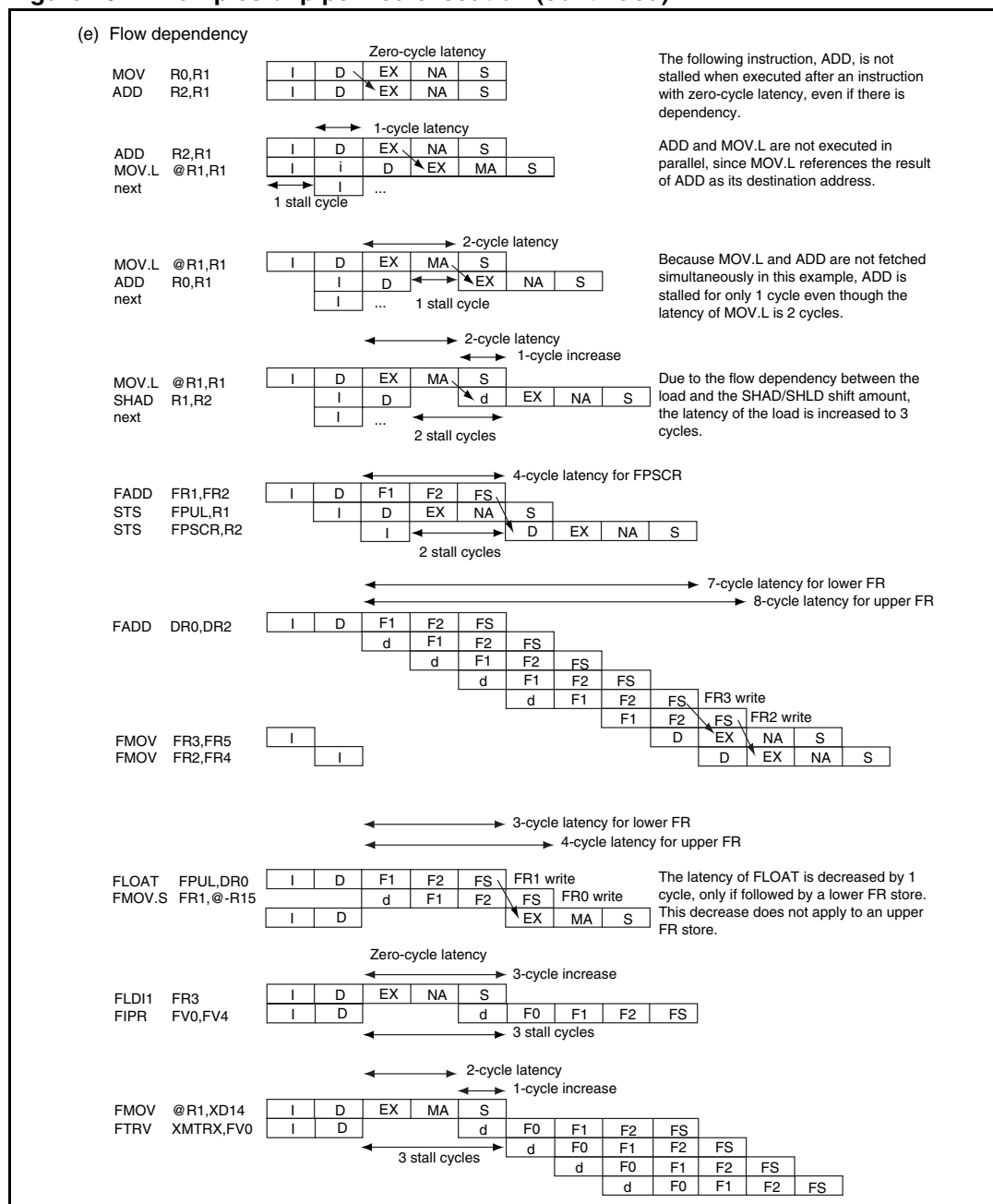
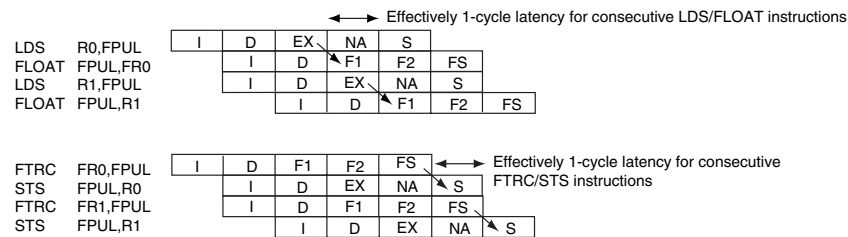
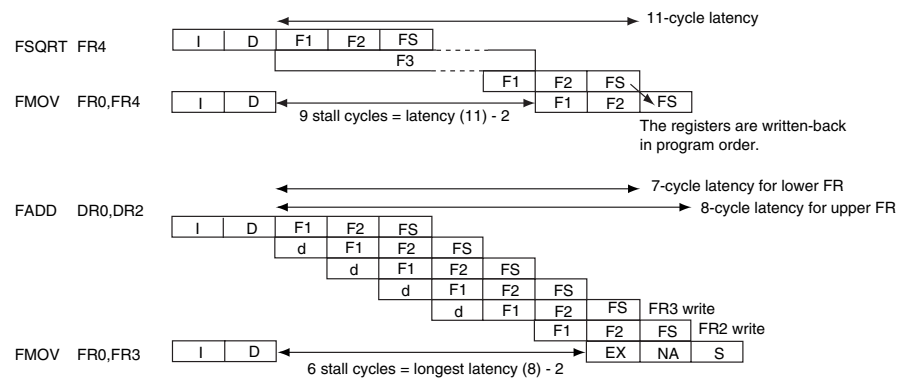


Figure 30. Examples of pipelined execution (continued)

(e) Flow dependency (cont)



(f) Output dependency



(g) Anti-flow dependency

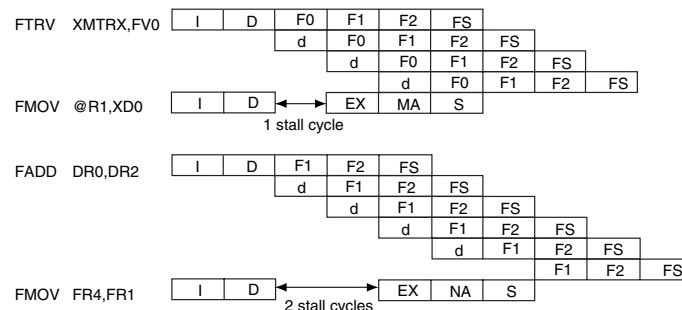


Figure 31. Examples of pipelined execution (continued)

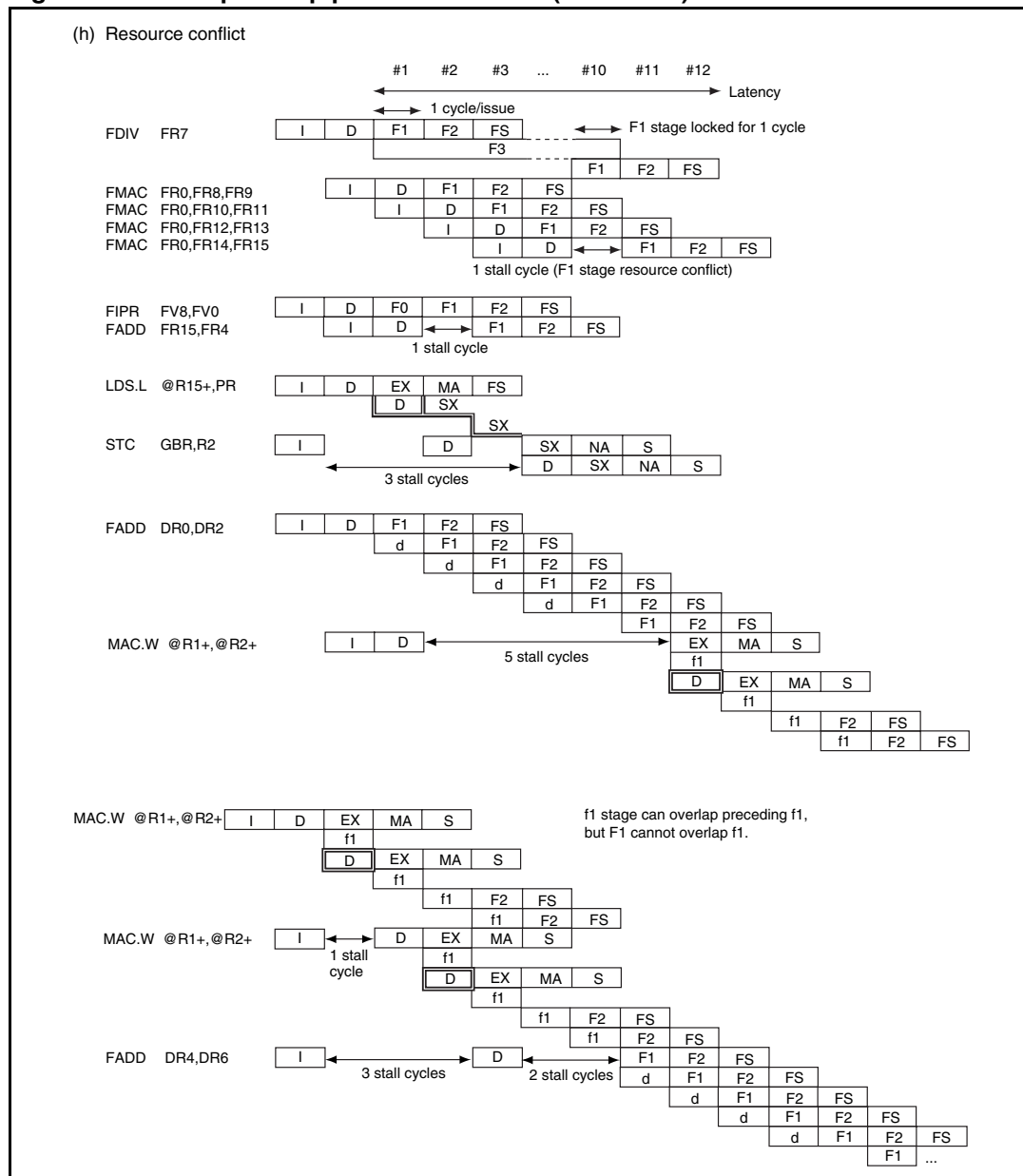


Table 135. Execution cycles

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|----------------------------|-------------|----------------|-------------------|------------|---------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Data transfer instructions | EXTS.B | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | EXTS.W | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | EXTU.B | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | EXTU.W | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | MOV | Rm,Rn | MT | 1 | 0 | #1 | - | - | - |
| | MOV | #imm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | MOVA | @(disp,PC),R0 | EX | 1 | 1 | #1 | - | - | - |
| | MOV.W | @(disp,PC),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.L | @(disp,PC),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.B | @Rm,Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.W | @Rm,Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.L | @Rm,Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.B | @Rm+,Rn | LS | 1 | 1/2 | #2 | - | - | - |
| | MOV.W | @Rm+,Rn | LS | 1 | 1/2 | #2 | - | - | - |
| | MOV.L | @Rm+,Rn | LS | 1 | 1/2 | #2 | - | - | - |
| | MOV.B | @(disp,Rm),R0 | LS | 1 | 2 | #2 | - | - | - |
| | MOV.W | @(disp,Rm),R0 | LS | 1 | 2 | #2 | - | - | - |
| | MOV.L | @(disp,Rm),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.B | @(R0,Rm),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.W | @(R0,Rm),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.L | @(R0,Rm),Rn | LS | 1 | 2 | #2 | - | - | - |
| | MOV.B | @(disp,GBR),R0 | LS | 1 | 2 | #3 | - | - | - |
| | MOV.W | @(disp,GBR),R0 | LS | 1 | 2 | #3 | - | - | - |
| | MOV.L | @(disp,GBR),R0 | LS | 1 | 2 | #3 | - | - | - |
| | MOV.B | Rm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | MOV.W | Rm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | MOV.L | Rm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | MOV.B | Rm,@-Rn | LS | 1 | 1/1 | #2 | - | - | - |
| | MOV.W | Rm,@-Rn | LS | 1 | 1/1 | #2 | - | - | - |
| | MOV.L | Rm,@-Rn | LS | 1 | 1/1 | #2 | - | - | - |
| | MOV.B | R0,@(disp,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | MOV.W | R0,@(disp,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | MOV.L | Rm,@(disp,Rn) | LS | 1 | 1 | #2 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|-------------------------------------|-------------|----------------|-------------------|------------|---------|-------------------|--------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Data transfer instructions | MOV.B | Rm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | MOV.W | Rm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | MOV.L | Rm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | MOV.B | R0,@(disp,GBR) | LS | 1 | 1 | #3 | - | - | - |
| | MOV.W | R0,@(disp,GBR) | LS | 1 | 1 | #3 | - | - | - |
| | MOV.L | R0,@(disp,GBR) | LS | 1 | 1 | #3 | - | - | - |
| | MOVCA.L | R0,@Rn | LS | 1 | 3-7 | #12 | M A | 4 | 3-7 |
| | MOVT | Rn | EX | 1 | 1 | #1 | - | - | - |
| | OCBI | @Rn | LS | 1 | 1-2 | #10 | M A | 4 | 1-2 |
| | OCBP | @Rn | LS | 1 | 1-5 | #11 | M A | 4 | 1-5 |
| | OCBWB | @Rn | LS | 1 | 1-5 | #11 | M A | 4 | 1-5 |
| | PREF | @Rn | LS | 1 | 1 | #2 | - | - | - |
| | SWAP.B | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SWAP.W | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | XTRCT | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| Fixed-point arithmetic instructions | ADD | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | ADD | #imm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | ADDC | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | ADDV | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | CMP/EQ | #imm,R0 | MT | 1 | 1 | #1 | - | - | - |
| | CMP/EQ | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/GE | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/GT | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/HI | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/HS | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/PL | Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/PZ | Rn | MT | 1 | 1 | #1 | - | - | - |
| | CMP/STR | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | DIV0S | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | DIV0U | | EX | 1 | 1 | #1 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|-------------------------------------|-------------|----------------|-------------------|------------|---------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Fixed-point arithmetic instructions | DIV1 | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | DMULS.L | Rm,Rn | CO | 2 | 4/4 | #34 | F1 | 4 | 2 |
| | DMULU.L | Rm,Rn | CO | 2 | 4/4 | #34 | F1 | 4 | 2 |
| | DT | Rn | EX | 1 | 1 | #1 | - | - | - |
| | MAC.L | @Rm+,@Rn+ | CO | 2 | 2/2/4/4 | #35 | F1 | 4 | 2 |
| | MAC.W | @Rm+,@Rn+ | CO | 2 | 2/2/4/4 | #35 | F1 | 4 | 2 |
| | MUL.L | Rm,Rn | CO | 2 | 4/4 | #34 | F1 | 4 | 2 |
| | MULS.W | Rm,Rn | CO | 2 | 4/4 | #34 | F1 | 4 | 2 |
| | MULU.W | Rm,Rn | CO | 2 | 4/4 | #34 | F1 | 4 | 2 |
| | NEG | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | NEGC | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SUB | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SUBC | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SUBV | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| Logical instructions | AND | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | AND | #imm,R0 | EX | 1 | 1 | #1 | - | - | - |
| | AND.B | #imm,@(R0,GBR) | CO | 4 | 4 | #6 | - | - | - |
| | NOT | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | OR | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | OR | #imm,R0 | EX | 1 | 1 | #1 | - | - | - |
| | OR.B | #imm,@(R0,GBR) | CO | 4 | 4 | #6 | - | - | - |
| | TAS.B | @Rn | CO | 5 | 5 | #7 | - | - | - |
| | TST | Rm,Rn | MT | 1 | 1 | #1 | - | - | - |
| | TST | #imm,R0 | MT | 1 | 1 | #1 | - | - | - |
| | TST.B | #imm,@(R0,GBR) | CO | 3 | 3 | #5 | - | - | - |
| | XOR | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | XOR | #imm,R0 | EX | 1 | 1 | #1 | - | - | - |
| | XOR.B | #imm,@(R0,GBR) | CO | 4 | 4 | #6 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|---------------------|-------------|-------|-------------------|------------|----------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Shift instructions | ROTL | Rn | EX | 1 | 1 | #1 | - | - | - |
| | ROTR | Rn | EX | 1 | 1 | #1 | - | - | - |
| | ROTCL | Rn | EX | 1 | 1 | #1 | - | - | - |
| | ROTCR | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHAD | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHAL | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHAR | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLD | Rm,Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLL | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLL2 | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLL8 | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLL16 | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLR | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLR2 | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLR8 | Rn | EX | 1 | 1 | #1 | - | - | - |
| | SHLR16 | Rn | EX | 1 | 1 | #1 | - | - | - |
| Branch instructions | BF | disp | BR | 1 | 2 (or 1) | #1 | - | - | - |
| | BF/S | disp | BR | 1 | 2 (or 1) | #1 | - | - | - |
| | BT | disp | BR | 1 | 2 (or 1) | #1 | - | - | - |
| | BT/S | disp | BR | 1 | 2 (or 1) | #1 | - | - | - |
| | BRA | disp | BR | 1 | 2 | #1 | - | - | - |
| | BRAF | Rn | CO | 2 | 3 | #4 | - | - | - |
| | BSR | disp | BR | 1 | 2 | #14 | SX | 3 | 2 |
| | BSRF | Rn | CO | 2 | 3 | #24 | SX | 3 | 2 |
| | JMP | @Rn | CO | 2 | 3 | #4 | - | - | - |
| | JSR | @Rn | CO | 2 | 3 | #24 | SX | 3 | 2 |
| | RTS | | CO | 2 | 3 | #4 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|-----------------------------|-------------|--------------|-------------------|------------|---------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| System control instructions | NOP | | MT | 1 | 0 | #1 | - | - | - |
| | CLRMAC | | CO | 1 | 3 | #28 | F1 | 3 | 2 |
| | CLRS | | CO | 1 | 1 | #1 | - | - | - |
| | CLRT | | MT | 1 | 1 | #1 | - | - | - |
| | SETS | | CO | 1 | 1 | #1 | - | - | - |
| | SETT | | MT | 1 | 1 | #1 | - | - | - |
| | TRAPA | #imm | CO | 7 | 7 | #13 | - | - | - |
| | RTE | | CO | 5 | 5 | #8 | - | - | - |
| | SLEEP | | CO | 4 | 4 | #9 | - | - | - |
| | LDTLB | | CO | 1 | 1 | #2 | - | - | - |
| | LDC | Rm,DBR | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC | Rm,GBR | CO | 3 | 3 | #15 | SX | 3 | 2 |
| | LDC | Rm,Rp_BANK | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC | Rm,SR | CO | 4 | 4 | #16 | SX | 3 | 2 |
| | LDC | Rm,SGR | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC | Rm,SSR | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC | Rm,SPC | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC | Rm,VBR | CO | 1 | 3 | #14 | SX | 3 | 2 |
| | LDC.L | @Rm+,DBR | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDC.L | @Rm+,GBR | CO | 3 | 3/3 | #18 | SX | 3 | 2 |
| | LDC.L | @Rm+,Rp_BANK | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDC.L | @Rm+,SGR | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDC.L | @Rm+,SR | CO | 4 | 4/4 | #19 | SX | 3 | 2 |
| | LDC.L | @Rm+,SSR | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDC.L | @Rm+,SPC | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDC.L | @Rm+,VBR | CO | 1 | 1/3 | #17 | SX | 3 | 2 |
| | LDS | Rm,MACH | CO | 1 | 3 | #28 | F1 | 3 | 2 |
| | LDS | Rm,MACL | CO | 1 | 3 | #28 | F1 | 3 | 2 |
| | LDS | Rm,PR | CO | 2 | 3 | #24 | SX | 3 | 2 |
| | LDS.L | @Rm+,MACH | CO | 1 | 1/3 | #29 | F1 | 3 | 2 |
| | LDS.L | @Rm+,MACL | CO | 1 | 1/3 | #29 | F1 | 3 | 2 |
| | LDS.L | @Rm+,PR | CO | 2 | 2/3 | #25 | SX | 3 | 2 |
| | STC | DBR,Rn | CO | 2 | 2 | #20 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|--|-------------|--------------|-------------------|------------|---------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| System control instructions continued | STC | SGR,Rn | CO | 3 | 3 | #21 | - | - | - |
| | STC | GBR,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC | Rp_BANK,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC | SR,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC | SSR,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC | SPC,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC | VBR,Rn | CO | 2 | 2 | #20 | - | - | - |
| | STC.L | DBR,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | SGR,@-Rn | CO | 3 | 3/3 | #23 | - | - | - |
| | STC.L | GBR,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | Rp_BANK,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | SR,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | SSR,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | SPC,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STC.L | VBR,@-Rn | CO | 2 | 2/2 | #22 | - | - | - |
| | STS | MACH,Rn | CO | 1 | 3 | #30 | - | - | - |
| | STS | MACL,Rn | CO | 1 | 3 | #30 | - | - | - |
| | STS | PR,Rn | CO | 2 | 2 | #26 | - | - | - |
| | STS.L | MACH,@-Rn | CO | 1 | 1/1 | #31 | - | - | - |
| | STS.L | MACL,@-Rn | CO | 1 | 1/1 | #31 | - | - | - |
| | STS.L | PR,@-Rn | CO | 2 | 2/2 | #27 | - | - | - |
| Single-precision floating-point instructions (only if FPU present) | FLDI0 | FRn | LS | 1 | 0 | #1 | - | - | - |
| | FLDI1 | FRn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV | FRm,FRn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV.S | @Rm,FRn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV.S | @Rm+,FRn | LS | 1 | 1/2 | #2 | - | - | - |
| | FMOV.S | @(R0,Rm),FRn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV.S | FRm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | FMOV.S | FRm,@-Rn | LS | 1 | 1/1 | #2 | - | - | - |
| | FMOV.S | FRm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | FLDS | FRm,FPUL | LS | 1 | 0 | #1 | - | - | - |
| | FSTS | FPUL,FRn | LS | 1 | 0 | #1 | - | - | - |
| | FABS | FRn | LS | 1 | 0 | #1 | - | - | - |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|--|-------------|--------------|-------------------|------------|-------------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Single-precision floating-point instructions continued | FADD | FRm,FRn | FE | 1 | 3/4 | #36 | - | - | - |
| | FCMP/EQ | FRm,FRn | FE | 1 | 2/4 | #36 | - | - | - |
| | FCMP/GT | FRm,FRn | FE | 1 | 2/4 | #36 | - | - | - |
| | FDIV | FRm,FRn | FE | 1 | 12/13 | #37 | F3 | 2 | 10 |
| | | | | | | | F1 | 11 | 1 |
| | FLOAT | FPUL,FRn | FE | 1 | 3/4 | #36 | F1 | 2 | 2 |
| | FMAC | FR0,FRm,FRn | FE | 1 | 3/4 | #36 | - | - | - |
| | FMUL | FRm,FRn | FE | 1 | 3/4 | #36 | - | - | - |
| | FNEG | FRn | LS | 1 | 0 | #1 | - | - | - |
| | FSQRT | FRn | FE | 1 | 11/12 | #37 | F3 | 2 | 9 |
| | | | | | | | F1 | 10 | 1 |
| | FSUB | FRm,FRn | FE | 1 | 3/4 | #36 | - | - | - |
| | FTRC | FRm,FPUL | FE | 1 | 3/4 | #36 | - | - | - |
| Double-precision floating-point instructions (only if FPU present) | FMOV | DRm,DRn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV | @Rm,DRn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV | @Rm+,DRn | LS | 1 | 1/2 | #2 | - | - | - |
| | FMOV | @(R0,Rm),DRn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV | DRm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | FMOV | DRm,@-Rn | LS | 1 | 1/1 | #2 | - | - | - |
| | FMOV | DRm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | FABS | DRn | LS | 1 | 0 | #1 | - | - | - |
| | FADD | DRm,DRn | FE | 1 | (7, 8)/9 | #39 | F1 | 2 | 6 |
| | FCMP/EQ | DRm,DRn | CO | 2 | 3/5 | #40 | F1 | 2 | 2 |
| | FCMP/GT | DRm,DRn | CO | 2 | 3/5 | #40 | F1 | 2 | 2 |
| | FCNVDS | DRm,FPUL | FE | 1 | 4/5 | #38 | F1 | 2 | 2 |
| | FCNVSD | FPUL,DRn | FE | 1 | (3, 4)/5 | #38 | F1 | 2 | 2 |
| | FDIV | DRm,DRn | FE | 1 | (24, 25)/26 | #41 | F3 | 2 | 23 |
| | | | | | | | F1 | 22 | 3 |
| | FLOAT | FPUL,DRn | FE | 1 | (3, 4)/5 | #38 | F1 | 2 | 2 |
| | FMUL | DRm,DRn | FE | 1 | (7, 8)/9 | #39 | F1 | 2 | 6 |
| | FNEG | DRn | LS | 1 | 0 | #1 | - | - | - |
| | FSQRT | DRn | FE | 1 | (23, 24)/25 | #41 | F3 | 2 | 22 |

Table 135. Execution cycles (continued)

| Functional category | Instruction | | Instruction group | Issue rate | Latency | Execution pattern | Lock | | |
|--|-------------|--------------|-------------------|------------|----------------|-------------------|-------|-------|--------|
| | | | | | | | Stage | Start | Cycles |
| Double-precision floating-point instructions | | | | | | | F1 | 21 | 3 |
| | FSUB | DRm,DRn | FE | 1 | (7, 8)/9 | #39 | F1 | 2 | 6 |
| | FTRC | DRm,FPUL | FE | 1 | 4/5 | #38 | F1 | 2 | 2 |
| FPU system control instructions (only if FPU present) | LDS | Rm,FPUL | LS | 1 | 1 | #1 | - | - | - |
| | LDS | Rm,FPSCR | CO | 1 | 4 | #32 | F1 | 3 | 3 |
| | LDS.L | @Rm+,FPUL | CO | 1 | 1/2 | #2 | - | - | - |
| | LDS.L | @Rm+,FPSCR | CO | 1 | 1/4 | #33 | F1 | 3 | 3 |
| | STS | FPUL,Rn | LS | 1 | 3 | #1 | - | - | - |
| | STS | FPSCR,Rn | CO | 1 | 3 | #1 | - | - | - |
| | STS.L | FPUL,@-Rn | CO | 1 | 1/1 | #2 | - | - | - |
| | STS.L | FPSCR,@-Rn | CO | 1 | 1/1 | #2 | - | - | - |
| Graphics acceleration instructions (only if FPU present) | FMOV | DRm,XDn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV | XDm,DRn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV | XDm,XDn | LS | 1 | 0 | #1 | - | - | - |
| | FMOV | @Rm,XDn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV | @Rm+,XDn | LS | 1 | 1/2 | #2 | - | - | - |
| | FMOV | @(R0,Rm),XDn | LS | 1 | 2 | #2 | - | - | - |
| | FMOV | XDm,@Rn | LS | 1 | 1 | #2 | - | - | - |
| | FMOV | XDm,@-Rm | LS | 1 | 1/1 | #2 | - | - | - |
| | FMOV | XDm,@(R0,Rn) | LS | 1 | 1 | #2 | - | - | - |
| | FIPR | FVm,FVn | FE | 1 | 4/5 | #42 | F1 | 3 | 1 |
| | FRCHG | | FE | 1 | 1/4 | #36 | - | - | - |
| | FSCA | FPUL,DRn | FE | 1 | 7/8 | #43 | F0 | 2 | 3 |
| | | | | | | | F1 | 3 | 3 |
| | FSCHG | | FE | 1 | 1/4 | #36 | - | - | - |
| | FSRRA | FRn | FE | 1 | 8 | #43 | F0 | 2 | 3 |
| | | | | | | | F1 | 3 | 3 |
| | FTRV | XMTRX,FVn | FE | 1 | (5, 5, 6, 7)/8 | #43 | F0 | 2 | 4 |
| | | | | | | | F1 | 3 | 4 |

Note: 1 See [Table 133](#) for the instruction groups.

2 Latency “L1/L2...”: latency corresponding to a write to each register, including MACH/MACL/FPSCR.

Example: MOV.B @Rm+, Rn “1/2”: The latency for Rm is 1 cycle, and the latency for Rn is 2 cycles.

- 3 *Branch latency: interval until the branch destination instruction is fetched*
- 4 *Conditional branch latency “2 (or 1)”: the latency is 2 for a nonzero displacement, and 1 for a zero displacement.*
- 5 *Double-precision floating-point instruction latency “(L1, L2)/L3”: L1 is the latency for FR [n+1], L2 that for FR [n], and L3 that for FPSCR.*
- 6 *FTRV latency “(L1, L2, L3, L4)/L5”: L1 is the latency for FR [n], L2 that for FR [n+1], L3 that for FR [n+2], L4 that for FR [n+3], and L5 that for FPSCR.*
- 7 *Latency “L1/L2/L3/L4” of MAC.L and MAC.W instructions: L1 is the latency for Rm, L2 that for Rn, L3 that for MACH, and L4 that for MACL.*
- 8 *Latency “L1/L2” of MUL.L, MULS.W, MULU.W, DMULS.L, and DMULU.L instructions: L1 is the latency for MACH, and L2 that for MACL.*
- 9 *Execution pattern: the instruction execution pattern number (see figure 8.2)*
- 10 *Lock/stage: stage locked by the instruction*
- 11 *Lock/start: locking start cycle; 1 is the first D-stage of the instruction.*
- 12 *Lock/cycles: number of cycles locked.*

Exceptions

1. When a floating-point computation instruction is followed by an FMOV store, an STS FPUL, Rn instruction, or an STS.L FPUL, @-Rn instruction, the latency of the floating-point computation is decreased by 1 cycle.
2. When the preceding instruction loads the shift amount of the following SHAD/SHLD, the latency of the load is increased by 1 cycle.
3. When an LS group instruction with a latency of less than 3 cycles is followed by a double-precision floating-point instruction, FIPR, or FTRV, the latency of the first instruction is increased to 3 cycles.
Example: In the case of FMOV FR4,FR0 and FIPR FV0,FV4, FIPR is stalled for 2 cycles.
4. When MAC*/MUL*/DMUL* is followed by an STS.L MAC*, @-Rn instruction, the latency of MAC*/MUL*/DMUL* is 5 cycles.
5. In the case of consecutive executions of MAC*/MUL*/DMUL*, the latency is decreased to 2 cycles.
6. When an LDS to MAC* is followed by an STS.L MAC*, @-Rn instruction, the latency of the LDS to MAC* is 4 cycles.
7. When an LDS to MAC* is followed by MAC*/MUL*/DMUL*, the latency of the LDS to MAC* is 1 cycle.
8. When an FSCHG or FRCHG instruction is followed by an LS group instruction that reads or writes to a floating-point register, the aforementioned LS group instruction[s] cannot be executed in parallel.
9. When a single-precision FTRC instruction is followed by an STS FPUL, Rn instruction, the latency of the single-precision FTRC instruction is 1 cycle.

11 ST40-300 performance characteristics

The ST40-300 implementation of the ST40 architecture has a dual-issue superscalar 7-stage integer pipeline and a 10-stage floating-point pipe. The other key features of the design are:

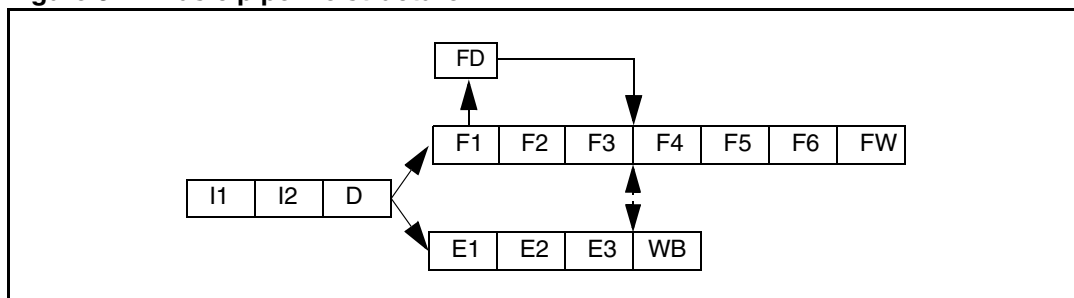
- a Branch Target Cache (BTC) supporting bubble-free branching along the predicted path
- four entry return stack for optimized procedure calling
- 64-bit instruction fetcher which can request 4 instructions from the cache every cycle
- two fully symmetric Arithmetic Logic Units (ALUs) supporting dual-issue of all integer arithmetic operations except multiply
- zero latency integer moves and constant loading
- dedicated integer multiplier
Combined with zero latency MAC, this reduces the total time to execute a multiply and place the result in a general register to just 2 cycles (compared to 7 cycles on all previous implementations).

This chapter gives a high-level description of the performance characteristics of this particular implementation of the ST40 architecture. The intention is to document the implementation in sufficient detail to aid instruction scheduling in the development of compilers and/or handwritten assembler code, not to provide a complete disclosure of the pipelines' micro architecture.

11.1 Basic pipeline structure

[Figure 32](#) shows the basic pipeline structure of the implementation. It is a dual-issue architecture so two instructions can progress simultaneously through any pipe stage provided that dependency and resource constraints are met.

Figure 32. Basic pipeline structure



The head of the pipeline consists of two stages of instruction fetch (I1, I2) which can fetch four instructions from the predicted path every cycle and store them in a queue ready for decode (D). After decode, the pipe splits into integer and floating point pipes. The integer pipe consists of three execute stages (E1, E2, E3) and one writeback stage (WB). The floating point pipe has six execute stages (F1, F2, F3, F4, F5, F6) and one writeback stage (FW). There is one additional FPU pipe stage used for divide and square root (FD). These instructions have a long latency so are offlined to the FD stage where they recirculate until re-inserted in to the main FPU pipe at stage F4.

Note: If an integer instruction is issued in parallel with a floating point one, and no stalls occur, the integer instruction will be written back three cycles earlier than the floating point one. To guarantee architectural preciseness can be maintained, there is an interlock between the F3 and E3 stages which only allows writeback of an instruction in the integer pipe to proceed once all exception checks have been performed on any earlier floating point instructions.

11.2 Issue constraints

Once an instruction reaches decode, its issue to the execute stages can be stalled by any data hazards or structural hazards in the pipe.

Data hazards are predominately read-after-write (RAW) dependencies; the latency of the instruction writing the data indicates the minimum number of cycles before a dependent instruction can be issued. Thus an instruction with a zero latency can be issued in parallel with a dependent instruction. The latencies of each instruction can be found in [Section 11.3: Instruction performance characteristics](#).

Structural hazards occur if any of the resources required to execute an instruction would not be available if the instruction was issued in that cycle. For example, the design only has one load/store unit, so if the two instructions available for issue both require the load/store unit, only the first can be issued in that cycle. However, there are two arithmetic logic units (ALUs) available for executing simple operations such as adds, subtracts, compares and register-register moves, so no structural hazards ever arise for these operations.

The pipe resources over which contention can occur are as follows:

| | |
|------------|---|
| LS | (Load/store) Handles load/store operations. The address calculations and auto-increment/decrement operations required by the load/store operations are handled by one of the ALUs. Additional stall cycles can result when using the LS resource, due to cache misses and uncached accesses. When such a stall occurs, all later instructions are held in their current states until the stall condition is resolved. |
| MUL | (Multiply) Handles multiplies, multiply/accumulate and MACL/MACH register transfer operations. |
| BR | (Branch) Used to perform branching operations. |
| FPT | (Floating point transfer) Used for floating point instructions associated with moving data into and out of the floating-point register file (either between the integer register file or memory). Instructions which use this resource are issued to both the integer and floating point pipe |
| FPU | (Floating point unit) Executes the floating-point arithmetic operations. Instructions which use this resource are issued to both the integer and floating point pipe. |
| FDS | (Floating point divide square root) Implements the iterative phases of the divide and square root operations. |
| ALL | This is a pseudo-resource used in the following descriptions to capture the concept that all resources are occupied by the operation, including the two ALUs. Consequently no further instructions can be issued while this resource is held. |

The resources required to execute each instruction are documented in [Section 11.3](#). Dual issue of two instructions can only occur if there is no clash in their resource requirements.

11.3 Instruction performance characteristics

Performance characteristics of the individual instructions are summarized in [Table 136](#) and in [Table 137 on page 506](#). Penalty cycles due to cache misses are not considered in these tables. The meaning of the individual columns is as follows:

| | |
|-----------------------|---|
| Resource usage | List of resources over which contention can occur when issuing the instruction. Some instructions occupy the resource for more than one cycle, stalling issue of a subsequent instruction if it is dependent upon the same resource. This multi-cycle use is indicated by suffixing the resource with a colon and a number (for example LS:2). Use of the ALL resource indicates that no further instructions can be issued whilst this resource is in use. |
| Latency | <p>Number of cycles between the issue of an instruction and when a instruction which depends on one of its results can be issued.</p> <p>If more than one piece of architectural state is updated by the instruction, the list includes the register name to which the result availability refers.</p> <p>A zero latency indicates that the instruction can be issued in parallel with a dependent instruction.</p> <p>A negative latency is associated with a source operand. It indicates that the operand is read later in the pipe and therefore the instruction upon which it depends has its effective latency reduced by that amount. To emphasize that these are relevant only to source operands the latency is parenthesized, for example, (R0:-2).</p> |
| Constraints | <p>Contains additional information about the instruction's execution profile, for example:</p> <ul style="list-style-type: none"> Some instructions cause a refetch of the next instruction when they reach the writeback stage. The consequence of this is that issue of <i>any</i> further instruction is stalled for the specified latency of the instruction. Some instructions, such as the atomic read-modify-write operations, occupy multiple issue slots. These issue slots can never be paired together, however they can be paired with either the previous or next instruction as long as there is no contention for resources. |

11.3.1 Integer instructions

All the following instructions are issued solely to the integer pipeline.

Table 136. Integer instructions

| Instruction | | Resource Usage | Latency | Notes |
|-------------|----------------|----------------|--|------------------------|
| ADD | Rm,Rn | - | 1 | - |
| ADD | #imm,Rn | - | 1 | - |
| ADDC | Rm,Rn | - | Rn:1, T:1 | - |
| ADDV | Rm,Rn | - | Rn:1, T:1 | - |
| AND | Rm,Rn | - | 1 | - |
| AND | #imm,R0 | - | 1 | - |
| AND.B | #imm,@(R0,GBR) | LS:2 | 0 | Occupies 2 issue slots |
| BF | label | BR | <i>Section 11.3.3: Branching scenarios on page 508</i> | |
| BF/S | label | BR | | |
| BRA | label | BR | | |
| BRAF | Rn | BR | | |
| BSR | label | BR | | |
| BSRF | Rn | BR | | |
| BT | label | BR | | |
| BT/S | label | BR | | |
| CLRMAC | | MUL | 1 | - |
| CLRS | | MUL | 1 | - |
| CLRT | | - | 1 | - |
| CMP/EQ | #imm,R0 | - | 1 | - |
| CMP/EQ | Rm,Rn | - | 1 | - |
| CMP/GE | Rm,Rn | - | 1 | - |
| CMP/GT | Rm,Rn | - | 1 | - |
| CMP/HI | Rm,Rn | - | 1 | - |
| CMP/HS | Rm,Rn | - | 1 | - |
| CMP/PL | Rn | - | 1 | - |
| CMP/PZ | Rn | - | 1 | - |
| CMP/STR | Rm,Rn | - | 1 | - |
| DIV0S | Rm,Rn | - | 1 | - |
| DIV0U | | - | 1 | - |
| DIV1 | Rm,Rn | - | 1 | - |
| DMULS.L | Rm,Rn | MUL | MACL:2, MACH:3 | - |
| DMULU.L | Rm,Rn | MUL | MACL:2, MACH:3 | - |

Table 136. Integer instructions (continued)

| Instruction | | Resource Usage | Latency | Notes |
|-------------|--------------|-------------------------|---|---|
| DT | Rn | - | 1 | - |
| EXTS.B | Rm,Rn | - | 1 | - |
| EXTS.W | Rm,Rn | - | 1 | - |
| EXTU.B | Rm,Rn | - | 1 | - |
| EXTU.W | Rm,Rn | - | 1 | - |
| ICBI | @Rm | LS:1 then ALL:7 | 8 ⁽¹⁾ | Refetches next instruction on writeback |
| JMP | @Rn | BR | Section 11.3.3: Branching scenarios on page 508 | |
| JSR | @Rn | BR | | |
| LDC | Rm,DBR | - | 1 | |
| LDC | Rm,GBR | - | 1 | |
| LDC | Rm,Rn_BANK | - | 1 | |
| LDC | Rm,SR | None:1 then ALL:7 | 8 ⁽¹⁾ | Refetches next instruction on writeback |
| LDC | Rm,SGR | - | 1 | |
| LDC | Rm,SSR | - | 1 | |
| LDC | Rm,SPC | - | 1 | |
| LDC | Rm,VBR | - | 1 | |
| LDC.L | @Rm+,DBR | LS | Rm:1, DBR:3 | |
| LDC.L | @Rm+,GBR | LS | Rm:1, GBR:3 | |
| LDC.L | @Rm+,Rn_BANK | LS | Rm:1, Rn_BANK:3 | |
| LDC.L | @Rm+,SGR | LS | Rm:1, SGR:3 | |
| LDC.L | @Rm+,SR | LS:1 then ALL:10 | 11 ⁽¹⁾ | Refetches next instruction on writeback |
| LDC.L | @Rm+,SSR | LS | Rm:1, SSR:3 | |
| LDC.L | @Rm+,SPC | LS | Rm:1, SPC:3 | |
| LDC.L | @Rm+,VBR | LS | Rm:1, VBR:3 | |
| LDS | Rm,MACH | MUL | 1 | - |
| LDS | Rm,MACL | MUL | 1 | - |
| LDS | Rm,PR | - | 1 | - |
| LDS.L | @Rm+,MACH | MUL, LS | Rm:1, MACH:3 | - |
| LDS.L | @Rm+,MACL | MUL, LS | Rm:1, MACL:3 | - |
| LDS.L | @Rm+,PR | LS | Rm:1, PR:3 | - |
| LDTLB | | LS:1 then perhaps ALL:7 | - or 8 ⁽¹⁾ | May refetch next instruction on writeback |

Table 136. Integer instructions (continued)

| Instruction | | Resource Usage | Latency | Notes |
|-------------|----------------|-------------------------------|------------------------------------|------------------------|
| MAC.L | @Rm+, @Rn+ | MUL:2/5 ⁽²⁾ , LS:2 | Rm:1, Rn:2, MAC:2/4 ⁽³⁾ | Occupies 2 issue slots |
| MAC.W | @Rm+, @Rn+ | MUL:2/5 ⁽²⁾ , LS:2 | Rm:1, Rn:2, MAC:2/4 ⁽³⁾ | Occupies 2 issue slots |
| MOV | Rm,Rn | - | 0 | - |
| MOV | #imm,Rn | - | 0 | - |
| MOVA | @(disp,PC),R0 | - | 1 | - |
| MOV.W | @(disp,PC),Rn | LS | 3 | - |
| MOV.L | @(disp,PC),Rn | LS | 3 | - |
| MOV.B | @Rm,Rn | LS | 3 | - |
| MOV.W | @Rm,Rn | LS | 3 | - |
| MOV.L | @Rm,Rn | LS | 3 | - |
| MOV.B | @Rm+,Rn | LS | Rm:1,Rn:3 | - |
| MOV.W | @Rm+,Rn | LS | Rm:1,Rn:3 | - |
| MOV.L | @Rm+,Rn | LS | Rm:1,Rn:3 | - |
| MOV.B | @(disp,Rm),R0 | LS | 3 | - |
| MOV.W | @(disp,Rm),R0 | LS | 3 | - |
| MOV.L | @(disp,Rm),Rn | LS | 3 | - |
| MOV.B | @(R0,Rm),Rn | LS | 3 | - |
| MOV.W | @(R0,Rm),Rn | LS | 3 | - |
| MOV.L | @(R0,Rm),Rn | LS | 3 | - |
| MOV.B | @(disp,GBR),R0 | LS | 3 | - |
| MOV.W | @(disp,GBR),R0 | LS | 3 | - |
| MOV.L | @(disp,GBR),R0 | LS | 3 | - |
| MOV.B | Rm,@Rn | LS | (Rm:-2) | - |
| MOV.W | Rm,@Rn | LS | (Rm:-2) | - |
| MOV.L | Rm,@Rn | LS | (Rm:-2) | - |
| MOV.B | Rm,@-Rn | LS | (Rm:-2), Rn:1 | - |
| MOV.W | Rm,@-Rn | LS | (Rm:-2), Rn:1 | - |
| MOV.L | Rm,@-Rn | LS | (Rm:-2), Rn:1 | - |
| MOV.B | R0,@(disp,Rn) | LS | (R0:-2) | - |
| MOV.W | R0,@(disp,Rn) | LS | (R0:-2) | - |
| MOV.L | Rm,@(disp,Rn) | LS | (Rm:-2) | - |
| MOV.B | Rm,@(R0,Rn) | LS | (Rm:-2) | - |
| MOV.W | Rm,@(R0,Rn) | LS | (Rm:-2) | - |
| MOV.L | Rm,@(R0,Rn) | LS | (Rm:-2) | - |
| MOV.B | R0,@(disp,GBR) | LS | (R0:-2) | - |

Table 136. Integer instructions (continued)

| Instruction | | Resource Usage | Latency | Notes |
|-------------|------------------|----------------|---|------------------------|
| MOV.W | R0, @ (disp,GBR) | LS | (R0:-2) | - |
| MOV.L | R0, @ (disp,GBR) | LS | (R0:-2) | - |
| MOVCA.L | R0, @Rn | LS | - | - |
| MOVT | Rn | - | 1 | - |
| MUL.L | Rm,Rn | MUL | 2 | - |
| MULS.W | Rm,Rn | MUL | 2 | - |
| MULU.W | Rm,Rn | MUL | 2 | - |
| NEG | Rm,Rn | - | 1 | - |
| NEGC | Rm,Rn | - | 1 | - |
| NOP | | - | - | - |
| NOT | Rm,Rn | - | 1 | - |
| OCBI | @Rn | LS | - | - |
| OCBP | @Rn | LS | - | - |
| OCBWB | @Rn | LS | - | - |
| OR | Rm,Rn | - | 1 | - |
| OR | #imm,R0 | - | 1 | - |
| OR.B | #imm, @ (R0,GBR) | LS:2 | - | Occupies 2 issue slots |
| PREF | @Rn | LS | - | - |
| ROTL | Rn | - | Rn:1, T:1 | - |
| ROTR | Rn | - | Rn:1, T:1 | - |
| ROTCL | Rn | - | Rn:1, T:1 | - |
| ROTCR | Rn | - | Rn:1, T:1 | - |
| RTE | | BR | See Section 11.3.3: Branching scenarios on page 508 | |
| RTS | | BR | | |
| SETS | | MUL | 1 | - |
| SETT | | - | 1 | - |
| SHAD | Rm,Rn | - | 1 | - |
| SHAL | Rn | - | Rn:1, T:1 | - |
| SHAR | Rn | - | Rn:1, T:1 | - |
| SHLD | Rm,Rn | - | 1 | - |
| SHLL | Rn | - | Rn:1, T:1 | - |
| SHLL2 | Rn | - | 1 | - |
| SHLL8 | Rn | - | 1 | - |
| SHLL16 | Rn | - | 1 | - |
| SHLR | Rn | - | Rn:1, T:1 | - |

Table 136. Integer instructions (continued)

| Instruction | | Resource Usage | Latency | Notes |
|-------------|--------------|----------------|------------------------|--|
| SHLR2 | Rn | - | 1 | - |
| SHLR8 | Rn | - | 1 | - |
| SHLR16 | Rn | - | 1 | - |
| SLEEP | | - | Stalls until interrupt | |
| STC | DBR,Rn | - | 0 | - |
| STC | SGR,Rn | - | 0 | - |
| STC | GBR,Rn | - | 0 | - |
| STC | Rm_BANK,Rn | - | 0 | - |
| STC | SR,Rn | - | 0 | - |
| STC | SSR,Rn | - | 0 | - |
| STC | SPC,Rn | - | 0 | - |
| STC | VBR,Rn | - | 0 | - |
| STC.L | DBR,@-Rn | LS | (DBR:-2), Rn:1 | - |
| STC.L | SGR,@-Rn | LS | (SGR:-2), Rn:1 | - |
| STC.L | GBR,@-Rn | LS | (GBR:-2), Rn:1 | - |
| STC.L | Rm_BANK,@-Rn | LS | (Rm_BANK:-2), Rn:1 | - |
| STC.L | SR,@-Rn | LS | (SR:-2), Rn:1 | - |
| STC.L | SSR,@-Rn | LS | (SSR:-2), Rn:1 | - |
| STC.L | SPC,@-Rn | LS | (SPC:-2), Rn:1 | - |
| STC.L | VBR,@-Rn | LS | (VBR:-2), Rn:1 | - |
| STS | MACH,Rn | MUL | 0 | - |
| STS | MACL,Rn | MUL | 0 | - |
| STS | PR,Rn | - | 0 | - |
| STS.L | MACH,@-Rn | LS, MUL | (MACH:-2), Rn:1 | - |
| STS.L | MACL,@-Rn | LS, MUL | (MACL:-2), Rn:1 | - |
| STS.L | PR,@-Rn | LS | (PR:-2), Rn:1 | - |
| SUB | Rm,Rn | - | 1 | - |
| SUBC | Rm,Rn | - | Rn:1, T:1 | - |
| SUBV | Rm,Rn | - | Rn:1, T:1 | - |
| SWAP.B | Rm,Rn | - | 1 | - |
| SWAP.W | Rm,Rn | - | 1 | - |
| SYNCO | | LS | - | Suspends all resources until no accesses are outstanding |
| TAS.B | @Rn | LS:3 | Varies ⁽⁴⁾ | Occupies 5 issue slots |

Table 136. Integer instructions (continued)

| Instruction | | Resource Usage | Latency | Notes |
|-------------|----------------|-------------------|------------------|---|
| TRAPA | #imm | None:1 then ALL:7 | 8 ⁽¹⁾ | Refetch on WB stalls all further instructions |
| TST | Rm,Rn | - | 1 | - |
| TST | #imm,R0 | - | 1 | - |
| TST.B | #imm,@(R0,GBR) | LS:1 | T:4 | Occupies 4 issue slots |
| XOR | Rm,Rn | - | 1 | - |
| XOR | #imm,R0 | - | 1 | - |
| XOR.B | #imm,@(R0,GBR) | LS:2 | - | Occupies 2 issue slots |
| XTRCT | Rm,Rn | - | 1 | - |

1. For instructions which can cause a refetch (ICBI, LDC Rm, SR, LDC @Rm+, SR, TRAPA and RTE) the latency is the number of cycles before any other instruction (apart from the delay slot of the RTE) is issued. Whether LDTLB causes a refetch of the next instruction depends on the setting of the IRMCr (See [Section 3.8.7: Instruction refetch inhibit control \(IRMCr\) on page 68](#)).
2. When executing MAC.(LW) the MUL resource is held for only 2 cycles if required for another MAC but 5 cycles for other MUL operations.
3. There is additional forwarding logic in the multiply pipe to allow a contiguous sequence of MAC.W and/or MAC.L instructions to be issued every two cycles.
4. The TAS.B instruction always accesses external memory, the latency is therefore dependent upon the memory system to which the core is attached.

11.3.2 Floating point instructions

Two instructions can be issued to the floating-point pipe in a single cycle, however, to do so, they must not depend on the same floating-point resource (FPU or FPT).

All floating point instructions are issued to both the integer and the floating point pipes. At the end of E3/F3 there is a synchronization between the two pipes, which ensures that:

- the floating point instruction will not except when it writes back 4 cycles later at the end of FW
- any earlier integer instructions are ready to complete, and in particular, are not stalled on a cache miss

FPSCR handling

All instructions which use the arithmetic floating-point resource (FPU) have implicit read and write dependencies upon the FPSCR register which could potentially prevent any pipelining of floating point calculations. However, the implicit reads are on the RM, Enable, DN, PR and FR, while the writes are to the Cause and Flag fields. The pipeline therefore manages the dependencies on the read and written fields independently to allow full pipelining of most floating-point operations.

Most instructions which use the FPT resource have implicit read dependencies on the FPSCR.SZ and FPSCR.FR fields but no write dependencies. The only instructions with read and write dependencies on the FPSCR are those which toggle the PR, FR and SZ bits (FPCHG, FRCHG and FSCHG respectively). These are all handled by the pipe as a special case, with the new value of the FPSCR after the toggling operation being available with a single cycle latency.

For instructions which explicitly read the FPSCR (STS FPSCR, Rn and STS.L FPSCR, @-Rn) the FPSCR is not available until all previous instructions which update the FPSCR have reached writeback (hence the 7 cycle FPSCR latencies for the FPU resourced instructions in [Table 137](#)). For instructions which explicitly write the FPSCR, no floating-point instruction can be issued until after the FPSCR has been updated (hence the 8 cycle latency of LDS Rm,FPSCR and LDS.L @Rm+,FPSCR).

Multi-cycle FPU instructions

There are a number of instructions which require use of the FPU resource in F1 for multiple cycles. These are FSCA, FSRRA, FTRV and double-precision FMUL.

The FSCA is handled slightly differently to the other three, as it occupies E1 for 2 cycles, stalling issue of any latter instruction by one cycle. The other instructions occupy a single issue slot, the only resource hazard being that on F1:FPU, thus later independent FPT and integer instructions can still be issued.

Table 137. Floating point instructions

| Instruction | | Resource Usage | Latency | Constraints |
|-------------|-------------|----------------|--------------------|-------------|
| FABS | DRn | FPT | 1 | - |
| FABS | FRn | FPT | 1 | - |
| FADD | DRm,DRn | FPU | DRn:6, FPSCR:7 | - |
| FADD | FRm,FRn | FPU | FRn:6, FPSCR:7 | - |
| FCMP/EQ | DRm,DRn | FPU | T:3, FPSCR:7 | - |
| FCMP/EQ | FRm,FRn | FPU | T:3, FPSCR:7 | - |
| FCMP/GT | DRm,DRn | FPU | T:3, FPSCR:7 | - |
| FCMP/GT | FRm,FRn | FPU | T:3, FPSCR:7 | - |
| FCNVDS | DRm,FPUL | FPU | FPUL:6, FPSCR:7 | - |
| FCNVSD | FPUL,DRn | FPU | DRn:6, FPSCR:7 | - |
| FDIV | DRm,DRn | FPU, FDS:32 | DRn:35, FPSCR:36 | - |
| FDIV | FRm,FRn | FPU, FDS:16 | FRn:19, FPSCR:20 | - |
| FIPR | FVm,FVn | FPU | FR(n+3):6, FPSCR:7 | - |
| FLDI0 | FRn | FPT | 1 | - |
| FLDI1 | FRn | FPT | 1 | - |
| FLDS | FRm,FPUL | FPT | 1 | - |
| FLOAT | FPUL,DRn | FPU | DRn:6, FPSCR:7 | - |
| FLOAT | FPUL,FRn | FPU | FRn:6, FPSCR:7 | - |
| FMAC | FR0,FRm,FRn | FPU | FRn:6, FPSCR:7 | - |
| FMOV | FRm,FRn | FPT | 1 | - |
| FMOV | DRm,DRn | FPT | 1 | - |
| FMOV | @Rm,DRn | FPT, LS | 4 | - |
| FMOV | @Rm+,DRn | FPT, LS | Rm:1, DRn:4 | - |

Table 137. Floating point instructions (continued)

| Instruction | | Resource Usage | Latency | Constraints |
|-------------|--------------|----------------|------------------------------|------------------------|
| FMOV | @(R0,Rm),DRn | FPT, LS | 4 | - |
| FMOV | DRm,@Rn | FPT, LS | - | - |
| FMOV | DRm,@-Rn | FPT, LS | Rn:1 | - |
| FMOV | DRm,@(R0,Rn) | FPT, LS | - | - |
| FMOV | DRm,XDn | FPT | 1 | - |
| FMOV | XDm,DRn | FPT | 1 | - |
| FMOV | XDm,XDn | FPT | 1 | - |
| FMOV | @Rm,XDn | FPT, LS | 4 | - |
| FMOV | @Rm+,XDn | FPT, LS | Rm:1, XDn:4 | - |
| FMOV | @(R0,Rm),XDn | FPT, LS | XDn:4 | - |
| FMOV | XDm,@Rn | FPT, LS | - | - |
| FMOV | XDm,@-Rn | FPT, LS | Rn:1 | - |
| FMOV | XDm,@(R0,Rn) | FPT, LS | - | - |
| FMOV.S | @Rm,FRn | FPT, LS | 4 | - |
| FMOV.S | @Rm+,FRn | FPT, LS | Rm:1, FRn:4 | - |
| FMOV.S | @(R0,Rm),FRn | FPT, LS | FRn:4 | - |
| FMOV.S | FRm,@Rn | FPT, LS | - | - |
| FMOV.S | FRm,@-Rn | FPT, LS | Rn:1 | - |
| FMOV.S | FRm,@(R0,Rn) | FPT, LS | - | - |
| FMUL | DRm,DRn | FPU:4 | DRn:9, FPSCR:10 | - |
| FMUL | FRm,FRn | FPU | FRn:6, FPSCR:7 | - |
| FNEG | DRn | FPT | 1 | - |
| FNEG | FRn | FPT | 1 | - |
| FPCHG | | FPU, FPT | 1 | - |
| FRCHG | | FPU, FPT | 1 | - |
| FSCA | FPUL,DRn | FPU:10 | FRn:10, FR(n+1):15, FPSCR:16 | Occupies 2 issue slots |
| FSCHG | | FPU, FPT | 1 | - |
| FSQRT | DRn | FPU, FDS:32 | DRn:35, FPSCR:36 | - |
| FSQRT | FRn | FPU, FDS:16 | FRn:19, FPSCR:20 | - |
| FSRRA | FRn | FPU:5 | FRn:10, FPSCR:11 | - |
| FSTS | FPUL,FRn | FPT | 1 | - |
| FSUB | DRm,DRn | FPU | DRn:6, FPSCR:7 | - |
| FSUB | FRm,FRn | FPU | FRn:6, FPSCR:7 | - |
| FTRC | DRm,FPUL | FPU | FPUL:6, FPSCR:7 | - |
| FTRC | FRm,FPUL | FPU | FPUL:6, FPSCR:7 | - |

Table 137. Floating point instructions (continued)

| Instruction | | Resource Usage | Latency | Constraints |
|-------------|------------|----------------|--|-------------|
| FTRV | XMTRX,FVn | FPU:4 | FRn:6, FR(n+1):7, FR(n+2):8, FR(n+3):9, FPSCR:10 | - |
| LDS | Rm,FPUL | FPT | 2 | - |
| LDS | Rm,FPSCR | FPU, FPT | 8 | - |
| LDS.L | @Rm+,FPUL | FPT, LS | Rm:1, FPUL:4 | - |
| LDS.L | @Rm+,FPSCR | FPU, FPT, LS | Rm:1, FPSCR:8 | - |
| STS | FPUL,Rn | FPT | 1 | - |
| STS | FPSCR,Rn | FPU, FPT | 1 | - |
| STS.L | FPUL,@-Rn | FPT, LS | Rn:1 | - |
| STS.L | FPSCR,@-Rn | FPU, FPT, LS | Rn:1 | - |

11.3.3 Branching scenarios

The effective latency of a particular branching operation varies according to the branching scenario. The scenarios depends upon:

- the type of instruction
- how the branch pairs with any delay slot instruction (if any)
- whether the branch instruction fetch hits the Branch Target Cache (BTC)
- in the case of conditionals, whether the branch is correctly predicted
- whether the predicted target address in the BTC matches that computed by the instruction

[Table 138](#) defines the branch latency in each possible scenario. The *latency* of a branching operation is defined for this purpose as the number of cycles from the issue of the branch instruction to the issue of the next architected instruction, *excluding* the delay slot instruction if there is one. Thus a latency of 0 indicates that the branch and the next architected instruction can be issued together. A latency of 1 indicates that the next architected instruction is issued on the cycle following the branch issue.

Table 138. Branch latencies

| Branch instruction | BTC ⁽¹⁾ | Branch is taken or not taken | BTC target address correct | Return stack target address correct | Delay slot issued in same cycle as branch | Branch latency |
|--------------------|--------------------|------------------------------|----------------------------|-------------------------------------|---|----------------|
| BF, BT | Miss | Taken | | | | 4 |
| | | Not taken | | | | 0 |
| | Hit | Taken | | | | 1 |
| | | Not taken | | | | 4 |

Table 138. Branch latencies (continued)

| Branch instruction | BTC ⁽¹⁾ | Branch is taken or not taken | BTC target address correct | Return stack target address correct | Delay slot issued in same cycle as branch | Branch latency |
|----------------------|--------------------|------------------------------|----------------------------|-------------------------------------|---|----------------|
| BF/S, BT/S | Miss | Taken | | | | 4 |
| | | Not taken | | | | 1 |
| | Hit | Taken | | | Yes | 1 |
| | | No | | | 2 | |
| | | Not taken | | | | 4 |
| BRA, BSR | Miss | Taken | | | | 4 |
| | Hit | Taken | | | Yes | 1 |
| | | No | | | 2 | |
| BRAf, BSRf, JMP, JSR | Miss | Taken | | | | 4 |
| | Hit | Taken | Yes | | Yes | 1 |
| | | | No | | 2 | |
| | | | No | | | |
| RTS | Miss | Taken | | | | 4 |
| | Hit | Taken | | Yes | Yes | 1 |
| | | | | No | 2 | |
| | | | | No | | 5 |
| RTE | | | | | | 9 |

1. The BTC implementation in the ST40-300 always predicts that a branch is taken if a hit occurs.

The PR update as a result of a JSR, BSRf or BSR execution is fully-forwarded and is made available with a single cycle of latency.

To avoid any unsafe situations where a virtual address previously mapped to instruction memory now maps to I/O registers, the BTC is automatically flushed on any operation which could affect the translation process.

- Any stores to the memory-mapped ITLB, UTLB, PMB arrays.
- Stores to MMUCR, PASCR or PTEH (contains current ASID).
- Execution of an LDTLB instruction.

The return stack is flushed on any exception/interrupt launch or refetch.

12 User break controller (UBC)

12.1 Overview

The user break controller (UBC) provides functions that simplify program debugging. When break conditions are set in the UBC, a user break interrupt is generated according to the contents of the bus cycle generated by the CPU. This function makes it easy to design an effective self-monitoring debugger, enabling programs to be debugged with the chip alone, without using an in-circuit emulator.

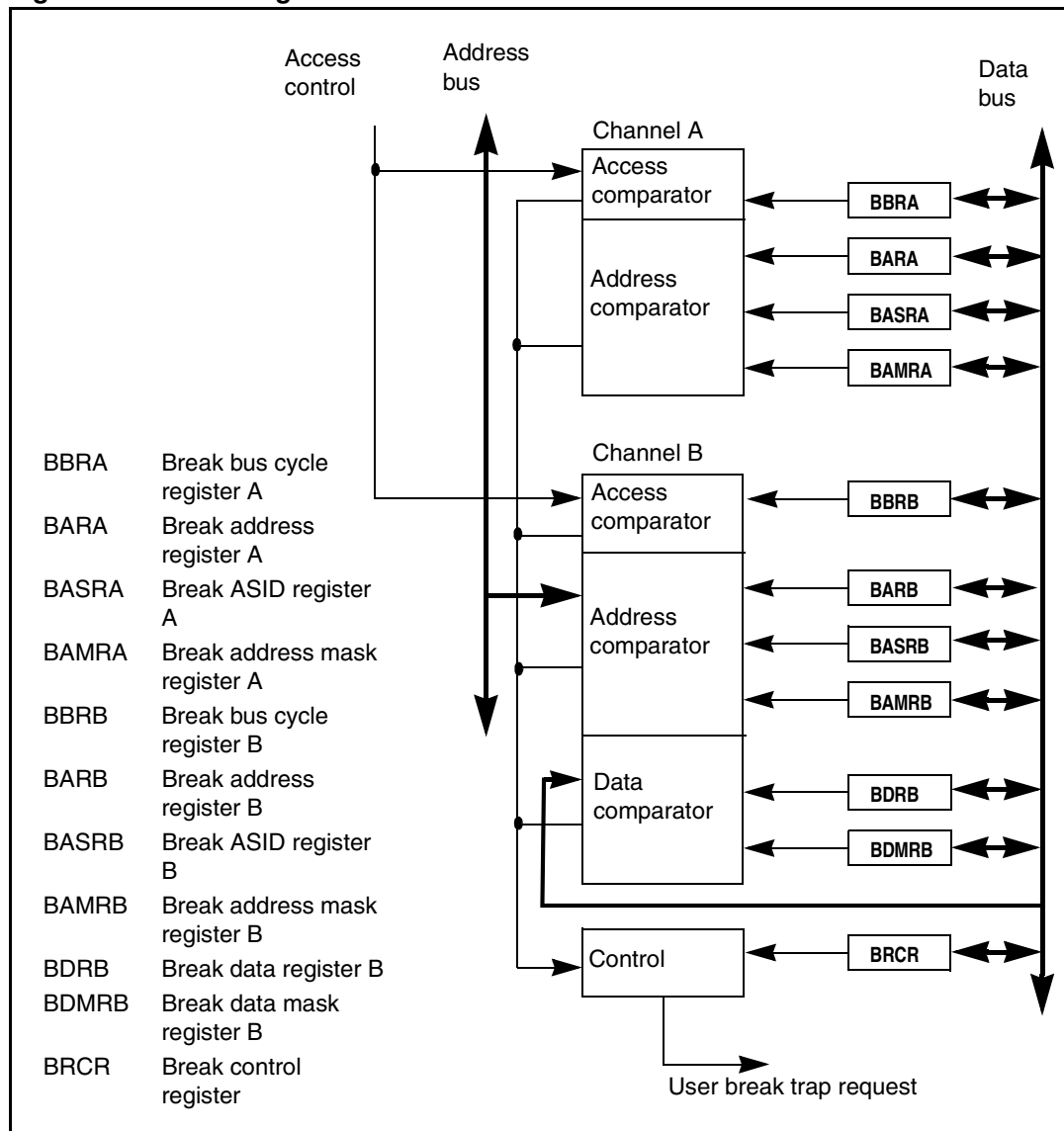
12.1.1 Features

- Two break channels (A and B): user break interrupts can be generated on independent conditions for channels A and B, or on sequential conditions (sequential break setting: channel A then channel B).
- Five break compare conditions:
 - address (selection of 32-bit virtual address and ASID for comparison)
 - data: channel B only, 32-bit mask capability
 - bus cycle: instruction access or operand access
 - read/write
 - operand size: byte, word, long-word, quad-word
- Address comparison has the following options:
 - all bits compared
 - lower 10 bits masked
 - lower 12 bits masked
 - lower 16 bits masked
 - lower 20 bits masked
 - all bits masked
- ASID comparison has the following options:
 - all bits compared
 - all bits masked
- An instruction access cycle break can be effected before or after the instruction is executed.

12.1.2 Block diagram

Figure 33 shows a block diagram of the UBC.

Figure 33. Block diagram of user break controller



12.2 Register overview

[Table 139](#) shows a summary of the UBC registers.

Table 139. UBC registers

| Register name | Description | Type | P4 virtual address ⁽¹⁾ | Access size | Initial value |
|---------------|--|------|-----------------------------------|-------------|---------------|
| UBC.BARA | Break address register A, see Section 12.3.2: Break address register A (UBC.BARA) on page 513 | RW | 0xFF200000 | 32 | Undefined |
| UBC.BAMRA | Break address mask register A, see Section 12.3.4: Break address mask register A (UBC.BAMRA) on page 514 | RW | 0xFF200004 | 8 | Undefined |
| UBC.BBRA | Break bus cycle register A, see Section 12.3.5: Break bus cycle register A (UBC.BBRA) on page 516 | RW | 0xFF200008 | 16 | 0x0000 |
| UBC.BASRA | Break ASID register A, see Section 12.3.3: Break ASID register A (UBC.BASRA) on page 514 | RW | 0xFF000014 | 8 | Undefined |
| UBC.BARB | Break address register B, see Section 12.3.6: Break address register B (UBC.BARB) on page 517 | RW | 0xFF20000C | 32 | Undefined |
| UBC.BAMRB | Break address mask register B, see Section 12.3.8: Break address mask register B (UBC.BAMRB) on page 517 | RW | 0xFF200010 | 8 | Undefined |
| UBC.BBRB | Break bus cycle register B, see Section 12.3.11: Break bus cycle register B (UBC.BBRB) on page 518 | RW | 0xFF200014 | 16 | 0x0000 |
| UBC.BASRB | Break ASID register B, see Section 12.3.7: Break ASID register B (UBC.BASRB) on page 517 | RW | 0xFF000018 | 8 | Undefined |
| UBC.BDRB | Break data register B, see Section 12.3.9: Break data register B (UBC.BDRB) on page 517 | RW | 0xFF200018 | 32 | Undefined |

Table 139. UBC registers (continued)

| Register name | Description | Type | P4 virtual address ⁽¹⁾ | Access size | Initial value |
|---------------|---|------|-----------------------------------|-------------|-----------------------|
| UBC.BDMRB | Break data mask register B, see Section 12.3.10: Break data mask register B (UBC.BDMRB) on page 518 | RW | 0xFF20001C | 32 | Undefined |
| UBC.BRCR | Break control register, see Section 12.3.12: Break control register (UBC.BRCR) on page 518 | RW | 0xFF200020 | 16 | 0x0000 ⁽²⁾ |

1. These registers may also be accessed using translations from other virtual address regions. See [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#).
2. Some bits are not initialized. See [Section 12.3.12: Break control register \(UBC.BRCR\) on page 518](#), for details.

12.3 Register descriptions

12.3.1 Access to UBC control registers

The access size must be the same as the control register size. If the sizes are different, a write will not be effected in a UBC register write operation, and a read operation will return an undefined value. UBC control register contents cannot be transferred to a floating-point register using a floating-point memory load instruction.

When a UBC control register is updated, software is responsible for ensuring that the execution of subsequent instructions occurs subject to the updated state.

For the ST40-100, ST40-200, ST40-400 and ST40-500 series, this can be achieved by executing at least 16 instructions before executing any instruction that matches either the old or new programming of the UBC. Alternatively, any of the methods listed in [Section 3.15.2: Achieving coherency \(ST40-100, ST40-200, ST40-400 and ST40-500 series cores\) on page 106](#) may be used.

For the ST40-300 series, any of the methods listed in [Section 3.15.3: Achieving coherency \(ST40-300 series cores\) on page 106](#) may be used.

12.3.2 Break address register A (UBC.BARA)

Table 140. UBC.BARA register description

| UBC.BARA | | | | |
|---------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Break address | [31:0] | 32 | Break address A31 to A0 (BAA31 to BAA0) | RW |
| | Operation | | These bits hold the virtual address (bits 31 to 0) used in the channel A break conditions. | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |

12.3.3 Break ASID register A (UBC.BASRA)

Table 141. UBC.BASRA register description

| UBC.BASRA | | | | |
|---------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Break address | [7:0] | 8 | Break ASID A7 to A0 (BASA7 to BASA0) | RW |
| | Operation | | These bits hold the ASID (bits 7 to 0) used in the channel A break conditions. | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |

12.3.4 Break address mask register A (UBC.BAMRA)

Table 142. UBC.BAMRA register description

| UBC.BAMRA | | | | |
|-------------------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| Reserved | [7:4] | 4 | | RES |
| | Operation | | Reserved | |
| | Reset | | Undefined | |
| Break ASID mask A | [2] | 32 | Break ASID mask A (BASMA) | RW |
| | Operation | | Specifies whether all bits of the channel A break ASID (BASA7 to BASA0) are to be masked. 0: All UBC.BASRA bits are included in break conditions 1: No UBC.BASRA bits are included in break conditions | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |

Table 142. UBC.BAMRA register description (continued)

| UBC.BAMRA | | | | |
|-----------------------------|------------|--|--|--------------------------|
| Field | Bits | Size | Synopsis | Type |
| Break address mask A2 to A0 | [3], [1:0] | 32 | Break address mask A2 to A0 (BAMA2 to BAMA0) | RW |
| | Operation | | These bits specify which bits of the channel A break address (BAA31 to BAA0) set in UBC.BARA are to be masked. | |
| | | | Bit 3: BAMA2 | Bit 1: BAMA1 |
| | | | Bit 0: BAMA0 | |
| | | | 0 | 0 |
| | | | 0 | 0 |
| | | | 0 | 1 |
| | | | 0 | 1 |
| | | | 0 | 1 |
| | | | 1 | 0 |
| | | | 1 | 0 |
| | | | 1 | 1 |
| | | | Any | Reserved (cannot be set) |
| | Reset | Undefined at power-on reset, held at manual reset. | | |

12.3.5 Break bus cycle register A (UBC.BBRA)

Break bus cycle register A (UBC.BBRA) is a 16-bit read/write register that sets three conditions:

- instruction access/operand access
- read/write
- operand size

from among the channel A break conditions.

Table 143. UBC.BBRA register description

| UBC.BBRA | | | | |
|-----------------------------------|-----------|--|--|---|
| Field | Bits | Size | Synopsis | Type |
| Reserved | [15:7] | 8 | | RW |
| | Operation | | Reserved | |
| | Reset | | Undefined | |
| Instruction access/operand access | [5:4] | 2 | Instruction access/operand access select A (IDA1, IDA0) | RW |
| | Operation | These bits specify whether an instruction access cycle or an operand access cycle is used as the bus cycle in the channel A break conditions | | |
| | | IDA1 | IDA0 | |
| | | 0 | 0 | Condition comparison is not performed (Initial value) |
| | | 0 | 1 | Instruction access cycle is used as break condition |
| | | 1 | 0 | Operand access cycle is used as break condition |
| | | 1 | 1 | Instruction access cycle or operand access cycle is used as break condition |
| | Reset | | 0 at power-on reset, held at manual reset and during standby. | |
| Read/write select | [3:2] | 2 | Read/write select A (RWA1, RWA0) | RW |
| | Operation | These bits specify whether a read cycle or write cycle is used as the bus cycle in the channel A break conditions. | | |
| | | RWA1 | RWA0 | |
| | | 0 | 0 | Condition comparison is not performed (initial value) |
| | | 0 | 1 | Read cycle is used as break condition |
| | | 1 | 0 | Write cycle is used as break condition |
| | | 1 | 1 | Read or write cycle is used as break condition |
| | Reset | | 0 at power-on reset, held at manual reset and during standby mode. | |

Table 143. UBC.BBRA register description (continued)

| UBC.BBRA | | | | | | |
|---------------------|--------------------------------|------|--|------|------|--|
| Field | Bits | Size | Synopsis | | | Type |
| Operand size select | [6], [1:0] | 3 | Operand size select A (SZA2, SZA1, SZA0) | | | RW |
| | Operation | | These bits select the operand size of the bus cycle used as a channel A break condition. | | | |
| | | | SZA2 | SZA1 | SZA0 | |
| | | | 0 | 0 | 0 | Operand size is not included in break conditions (initial value) |
| | | | 0 | 0 | 1 | Byte access is used as break condition |
| | | | 0 | 1 | 0 | Word access is used as break condition |
| | | | 0 | 1 | 1 | Long-word access is used as break condition |
| | | | 1 | 0 | 0 | Quad-word access is used as break condition |
| | | | 1 | 0 | 1 | Reserved (cannot be set) |
| | | | 1 | 1 | any | Reserved (cannot be set) |
| | Power-on reset Manual reset | | 0 at power-on reset, held at manual reset and during standby. | | | |

12.3.6 Break address register B (UBC.BARB)

UBC.BARB is the channel B break address register. The bit configuration is the same as for UBC.BARA.

12.3.7 Break ASID register B (UBC.BASRB)

UBC.BASRB is the channel B break ASID register. The bit configuration is the same as for UBC.BASRA.

12.3.8 Break address mask register B (UBC.BAMRB)

UBC.BAMRB is the channel B break address mask register. The bit configuration is the same as for UBC.BAMRA.

12.3.9 Break data register B (UBC.BDRB)

Table 144. UBC.BDRB register description

| UBC.BDRB | | | | |
|------------|-----------|--|--------------------------------------|------|
| Field | Bits | Size | Synopsis | Type |
| Break data | [31:0] | 32 | Break data A31 to A0 (BDB31 to BDB0) | RW |
| | Operation | These bits hold the data (bits 31 to 0) to be used in the channel B break conditions | | |
| | Reset | Undefined at power-on reset, held at manual reset. | | |

12.3.10 Break data mask register B (UBC.BDMRB)

Table 145. UBC.BDMRB register description

| UBC.BDMRB | | | | |
|------------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| Break data | [31:0] | 32 | Break data mask B31 to B0 (BDMB31 to BDMB0) | RW |
| | Operation | | These bits specify whether the corresponding bit of the channel B break data (BDB31 to BDB0) set in UBC.BDRB is to be masked. 0: Channel B break data bit BDB[n] is included in break conditions 1: Channel B break data bit BDB[n] is masked, and not included in break conditions | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |

Note: When the data bus value is included in the break conditions, the operand size should be specified. When byte size is specified, set the same data in bits 15 to 8 and 7 to 0 of UBC.BDRB and UBC.BDMRB.

12.3.11 Break bus cycle register B (UBC.BBRB)

UBC.BBRB is the channel B bus break register. The bit configuration is the same as for UBC.BBRA.

12.3.12 Break control register (UBC.BRCR)

The break control register (UBC.BRCR) is a 16-bit read/write register that specifies:

- whether channels A and B are to be used as 2 independent channels or in a sequential condition
- whether the break is to be effected before or after instruction execution
- whether the UBC.BDRB register is to be included in the channel B break conditions
- whether the user break debug function is to be used. UBC.BRCR also contains condition match flags. The CMFA, CMFB, and UBDE bits in UBC.BRCR are initialized to 0 by a power-on reset, but retain their value in standby mode. The value of the PCBA, DBEB, PCBB, and SEQ bits is undefined after a power-on reset or manual reset, so these bits should be initialized by software as necessary.

Table 146. UBC.BRCR register description

| UBC.BRCR | | | | |
|----------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| CMFA | [15] | 1 | Condition match flag A | RW |
| | Operation | | Set to 1 when a break condition set for channel A is satisfied. This flag remains set until cleared by a write to the register or a power-on reset. It retains its value while in standby mode | |
| | Reset | | 0 at power-on reset, held at manual reset. | |

Table 146. UBC.BRCR register description (continued)

| UBC.BRCR | | | | |
|----------|-----------|------|---|------|
| Field | Bits | Size | Synopsis | Type |
| CMFB | [14] | 1 | Condition match flag B | RW |
| | Operation | | Set to 1 when a break condition set for channel B is satisfied. This flag remains set until cleared by a write to the register. | |
| | Reset | | 0 at power-on reset, held at manual reset. | |
| Reserved | [13:11] | 3 | | RES |
| | Operation | | Reserved | |
| | Reset | | Undefined | |
| PCBA | [10] | 1 | Instruction access break select A | RW |
| | Operation | | Specifies whether a channel A instruction access cycle break is to be effected before or after the instruction is executed. This bit is not initialized by a power-on reset or manual reset. 0: Channel A PC break is effected before instruction execution 1: Channel A PC break is effected after instruction execution | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |
| Reserved | [9:8] | 2 | | RES |
| | Operation | | Reserved | |
| | Reset | | Undefined | |
| DBEB | [7] | 1 | Data break enable B | RW |
| | Operation | | Specifies whether the data bus condition is to be included in the channel B break conditions. This bit is not initialized by a power-on reset or manual reset. 0: Data bus condition is not included in channel B conditions 1: Data bus condition is included in channel B conditions | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |
| PCBB | [6] | 1 | PC break select B | RW |
| | Operation | | Specifies whether a channel B instruction access cycle break is to be effected before or after the instruction is executed. This bit is not initialized by a power-on reset or manual reset. 0: Channel B PC break is effected before instruction execution 1: Channel B PC break is effected after instruction execution | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |
| Reserved | [5:4] | 2 | | RES |
| | Operation | | Reserved | |
| | Reset | | Undefined | |

Table 146. UBC.BRCR register description (continued)

| UBC.BRCR | | | | |
|----------|-----------|------|--|------|
| Field | Bits | Size | Synopsis | Type |
| SEQ | [3] | 1 | Sequence condition select | RW |
| | Operation | | Specifies whether the conditions for channels A and B are to be independent or sequential. This bit is not initialized by a power-on reset or manual reset 0: Channel A and B comparisons are performed as independent conditions 1: Channel A and B comparisons are performed as sequential conditions (channel A then channel B) | |
| | Reset | | Undefined at power-on reset, held at manual reset. | |
| Reserved | [2:1] | 2 | | RES |
| | Operation | | Reserved | |
| | Reset | | Undefined | |
| UBDE | [0] | 1 | User break debug enable | RW |
| | Operation | | Specifies whether the user break debug function (see Section 12.6: User break debug function on page 529) is to be used 0: User break debug function is not used 1: User break debug function is used | |
| | Reset | | 0 at power-on reset, held at manual reset. | |

Note: When the data bus is included in the break conditions, bits IDB1 to 0 in break bus cycle register B (UBC.BBRB) should be set to 10 or 11.

12.4 Operation

12.4.1 Explanation of terms relating to accesses

An instruction access is an access that obtains an instruction. An operand access is any memory access for the purpose of instruction execution. For example, the access to address $PC + \text{dispx2} + 4$ in the instruction `MOV.W @ (disp, PC), Rn` (an access very close to the program counter) is an operand access. The fetching of an instruction from the branch destination when a branch instruction is executed is also an instruction access. As the term “data” is used to distinguish data from an address, the term “operand access” is used in this section.

In the ST40, all operand accesses are treated as either read accesses or write accesses. The following instructions require special attention:

- PREF, OCBP, and OCBWB instructions: treated as read accesses
- MOVCA and OCBI instructions: treated as write accesses
- TAS instruction: treated as 1 read access and 1 write access

The operand accesses for the PREF, OCBP, OCBWB, and OCBI instructions are accesses with no access data.

The ST40 handles all operand accesses as having a data size. The data size can be byte, word, long-word, or quad-word. The operand data size for the PREF, OCBP, OCBWB, MOVCA, and OCBI instructions is treated as long-word.

12.4.2 Explanation of terms relating to instruction intervals

In this section, “1 (2, 3,...) instruction(s) after...”, as a measure of the distance between 2 instructions, is defined as follows. A branch is counted as an interval of 2 instructions.

Example of sequence of instructions with no branch

```

100  Instruction A (0 instructions after instruction A)
102  Instruction B (1 instruction after instruction A)
104  Instruction [c] (2 instructions after instruction A)
106  Instruction [d] (3 instructions after instruction A)

```

Example of sequence of instructions with a branch

```

100  Instruction A:BT/S L200 (0 instructions after
      instruction A)
102  Instruction B (1 instruction after instruction A, 0
      instructions after instruction B)
L200 200 Instruction [c] (3 instructions after instruction A, 2
      instructions after instruction B)
202  Instruction [d] (4 instructions after instruction A, 3
      instructions after instruction B)

```

The example of a sequence of instructions with no branch should be applied when the branch destination of a delayed branch instruction is the instruction itself + 4.

12.4.3 User break operation sequence

The sequence of operations from setting of break conditions to user break exception handling is described below.

1. Specify pre- or post-execution breaking in the case of an instruction access, inclusion or exclusion of the data bus value in the break conditions in the case of an operand access, and use of independent or sequential channel A and channel B break conditions, in the break control register (UBC.BRCR).
2. Set the break addresses in the break address registers for each channel (UBC.BARA, UBC.BARB), the ASIDs corresponding to the break space in the break ASID registers (UBC.BASRA, UBC.BASRB), and the address and ASID masking methods in the break address mask registers (UBC.BAMRA, UBC.BAMRB).
3. If the data bus value is to be included in the break conditions, also set the break data in the break data register (UBC.BDRB) and the data mask in the break data mask register (UBC.BDMRB).
4. Set the break bus conditions in the break bus cycle registers (UBC.BBRA, UBC.BBRB).
5. If either of the instruction access/operand access select group (IDA[n] and IDB[n] bits) or read/write select group (RWA[n] and RWB[n] bits) of the UBC.BBRA OR UBC.BBRB

registers are set to 00, a user break interrupt is not generated on the corresponding channel.

6. Make the UBC.BBRA and UBC.BBRB settings after all other break-related register settings have been completed. If breaks are enabled with UBC.BBRA/UBC.BBRB while the break address, data, or mask register, or the break control register is in the initial state after a reset, a break may be generated inadvertently.
7. The operation when a break condition is satisfied depends on the BL bit (in the CPU's SR register). When the BL bit is 0, exception handling is started and the condition match flag (CMFA, CMFB) for the respective channel is set for the matched condition. When the BL bit is 1, the condition match flag (CMFA, CMFB) for the respective channel is set for the matched condition but exception handling is not started.

The condition match flags (CMFA, CMFB) are set by a branch condition match, but are not reset. Therefore, a memory store instruction should be used on the UBC.BRCR register to clear the flags to 0. See [Section 12.4.6: Condition match flag setting on page 524](#), for the exact setting conditions for the condition match flags.

8. When sequential condition mode has been selected, and the channel B condition is matched after the channel A condition has been matched, a break is effected at the instruction at which the channel B condition was matched. See [Section 12.4.8: Contiguous A and B settings for sequential conditions on page 526](#), for the operation when the channel A condition match and channel B condition match occur close together. With sequential conditions, only the channel B condition match flag is set. When sequential condition mode has been selected, if it is wished to clear the channel A match when the channel A condition has been matched but the channel B condition has not yet been matched, this can be done by writing 0 to the SEQ bit in the UBC.BRCR register.

12.4.4 Instruction access cycle break

1. When an instruction access, read or word setting is made in the break bus cycle register (UBC.BBRA, UBC.BBRB), an instruction access cycle can be used as a break condition. In this case, breaking before or after execution of the relevant instruction can be selected with the PCBA/PCBB bit in the break control register (UBC.BRCR). When an instruction access cycle is used as a break condition, clear the LSB of the break address registers (UBC.BARA, UBC.BARB) to 0. A break will not be generated if this bit is set to 1.
2. When a pre-execution break is specified, the break is effected when it is confirmed that the instruction is to be fetched and executed. Therefore, an overrun-fetched instruction (an instruction that is fetched but not executed when a branch or exception occurs) cannot be used in a break. However, if a TLB miss or TLB protection violation exception occurs at the time of the fetch of an instruction subject to a break, the break exception handling is carried out first. The instruction TLB exception handling is performed when the instruction is re-executed. Also, since a delayed branch instruction and the delay slot instruction are executed as a single instruction, if a pre-execution break is specified for a delay slot instruction, the break will be effected before execution of the delayed branch instruction. However, a pre-execution break cannot be specified for the delay slot instruction for an RTE instruction.
3. With a post-execution break, the instruction set as a break condition is executed, then a break interrupt is generated before the next instruction is executed. When a post-execution break is set for a delayed branch instruction, the delay slot is executed and the break is effected before execution of the instruction at the branch destination (when

the branch is made) or the instruction 2 instructions ahead of the branch instruction (when the branch is not made).

4. When an instruction access cycle is set for channel B, break data register B (UBC.BDRB) is ignored in judging whether there is an instruction access match. Therefore, a break condition specified by the DBEB bit in UBC.BRCR is not executed.

12.4.5 Operand access cycle break

1. In the case of an operand access cycle break, the bits included in address bus comparison vary as shown in [Table 147](#) according to the data size specification in the break bus cycle register (UBC.BBRA, UBC.BBRB).

Table 147. Bits included in address bus comparison

| Data size | Address bits compared |
|---------------------------------|--|
| Quad-word (100) | Address bits A31 to A3 |
| Long-word (011) | Address bits A31 to A2 |
| Word (010) | Address bits A31 to A1 |
| Byte (001) | Address bits A31 to A0 |
| Not included in condition (000) | In quad-word access, address bits A31 to A3 In long-word access, address bits A31 to A2 In word access, address bits A31 to A1 In byte access, address bits A31 to A0 |

2. When data value is included in the break conditions in channel B, set the DBEB bit in the break control register (UBC.BRCR) to 1. In this case, break data register B (UBC.BDRB) and break data mask register B (UBC.BDMRB) settings are necessary in addition to the address condition. A user break interrupt is generated when all 3 conditions (address, ASID, and data) are matched. When a quad-word access occurs, the 64-bit access data is divided into an upper 32 bits and lower 32 bits, and interpreted as two 32-bit data units. A break is generated if either of the 32-bit data units satisfies the data match condition.

Set the IDB1 and IDB0 bits in break bus cycle register B (UBC.BBRB) to 10 or 11. When byte data is specified, the same data should be set in the 2 bytes comprising bits 15 to 8 and bits 7 to 0 in break data register B (UBC.BDRB) and break data mask register B (UBC.BDMRB). When word or byte is set, bits 31 to 16 of UBC.BDRB and UBC.BDMRB are ignored.
3. When the DBEB bit in the break control register (UBC.BRCR) is set to 1, a break is not generated by an operand access with no access data (an operand access in a PREF, OCBP, OCBWB, or OCBI instruction).

12.4.6 Condition match flag setting

Instruction access with post-execution condition, or operand access

The flag is set when execution of the instruction that causes the break is completed. As an exception to this, however, in the case of an instruction with more than 1 operand access the flag may be set on detection of the match condition alone, without waiting for execution of the instruction to be completed.

Example 1

```
100      BT L200 (branch performed)
102      Instruction (operand access break on channel A): flag not
        set
```

Example 2

```
110      FADD (FPU exception)
112      Instruction (operand access break on channel A): flag not
        set
```

Instruction access with pre-execution condition

The flag is set when the break match condition is detected.

Example 1

```
110      Instruction (pre-execution break on channel A): flag set
112      Instruction (pre-execution break on channel B): flag not set
```

Example 2

```
110      Instruction (pre-execution break on channel B, instruction
        access TLB miss): flag set
```

12.4.7 Program counter (PC) value saved

1. When instruction access (pre-execution) is set as a break condition, the program counter (PC) value saved to SPC in user break interrupt handling is the address of the instruction at which the break condition match occurred. In this case, a user break interrupt is generated and the fetched instruction is not executed.
2. When instruction access (post-execution) is set as a break condition, the program counter (PC) value saved to SPC in user break interrupt handling is the address of the instruction to be executed after the instruction at which the break condition match occurred. In this case, the fetched instruction is executed, and a user break interrupt is generated before execution of the next instruction.
3. When an instruction access (post-execution) break condition is set for a delayed branch instruction, the delay slot instruction is executed and a user break is effected before execution of the instruction at the branch destination (when the branch is made) or the instruction 2 instructions ahead of the branch instruction (when the branch is not made). In this case, the PC value saved to SPC is the address of the branch destination (when the branch is made) or the instruction following the delay slot instruction (when the branch is not made).
4. When operand access (address only) is set as a break condition, the address of the instruction to be executed after the instruction at which the condition match occurred is saved to SPC.
5. When operand access (address + data) is set as a break condition, execution of the instruction at which the condition match occurred is completed. A user break interrupt is generated before execution of instructions from 1 instruction later to 4 instructions later. It is not possible to specify at which instruction, from 1 later to 4 later, the interrupt will be generated. The start address of the instruction after the instruction for which execution is completed at the point at which user break interrupt handling is started is saved to SPC.
6. If an instruction between 1 instruction later and 4 instructions later causes another exception, control is performed as follows. Designating the exception caused by the break as exception 1, and the exception caused by an instruction between 1 instruction later and 4 instructions later as exception 2, the fact that memory updating and register updating that essentially cannot be performed by exception 2 cannot be performed is guaranteed irrespective of the existence of exception 1. The program counter value saved is the address of the first instruction for which execution is suppressed.

Whether exception 1 or exception 2 is used for the exception jump destination and the value written to the exception register (EXPEVT, INTEVT) is not guaranteed. However, if exception 2 is from a source not synchronized with an instruction (external interrupt or peripheral module interrupt), exception 1 is used for the exception jump destination and the value written to the exception register (EXPEVT, INTEVT).

12.4.8 Contiguous A and B settings for sequential conditions

When channel A match and channel B match timings are close together, a sequential break may not be guaranteed. Rules relating to the guaranteed range are given below.

Instruction access matches on both channel A and channel B

| | |
|---|--|
| Instruction B is 0 instructions after instruction A | Equivalent to setting the same address. Do not use this setting. |
| Instruction B is 1 instruction after instruction A | Sequential operation is not guaranteed. |
| Instruction B is 2 or more instructions after instruction A | Sequential operation is guaranteed. |

Instruction access match on channel A, operand access match on channel B

| | |
|---|---|
| Instruction B is 0 or 1 instruction after instruction A | Sequential operation is not guaranteed. |
| Instruction B is 2 or more instructions after instruction A | Sequential operation is guaranteed. |

Operand access match on channel A, instruction access match on channel B

| | |
|---|---|
| Instruction B is 0 to 3 instructions after instruction A | Sequential operation is not guaranteed. |
| Instruction B is 4 or more instructions after instruction A | Sequential operation is guaranteed. |

Operand access matches on both channel A and channel B

Do not make a setting such that a single operand access will match the break conditions of both channel A and channel B. There are no other restrictions. For example, sequential operation is guaranteed even if 2 accesses within a single instruction match channel A and channel B conditions in turn.

12.5 Usage notes

1. Do not execute a post-execution instruction access break for the SLEEP instruction. Do not execute a post-execution instruction access break on a branch instruction that has the SLEEP instruction in its delay slot.
2. Do not make an operand access break setting between 1 and 3 instructions before a SLEEP instruction.
3. The value of the BL bit referenced in a user break exception depends on the break setting, as follows.
 - a) Pre-execution instruction access break: the BL bit value before the executed instruction is referenced.
 - b) Post-execution instruction access break: the OR of the BL bit values before and after the executed instruction is referenced.
 - c) Operand access break (address or data): the BL bit value after the executed instruction is referenced.
 - d) Instruction that modifies the BL bit: see [Table 148](#).

Table 148. Break setting for an instruction that modifies the BL bit

| SL.BL | Pre-execution instruction access | Post-execution instruction access | Operand access (address/data) |
|--------|----------------------------------|-----------------------------------|-------------------------------|
| 0 to 0 | Accepted | Accepted | Accepted |
| 1 to 0 | Masked | Masked | Accepted |
| 0 to 1 | Accepted | Masked | Masked |
| 1 to 1 | Masked | Masked | Masked |

- e) RTE delay slot: the BL bit value before execution of a delay slot instruction is the same as the BL bit value before execution of an RTE instruction. The BL bit value after execution of a delay slot instruction is the same as the first BL bit value for the

first instruction executed on returning by means of an RTE instruction (the same as the value of the BL bit in SSR before execution of the RTE instruction).

- f) If an interrupt or exception is accepted with the BL bit cleared to 0, the value of the BL bit before execution of the first instruction of the exception handling routine is 1.
4. If channels A and B both match independently at virtually the same time, and, as a result, the SPC value is the same for both user break interrupts, only 1 user break interrupt is generated, but both the CMFA bit and the CMFB bit are set.

For example:

110 Instruction (post-execution instruction break on
channel A) →SPC = 112, CMFA = 1

```
112      Instruction (pre-execution instruction break on
      channel B) →SPC = 112, CMFB = 1
```

5. The PCBA or PCBB bit in UBC.BRCR is invalid for an operand access break setting.
6. When the SEQ bit in UBC.BRCR is 1, the internal sequential break state is initialized by a channel B condition match.

For example:

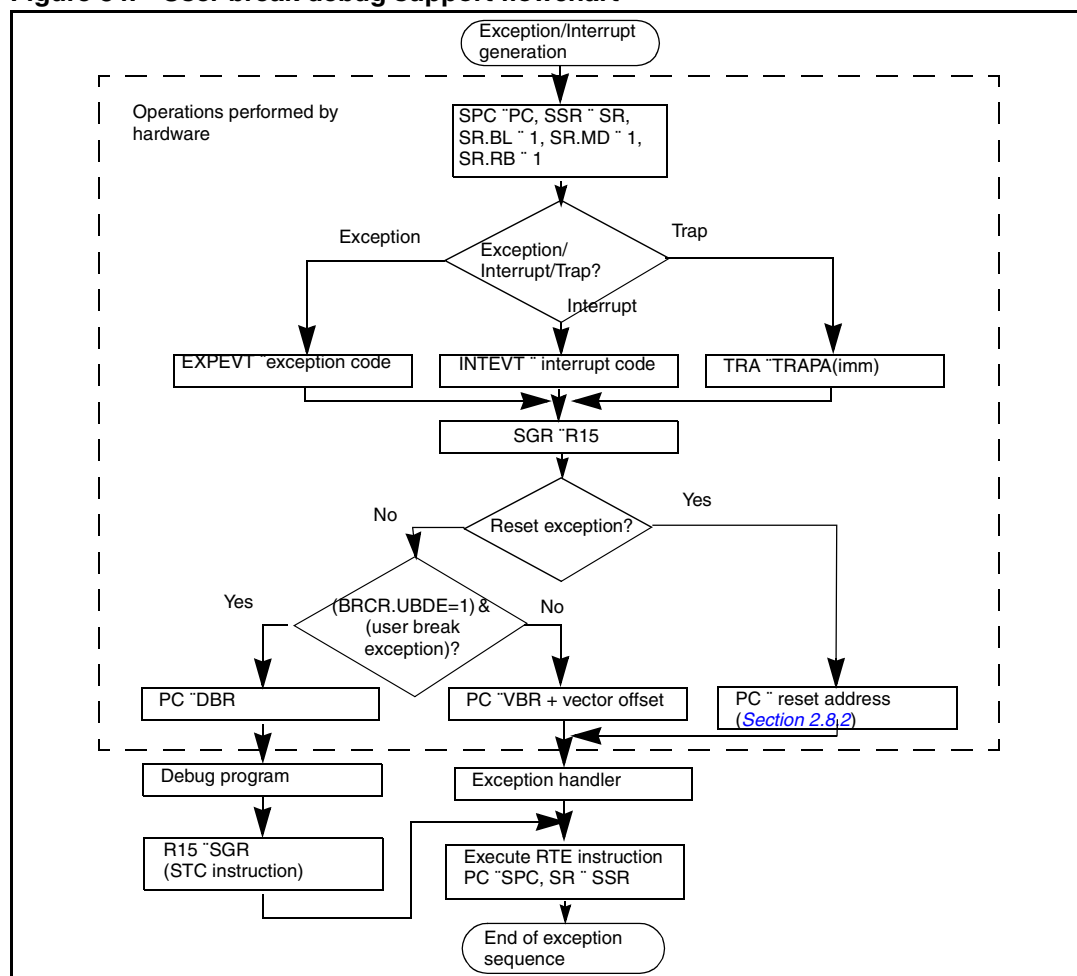
$A \rightarrow A \rightarrow B$ (user break generated) $B \rightarrow$ (no break generated)

7. In the event of contention between a re-execution type exception and a post-execution break in a multistep instruction, the re-execution type exception is generated. In this case, the CMF bit may or may not be set to 1 when the break condition occurs.
8. A post-execution break is classified as a completion type exception. Consequently, in the event of contention between a completion type exception and a post-execution break, the post-execution break is suppressed in accordance with the priorities of the 2 events. For example, in the case of contention between a TRAPA instruction and a post-execution break, the user break is suppressed. However, in this case, the CMF bit is set by the occurrence of the break condition.

12.6 User break debug function

The user break debug function enables the processing used in the event of a user break exception to be changed. When a user break exception occurs, if the UBDE bit is set to 1 in the UBC.BRCR register, the DBR register value will be used as the branch destination address instead of [VBR + offset]. The value of R15 is saved in the SGR register regardless of the value of the UBDE bit in the UBC.BRCR register or the kind of exception event. A flowchart of the user break debug support function is shown in [Figure 34](#).

Figure 34. User break debug support flowchart



12.7 Examples of use

12.7.1 Instruction access cycle break condition settings

Independent channel A channel B mode: user break interrupt generated

Under these conditions a user break is generated after execution of the instruction at address 0x00000404 with ASID = 0x80, or before execution of an instruction at addresses 0x00008000 to 0x000083FE with ASID = 0x70.

Register settings

UBC.BASRA = 0x80

UBC.BARA = 0x00000404

UBC.BAMRA = 0x00

UBC.BBRA = 0x0014

UBC.BASRB = 0x70

UBC.BARB = 0x00008010

UBC.BAMRB = 0x01

UBC.BBRB = 0x0014

UBC.BDRB = 0x00000000

UBC.BDMRB = 0x00000000

UBC.BRCR = 0x0400

Conditions set

Independent channel A channel B mode.

Channel A

ASID: 0x80

Address: 0x00000404

Address mask: 0x00

Bus cycle: instruction access (post-instruction- execution), read (operand size not included in conditions)

Channel B

ASID: 0x70

Address: 0x00008010

Address mask: 0x01

Data: 0x00000000

Data mask: 0x00000000

Bus cycle: instruction access (pre-instruction- execution), read (operand size not included in conditions)

Channel A channel B sequential mode: user break interrupt generated

Under these conditions the instruction at address 0x00037266 with ASID = 0x80 is executed, then a user break is generated before execution of the instruction at address 0x0003722E with ASID = 0x70.

Register settings

UBC.BASRA = 0x80

UBC.BARA = 0x00037226

UBC.BAMRA = 0x00

UBC.BBRA = 0x0016

UBC.BASRB = 0x70

UBC.BARB = 0x0003722E

UBC.BAMRB = 0x00

UBC.BBRB = 0x0016

UBC.BDRB = 0x00000000

UBC.BDMRB = 0x00000000

UBC.BRCR = 0x0008

Conditions set

Channel A channel B sequential mode.

Channel A

ASID: 0x80

Address: 0x00037226

Address mask: 0x00

Bus cycle: instruction access (pre-instruction- execution), read, word

Channel B

ASID: 0x70

Address: 0x0003722E

Address mask: 0x00

Data: 0x00000000

Data mask: 0x00000000

Bus cycle: instruction access (pre-instruction- execution), read, word

Independent channel A channel B mode: user break interrupts not generated

Under these conditions a user break interrupt is not generated on channel A since the instruction access is not a write cycle.

A user break interrupt is not generated on channel B since instruction access is performed on an even address.

Register settings

UBC.BASRA = 0x80

UBC.BARA = 0x00027128

UBC.BAMRA = 0x00

UBC.BBRA = 0x001A

UBC.BASRB = 0x70

UBC.BARB = 0x00031415

UBC.BAMRB = 0x00

UBC.BBRB = 0x0014

UBC.BDRB = 0x00000000

UBC.BDMRB = 0x00000000

UBC.BRCR = 0x0000

Conditions set

Independent channel A and channel B mode.

Channel A

ASID: 0x80

Address: 0x00027128

Address mask: 0x00

Bus cycle: CPU, instruction access (pre-instruction- execution), write, word

Channel B

ASID: 0x70

Address: 0x00031415

Address mask: 0x00

Data: 0x00000000

Data mask: 0x00000000

Bus cycle: CPU, instruction access (pre-instruction- execution), read (operand size not included in conditions)

12.7.2 Operand access cycle break condition settings

Under these conditions on channel A, a user break interrupt is generated in the event of a long-word read at address 0x00123454, a word read at address 0x00123456, or a byte read at address 0x00123456, with ASID = 0x80.

On channel B, a user break interrupt is generated when 0xA512 is written by word access to any address from 0x000AB000 to 0x000ABFFE with ASID = 0x70.

Register settings

UBC.BASRA = 0x80

UBC.BARA = 0x00123456

UBC.BAMRA = 0x00

UBC.BBRA = 0x0024

UBC.BASRB = 0x70

UBC.BARB = 0x000ABCDE

UBC.BAMRB = 0x02

UBC.BBRB = 0x002A

UBC.BDRB = 0x0000A512

UBC.BDMRB = 0x00000000

UBC.BRCR = 0x0080

Conditions set

Independent channel A channel B mode.

Channel A

ASID: 0x80

Address: 0x00123456

Address mask: 0x00

Bus cycle: operand access, read (operand size not included in conditions)

Channel B

ASID: 0x70

Address: 0x000ABCDE

Address mask: 0x02

Data: 0x0000A512

Data mask: 0x00000000

Bus cycle: operand access, write, word data break enabled

12.7.3 User break controller stop function

The clock supply to the UBC can be stopped by a signal exported from the core which is typically driven from a control register in a clock controller. For further details of how to do this refer to the product's *Core Support Peripheral Architecture Manual*. The purpose of this is to minimize power dissipation when the chip is operating.

Note: If you use this function, you cannot use the user break controller.

Appendix A Address list

Table 149. Address list

| Register | P4 address ⁽¹⁾ | Size | Power-on reset | Manual reset | Sleep | Standby | Series ⁽²⁾ |
|----------|---------------------------|------|----------------------------|----------------------------|-------|---------|-----------------------|
| PTEH | 0xFF00 0000 | 32 | Undefined | Undefined | Held | Held | 100, 200, 300, 500 |
| PTL | 0xFF00 0004 | 32 | Undefined | Undefined | Held | Held | |
| TTB | 0xFF00 0008 | 32 | Undefined | Undefined | Held | Held | |
| TEA | 0xFF00 000C | 32 | Undefined | Held | Held | Held | |
| MMUCR | 0xFF00 0010 | 32 | 0x0000 0000 | 0x0000 0000 | Held | Held | |
| BASRA | 0xFF00 0014 | 8 | Undefined | Held | Held | Held | |
| BASRB | 0xFF00 0018 | 8 | Undefined | Held | Held | Held | |
| CCR | 0xFF00 001C | 32 | 0x0000 0000 | 0x0000 0000 | Held | Held | |
| TRA | 0xFF00 0020 | 32 | Undefined | Undefined | Held | Held | |
| EXPEVT | 0xFF00 0024 | 32 | 0x0000 0000 | 0x0000 0020 | Held | Held | |
| INTEVT | 0xFF00 0028 | 32 | Undefined | Undefined | Held | Held | |
| QACR0 | 0xFF00 0038 | 32 | Undefined | Undefined | Held | Held | |
| QACR1 | 0xFF00 003C | 32 | Undefined | Undefined | Held | Held | |
| PASCR | 0xFF00 0070 | 32 | 0x0000 0000 or 0x0000 00FF | 0x0000 0000 or 0x0000 00FF | Held | Held | 300 |
| RAMCR | 0xFF00 0074 | 32 | 0x0000 00C0 | 0x0000 00C0 | Held | Held | 300 |
| IRMCR | 0xFF00 0078 | 32 | 0x0000 0000 | 0x0000 0000 | Held | Held | 300 |
| BARA | 0xFF20 0000 | 32 | Undefined | Held | Held | Held | |
| BAMRA | 0xFF20 0004 | 8 | Undefined | Held | Held | Held | |
| BBRA | 0xFF20 0008 | 16 | 0x0000 | Held | Held | Held | |
| BARB | 0xFF20 000C | 32 | Undefined | Held | Held | Held | |
| BAMRB | 0xFF20 0010 | 8 | Undefined | Held | Held | Held | |
| BBRB | 0xFF20 0014 | 16 | 0x0000 | Held | Held | Held | |
| BDRB | 0xFF20 0018 | 32 | Undefined | Held | Held | Held | |
| BDMRB | 0xFF20 001C | 32 | Undefined | Held | Held | Held | |
| BRCR | 0xFF20 0020 | 16 | 0x0000 ⁽³⁾ | Held | Held | Held | |

1. Many of these registers may also be accessed through an address translation from another virtual address region. This is described in [Section 3.7.2: Resources accessible through P4 and through translations on page 54](#). The restrictions applying to such translations are described in [Section 3.15.1: Coherency requirements on page 105](#) and [Section 4.7: Cache state coherency on page 145](#).

2. If a cell in this column is blank, it means that register is present on all variants. Otherwise the variants having the register are listed.

3. Includes undefined bits, see the description of the actual module for full details.

Note: *The address map for peripheral devices is contained in the system manual for the part.*

Revision history

Table 150. Document revision history

| Date | Revision | Changes |
|-------------|----------|--|
| 18-Oct-2011 | 12 | Minor modifications throughout. Updated Table 103: Floating-point control instructions (only if FPU present) on page 188 . |
| 28-Nov-2008 | K | Moved to new template. Expanded Section 1.1: ST40 core features on page 12 . Revisions made to Section 3.4: Physical address space on page 37 . Minor revision to Table 11: Summary of virtual to physical translation in each virtual address region on page 47 . Revisions made to Section 3.9: Unified TLB (UTLB) on page 69 . Footnotes added to Table 27: Fields in each UTLB data array entry on page 75 . Revisions made to Section 3.11: Privileged mapping buffer (PMB) on page 88 . Footnote added to Table 40: Fields in each PMB address array entry on page 90 . Footnote added to Table 42: Fields in each PMB data array entry on page 92 . Major revisions made to Section 3.15.1: Coherency requirements on page 105 . Revisions made to Table 48: Coherency requirements for access to MMU state on page 106 . New Section 3.15.2: Achieving coherency (ST40-100, ST40-200, ST40-400 and ST40-500 series cores) on page 106 and Section 3.15.3: Achieving coherency (ST40-300 series cores) on page 106 replace “Achieving coherency”. Major revisions made to Section 3.16: Side effects of instruction pre-fetch on page 107 including new Section 3.16.1: Inline fetching near ends of memory regions on page 107 and Section 3.16.2: Branches following exceptions on page 108 . Revisions made to Table 80: Coherency requirements for access to cache state on page 145 . Revisions made to layout of Section 5.5.2: General exceptions on page 156 . (Several pages of text replaced by a table.) Minor formatting changes made to MAC.L @Rm+, @Rn+ on page 358 and MAC.W @Rm+, @Rn+ on page 359 . Revisions made to layout of Table 148: Break setting for an instruction that modifies the BL bit on page 527 . |

Table 150. Document revision history (continued)

| Date | Revision | Changes |
|-------------|----------|--|
| 21-Nov-2006 | J | <p>Removed obsolete footnote from Table 7: ST40 MMU variants on page 33.</p> <p>Revisions to Section 3.4: Physical address space on page 37.</p> <p>Added Physical address generation on page 51.</p> <p>Revisions to Section 3.7: P4 address region on page 54.</p> <p>Revisions to Table 15: P4 virtual address region on page 55.</p> <p>Addition to Table 18: PTEL register description on page 59.</p> <p>Minor revision to Table 22: PASCRC description on page 67.</p> <p>Revision to Hardware refill from the UTLB on page 80.</p> <p>Revision to Hardware refill from the PMB on page 80.</p> <p>Minor revision to Table 40: Fields in each PMB address array entry on page 90.</p> <p>Revisions to Section 3.14.2: Coherency mechanisms on ST40-300 series cores on page 103.</p> <p>Minor revision to Table 48: Coherency requirements for access to MMU state on page 106.</p> <p>Revisions to Section 3.15.2: Achieving coherency (ST40-100, ST40-200, ST40-400 and ST40-500 series cores) on page 106.</p> <p>Added Section 3.16: Side effects of instruction pre-fetch on page 107.</p> <p>Moved details of ICBI from Section 4.3.8: Explicit cache controls on page 127 to new Section 4.4.4: Explicit cache controls on page 130.</p> <p>Added Section 4.4.4: Explicit cache controls on page 130 for Instruction Cache.</p> <p>Minor revisions to Section 4.6: Store queues (SQs) on page 140.</p> <p>Minor revisions to Section 4.6.3: SQ writes on page 141.</p> <p>Minor revisions to Section 5.5.2: General exceptions on page 156.</p> <p>Minor revisions to Section 6.2: Rounding on page 166.</p> <p>Minor formatting changes throughout Chapter 9: Instruction descriptions on page 213.</p> <p>Minor revisions to the following instructions: BRAFP Rn on page 223, BSRFP Rn on page 225, FRCHG on page 308, FSCHG on page 310, JMP @Rn on page 329, JSR @Rn on page 330, MOVCA.L R0, @Rn on page 396, and RTS on page 417.</p> <p>Revisions made to Section 11.1: Basic pipeline structure on page 497.</p> <p>Revisions made to Section 11.2: Issue constraints on page 498.</p> <p>Revisions made to Table 136: Integer instructions on page 500.</p> <p>Revisions made to Table 137: Floating point instructions on page 506.</p> |

Table 150. Document revision history (continued)

| Date | Revision | Changes |
|-------------|----------|---|
| 15-Mar-2006 | I | <p>Revision made to Table 2: Initial register values on page 18.</p> <p>Revision made to Table 6: SR register description on page 25.</p> <p>Revisions to Section 2.5: Memory-mapped control registers on page 28.</p> <p>Added new Section 2.8.2: Reset address on page 29.</p> <p>Revisions to Section 2.8.3: Exception-handling state on page 29.</p> <p>Major revision to Chapter 3: Memory management unit (MMU) on page 32.</p> <p>Major revision to Chapter 4: Caches on page 114.</p> <p>Revisions to Section 5.1: Register descriptions on page 146.</p> <p>Revisions to Section 5.2.2: Exception handling vector addresses on page 148.</p> <p>Minor revisions to Section 5.3: Exception types and priorities on page 149.</p> <p>Revisions to Section 5.5: Description of exceptions on page 153.</p> <p>Minor revisions to Section 5.5.2: General exceptions on page 156.</p> <p>Minor revisions to Section 5.6: Usage notes on page 161.</p> <p>Revisions to Section 6.5: 64-bit data transfer on page 169 (formerly “Pair single precision data transfer”).</p> <p>Added new sections Section 6.5.1: Register-to-register transfers on page 169 and Section 6.5.2: Memory transfers on page 170.</p> <p>Revisions made to Section 7.1: Execution environment on page 172.</p> <p>Introductory text added for Table 99: System control instructions on page 183.</p> <p>Introductory text added for Table 102: Floating-point data transfer instructions (only if FPU present) on page 187.</p> <p>Introductory text added for Table 103: Floating-point control instructions (only if FPU present) on page 188.</p> <p>Revisions made to Table 113: Conversion to floating-point register formats on page 195.</p> <p>Addition made to Section 8.4: Architectural state on page 199.</p> <p>Added Section 8.5.2: Instruction fetch on page 202.</p> <p>Revisions made to Section 8.5.3: Reading memory on page 202.</p> <p>Revisions made to Section 8.5.4: Prefetching memory on page 203.</p> <p>Revisions made to Section 8.5.5: Writing memory on page 204.</p> <p>Added Section 8.6: Sleep and synchronization operations on page 205.</p> <p>Major revisions made to Section 8.9: Abstract sequential model on page 209.</p> <p>Minor revisions to many of the instructions in Chapter 9: Instruction descriptions on page 213.</p> <p>In Chapter 9, the following new instructions have been added: FPCHG on page 307, ICBI @Rn on page 327, and SYNCO on page 464.</p> <p>Renamed Chapter 10 “Performance characteristics” to Chapter 10: ST40-100, 200, 400, 500 performance characteristics on page 474.</p> <p>Revisions made to Section 10.3: Execution cycles and pipeline stalling on page 482.</p> |

Table 150. Document revision history (continued)

| Date | Revision | Changes |
|-------------|-----------|--|
| 15-Mar-2006 | I (cont.) | Added new Chapter 11: ST40-300 performance characteristics on page 497 . Revisions to Section 12.3.1: Access to UBC control registers on page 513 . |
| 18-Jan-2006 | H | No previous revision history available |

Index

A

ADD 210, 213-214, 254
 ADDC 215
 Address 510, 512-515, 517, 520-523, 525-527,
 529-533
 ADDV 216
 AND 193, 202-204, 217-219
 AND.B 219

B

BF 220-221
 BRA 222
 BRAF 223
 Branch instruction 520, 523, 525
 Break 510-531, 533
 BSR 224
 BSRF 225
 BT 226-227
 Byte 523

C

Channel 510, 513-523, 526, 528, 530-533
 CMP/GT 260
 Control registers (CR) 513, 518, 521-523

D

Data 512-513, 517-523, 525, 527, 530-533
 DBR 529
 Debug 518, 520, 529
 Delayed branch 521-522, 525
 DIV0S 240
 DIV1 242
 DMULS.L 243
 DMULU.L 244
 DT 245

E

ELSE 197
 Event 528-529, 533
 Exception 521-522, 524-525, 527-529
 EXTS.B 246
 EXTS.W 247
 EXTU.B 248
 EXTU.W 249

F

FABS 207, 250-251, 506
 FADD 207, 211, 252-253, 506
 FCMP/EQ 208, 506
 FCMP/GT 208, 506
 FCNV 208
 FCNVDS 208, 261-262, 506
 FCNVSD 262, 506
 FDIV 207, 264-265, 506
 FIPR 208, 267-268, 506
 FLDI 271-272
 FLDI0 506
 FLDI1 506
 FLDS 270, 506
 FLOAT 208, 273-274, 506
 Floating-point 513
 FMAC 208, 275, 506
 FMOV 279-301, 506
 FMOV.S 284-287, 296-298, 507
 FMUL 207, 302-303, 507
 FNEG 207, 305-306, 507
 FOR 192-194, 197, 200, 202-204
 FPCHG 188, 307, 505, 507
 FPU 199, 206, 253, 256, 259, 262, 265, 268,
 274-276, 303, 309, 312, 314, 318, 321, 324
 FPUDIS 250-253, 255-256, 258-259, 261-262,
 264-265, 267, 270-275, 279-303, 305-312, 314,
 316-318, 320-321, 324, 347-350, 435-436,
 454-455
 FPUExc 212
 FPUL 186, 199, 261-262, 270, 273-274, 309,
 316, 320-321, 349-350, 436, 455
 FRCHG 169, 188, 308, 481, 495, 505, 507
 FRn 314
 FROM 197
 FSCL 169, 309, 507
 FSCL, FPUL 309
 FSCHG 169, 188, 310, 507
 FSQRT 207, 311-312, 507
 FSRRA 169, 314, 507
 FSRRA.S 208, 314
 FSTS 314, 316, 507
 FSUB 207, 317-318, 507
 FTRC 208, 320-321, 507
 FTRV 208, 323-324, 508
 FTRV.S 324
 Function 510, 518, 520, 529
 Bit(i) 194

| | | | |
|--------------------------------------|----------|--|--|
| DataAccessMiss(address) | 201, 203 | FTRC_DL | 208 |
| ExecuteProhibited(address) | 201 | FTRC_SL | 208 |
| FABS_D | 207 | FTRV_S | 208 |
| FABS_S | 207 | InstFetchMiss(address) | 201 |
| FADD_D | 207 | IsLittleEndian() | 201 |
| FADD_S | 207, 212 | MalformedAddress(address) | 201-204 |
| FCMPEQ_D | 208 | MMU() | 201-204 |
| FCMPEQ_S | 208 | OCBI(address) | 205 |
| FCMPGT_D | 208 | OCBP(address) | 205 |
| FCMPGT_S | 208 | OCBWB(address) | 205 |
| FCNV_DS | 208 | PrefetchMemory(address) | 203 |
| FCNV_SD | 208 | PREFO(address) | 203, 205 |
| FDIV_D | 207 | ReadMemoryLown(address) | 203 |
| FDIV_S | 207 | ReadMemoryn(address) | 200, 202 |
| FIPR_S | 208 | ReadMemoryPairn(address) | 202-203 |
| FLOAT_LD | 208 | ReadProhibited(address) | 201-203 |
| FLOAT_LS | 208 | Register(i) | 194 |
| FloatRegister32(i) | 195 | SignExtendn(i) | 194 |
| FloatRegister64(i) | 195 | WriteControlRegister(index, value) | 204 |
| FloatRegisterMatrix32(a) | 195 | WriteMemoryLown(address, value) | 205 |
| FloatRegisterVector32(a) | 195 | WriteMemoryn(address, value) | 204 |
| FloatValue32(r) | 195 | WriteMemoryPairn(address, value) | 204 |
| FloatValue64(r) | 195 | WriteProhibited(address) | 201, 204 |
| FloatValueMatrix32(r) | 195 | ZeroExtendn(i) | 194 |
| FloatValueVector32(r) | 195 | Functions | 510 |
| FMAC_S | 208 | | |
| FMUL_D | 207 | I | |
| FMUL_S | 207 | ICBI | 34, 46, 103, 106, 131, 327-328, 501 |
| FNEG_D | 207 | IF | 197, 202-204, 212 |
| FNEG_S | 207 | ILLSLOT | 210, 220-227, 329-330, 382, 393, 395, 416-417, 432, 466 |
| FpuCauseE() | 206 | Instruction set | 522 |
| FpuCauseI() | 206 | INT | 194 |
| FpuCauseO() | 206 | IRMCR | 68 |
| FpuCauseU() | 206 | ISA | 223 |
| FpuCauseV() | 206 | | |
| FpuCauseZ() | 206 | J | |
| FpuEnableI() | 206 | JMP | 329 |
| FpuEnableO() | 206 | JSR | 330 |
| FpuEnableU() | 206 | | |
| FpuEnableV() | 206 | L | |
| FpuEnableZ() | 206 | LDC | 331-346, 434, 440-445 |
| FpuFlagI() | 206 | LDC.L | 332, 339-346 |
| FpuFlagO() | 206 | LDS | 169, 188, 341, 347-356, 508 |
| FpuFlagU() | 206 | LDS.L | 188, 348, 350, 352, 354, 356, 508 |
| FpuFlagV() | 206 | Load instruction | 513 |
| FpuFlagZ() | 206 | | |
| FpulsDisabled() | 206 | M | |
| FSQRT_D | 207 | MAC.L | 358 |
| FSQRT_S | 207 | | |
| FSRRA_S | 208 | | |
| FSUB_D | 207 | | |
| FSUB_S | 207 | | |

MAC.W 359
 MACH ... 199, 228, 243-244, 351-352, 358-359,
 398-400, 437, 456
 MACL ... 199, 228, 243-244, 353-354, 358-359,
 398-400, 438, 457
 MD 199
 MEM 199-200, 202-204
 Memory 520, 522, 525
 MMU 201-204
 Mode 518, 522, 530-533
 MOV 361-394
 MOV.B 363-372
 MOV.L 373-383
 MOV.W 384-394, 520
 MOVA 395
 MOVCA.L 396
 MOVT 397
 MUL.L 398
 MULS.W 399
 MULU.W 400

N

NEG 401
 NEGC 402
 NOT 194, 203, 404

O

OCBI 205, 327, 405, 520-521, 523
 OCBP 205, 406, 520-521, 523
 OCBWB 205, 407, 520-521, 523
 Operand .510, 516-518, 520-521, 523-527, 530,
 532-533
 OR 193, 202-204, 408-410, 527
 OR.B 410

P

P0 207-208
 PC .199-200, 209-210, 220-227, 382, 393, 395,
 519, 525
 Power-on reset 518-520
 PR 199, 209, 224-225, 330, 355-356, 417, 439,
 458
 PREF 411, 520-521, 523
 PREFO 203, 205

R

RADDERR 202-203
 ReadMemoryPairn 202
 READPROT 202-203

Register 512-514, 516-518, 520-523, 525,
 529-533
 DR 199
 EXPEVT 525
 FPSCR 199, 206-208, 212, 252-253, 256, 259,
 261-262, 264-265, 268, 273-276, 302-303,
 307-308, 310-312, 314, 317-318, 320-321,
 324, 347-348, 435, 454
 FPSCR.CAUSE.E 206
 FPSCR.CAUSE.I 206
 FPSCR.CAUSE.O 206
 FPSCR.CAUSE.U 206
 FPSCR.CAUSE.V 206
 FPSCR.CAUSE.Z 206
 FPSCR.DN 253, 256, 259, 262, 265, 268,
 275-276, 303, 312, 314, 318, 321, 324
 FPSCR.ENABLE.I 206
 FPSCR.ENABLE.O 206
 FPSCR.ENABLE.U 206
 FPSCR.ENABLE.V 206
 FPSCR.ENABLE.Z 206
 FPSCR.FLAG.I 206
 FPSCR.FLAG.O 206
 FPSCR.FLAG.U 206
 FPSCR.FLAG.V 206
 FPSCR.FLAG.Z 206
 FPSCR.FR 308
 FPSCR.RM 252-253, 261-262, 264-265,
 273-275, 302-303, 311-312, 317-318,
 320-321
 FPSCR.SZ 307, 310
 FR 275, 308
 GBR .. 199, 219, 331-334, 336-342, 344-346,
 ... 366, 371, 376, 381, 387, 392, 410, 433,
 440, 443, 446-453, 469, 472
 INTEVT 525
 MTRX 199
 R 218-219, 232, 283, 287, 292, 295, 298, 301,
 365-367, 370-372, 375-376, 380-381,
 386-388, 391-392, 394-396, 409-410,
 468-469, 471-472, 512-513, 529
 Rm ... 213, 215-217, 231, 233-236, 239-240,
 242-244, 246-249, 293-301, 331-356,
 358-361, 363-365, 368-370, 372-375,
 377-380, 383-386, 388-391, 394, 398-402,
 ... 404, 408, 420, 423, 440, 443, 459-463,
 467, 470, 473
 SR 199, 206, 522
 SR.FD 206
 SSR 528
 Registers
 Floating point 513

Program Counter 520, 525
 REPEAT 197
 Reset 518-520, 522
 ROTCL 412
 ROTCR 413
 ROTL 414
 ROTR 415
 RTE 522, 527-528
 RTLBMIS 202-204

S

Section 513, 520-522
 SHAD 420
 SHAL 421
 SHAR 422
 SHLD 423
 SHLL 424-427
 SHLR 428-432
 SLEEP 205, 527
 SLOTFPUDIS 250-253, 255-256, 258-259,
 261-262, 264-265, 267, 270-275, 279-303,
 305-312, 314, 316-318, 320, 324, 347-350,
 435-436, 454-455
 SPC 525, 528
 Standby 518
 Status register field
 ASID . 510, 512, 514, 517, 521, 523, 530-533
 BL 522, 527-528
 STC 433, 446-453
 STC.L 446-453
 STEP 197-198
 STS 188, 435-439, 454-458, 508
 STS.L 188, 454-458, 508
 SUB 319, 459
 SUBC 460
 SUBV 461
 SWAP.B 462
 SWAP.W 463
 SYNCO 34, 67, 103, 205, 464, 504
 SZ 307, 310

T

TAS.B 465
 THROW 198, 202-204, 212
 TLB 522
 TRAPA 466, 528
 TST 467-469
 TST.B 469
 Type 522, 528

U

UNDEFINED 196-197

W

WRITEPROT 204
 WTLBMIS 204

XYZ

XMTRX 323-324
 XOR 193, 470-472
 XOR.B 472
 XTRCT 473

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries ("ST") reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST's terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST'S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS EXPRESSLY APPROVED IN WRITING BY TWO AUTHORIZED ST REPRESENTATIVES, ST PRODUCTS ARE NOT RECOMMENDED, AUTHORIZED OR WARRANTED FOR USE IN MILITARY, AIR CRAFT, SPACE, LIFE SAVING, OR LIFE SUSTAINING APPLICATIONS, NOR IN PRODUCTS OR SYSTEMS WHERE FAILURE OR MALFUNCTION MAY RESULT IN PERSONAL INJURY, DEATH, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE. ST PRODUCTS WHICH ARE NOT SPECIFIED AS "AUTOMOTIVE GRADE" MAY ONLY BE USED IN AUTOMOTIVE APPLICATIONS AT USER'S OWN RISK.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.

Information in this document supersedes and replaces all information previously supplied.

The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2011 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com